Polyspace® Code Prover™

Reference
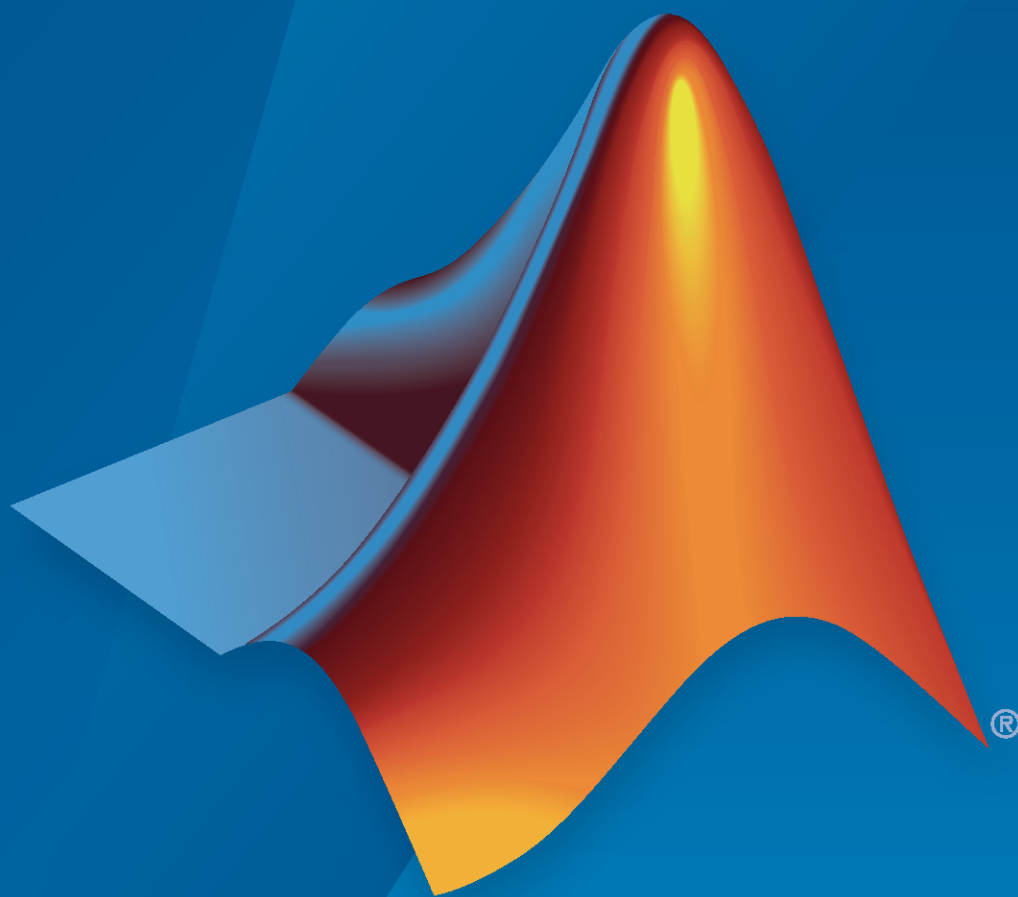
MathWorks®

How to Contact MathWorks

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

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

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

Contents

5 | **Functions, Classes, Methods, Properties, and Apps**

6 | **MISRA C 2012**

7 | **MISRA C++: 2008**

8 | **Code Metrics**

9 | **Custom Coding Rules**

Global Variables

10

Report Components

11

Configuration Parameters

12

Analysis Options

Source code language (`-lang`)

Specify language of source files

Description

Specify the language of your source files. Before specifying other configuration options, choose this option because other options change depending on your language selection.

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

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

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See "Dependencies" on page 1-2 for ways in which the source code language can be automatically determined.

Command line and options file: Use the option `-lang`. See "Command-Line Information" on page 1-3.

Settings

Default: Based on file extensions.

C

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

CPP

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

C-CPP

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

Dependencies

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

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

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

Command-Line Information

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

See Also

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

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

C standard version (`-c-version`)

Specify C language standard followed in source code

Description

Specify the C language standard that you follow in your source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See "Dependencies" on page 1-5 for other options that you must enable.

Command line and options file: Use the option `-c-version`. See "Command-Line Information" on page 1-5.

Why Use This Option

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

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

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

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

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

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

`c90`

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

`c99`

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

`c11`

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

Dependencies

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

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

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

Command-Line Information

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

See Also

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

Topics
"Specify Polyspace Analysis Options"
"C/C++ Language Standard Used in Polyspace Analysis"
"C11 Language Elements Supported in Polyspace"

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

Specify C++ language standard followed in source code

Description

Specify the C++ language standard that you follow in your source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See "Dependencies" on page 1-7 for other options that you must enable.

Command line and options file: Use the option `-cpp-version`. See "Command-Line Information" on page 1-7.

Why Use This Option

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

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

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

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

If the code compiles, use `cpp11` for this option.

Settings

Default: `defined-by-compiler`

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

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

`cpp03`
    The analysis uses the C++03 Standard (ISO/IEC 14882:2003).

cpp11

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

cpp14

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

cpp17

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

Dependencies

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

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

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

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

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

Topics
"Specify Polyspace Analysis Options"
"C/C++ Language Standard Used in Polyspace Analysis"
"C++11 Language Elements Supported in Polyspace"
"C++14 Language Elements Supported in Polyspace"
"C++17 Language Elements Supported in Polyspace"

Target processor type (`-target`)

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

Description

Specify the processor on which you deploy your code.

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

Set Option

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

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

Command line and options file: Use the option `-target`. See "Command-Line Information" on page 1-10.

Why Use This Option

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

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

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

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

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

Settings

Default: i386

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

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

Target	char	short	int	long	long long	float	double	long double[a]	ptr	Default sign of char	endian	Align ment
Targets for IAR Embedded Workbench compiler	See `IAR Embedded Workbench Compiler (-compiler iar-ew)`.											
Targets for MPLAB XC8 C compiler	See `MPLAB XC8 C Compiler (-compiler microchip)`											
Targets for Renesas® compiler	See `Renesas Compiler (-compiler renesas)`.											
Targets for TASKING compiler	See `TASKING Compiler (-compiler tasking)`.											
Targets for Texas Instruments™ compiler	See `Texas Instruments Compiler (-compiler ti)`.											

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

- For targets `i386`, `x86_64` and `m68k`, 80 bits are used for computations, following the practice in common compilers.

- For the target `tms320c3x`, 40 bits are used for computation, following the TMS320C3x specifications.

- If you use a Visual compiler, the size of `long double` used for computations is the same as size of `double`, following the specification of Visual C++® compilers.

b. The M68k family (68000, 68020, and so on) includes the "ColdFire" processor

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

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

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

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

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

Tips

If your processor is not listed, use a similar processor that shares the same characteristics, or create an `mcpu` generic target processor. See `Generic target options`.

You can also create a custom target by explicitly stating sizes of fundamental types and so on with the option `-custom-target`.

Command-Line Information
Parameter: `-target`
Value: `i386` | `sparc` | `m68k` | `powerpc` | `c-167` | `tms320c3x` | `sharc21x61` | `necv850` | `hc08` | `hc12` | `mpc5xx` | `c18` | `x86_64` | `mcpu`
Default: `i386`
Example (Bug Finder): `polyspace-bug-finder -target m68k`
Example (Code Prover): `polyspace-code-prover -target m68k`

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

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

See Also

Polyspace Analysis Options
`-custom-target`

Polyspace Results
`Higher Estimate of Local Variable Size|Lower Estimate of Local Variable Size`

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

Generic target options

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

Description

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

Settings

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

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

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

- *char* [8]
- *short* [16]
- *int* [16]
- *long* [32]

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

Dependency

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

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

Command-Line Options

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

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

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

Option	Description	Available With	Example
-long-long-is-64bits	Define long long as 64 bits, regardless of sign. Alignment is also set to 64 bits.	mcpu	polyspace-code-prover -target mcpu -long-long-is-64bits
-double-is-64bits	Define double and long double as 64 bits, regardless of sign.	mcpu, sharc21x61, hc08, hc12, mpc5xx	polyspace-code-prover -target mcpu -double-is-64bits
-pointer-is-24bits	Define pointer as 24 bits, regardless of sign.	c18	polyspace-code-prover -target c18-pointer-is-24bits
-pointer-is-32bits	Define pointer as 32 bits, regardless of sign.	mcpu	polyspace-code-prover -target mcpu -pointer-is-32bits
-align [32|16|8]	Specifies the largest alignment of struct or array objects to the 32, 16 or 8 bit boundaries. Consequently, the array or struct storage is strictly determined by the size of the individual data objects without member and end padding.	mcpu, hc08, hc12, mpc5xx. Other than mcpu, all targets support only 16 or 32 bits.	polyspace-code-prover -target mcpu -align 16

See also:

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

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

Examples

GCC Toolchains

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

- ARM Ltd's GNU Arm Embedded Toolchain
- HighTec EDV-Systeme
- Linaro® GNU cross-toolchain
- MENTOR® Embedded Sourcery CodeBench

- QNX® Software Development Platform
- Rowley Associates' CrossWorks
- STMicroelectronics® TrueSTUDIO® for STM32
- Texas Instruments Code Composer Studio™
- Wind River® GNU Compiler

Use `polyspace-configure` to trace your build system and extract information about your compiler configuration. The command creates a Polyspace project. To generate an options file that you then pass to Polyspace at the command line, run `polyspace-configure` with `-output-options-file`.

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

- Select the `gnu#.x` compiler that corresponds to your compiler version for `Compiler (-compiler)`.
- Specify your target by using the "Command-Line Options" on page 1-13. For an example of targets you can specify, see "Targets for GCC Based Compilers" on page 1-16.
- Specify your compiler macro definitions with `Preprocessor definitions (-D)`.

Targets for GCC Based Compilers

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

For instance, you can create these targets:

- **Tricore**: Use these options:

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

- **PowerPC**: Use these options:

```
-target mcpu
-int-is-32bits
-long-long-is-64bits
-double-is-64bits
-pointer-is-32bits
-wchar-t-type-is signed-int
```

- **ARM**: Use these options:

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

- **MSP430**: Use these options:

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

Emulate Microchip MPLAB XC16 and XC32 Compilers

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

Compiler	Target Processor Families	Options

MPLAB XC16	PIC24
	
	dsPIC
		```-compiler gnu4.6```
		```-to compile```
		```-D__XC__```
		```-D__XC16__```
		```-target=mcpu```
		```-wchar-t-type-is```
		``` unsigned-int```
		```-align 16```
		```-long-long-is-64bits```

MPLAB XC32	PIC32	```-compiler gnu4.8```
		```-custom-target true,8,2,4,-1,4,8,4,4,8,4,8,1,```
		``` big,unsigned_long,long,int```
		```-D__PIC32M```
		```-D__PIC32MX```
		```-D__PIC32MX__```
		```-D__XC32```
		```-D__XC32__```
		```-D__XC```
		```-D__XC__```
		```-D__mips=32```
		```-D__mips__```
		```-D_mips```
		
		```-compiler gnu4.8```
		```-custom-target```
		``` true,8,2,4,-1,4,8,4,```
		``` 4,8,4,8,1,big,unsigned_long,long,int```
		```-D__PIC32M```
		```-D__PIC32MX```
		```-D__PIC32MX__```
		```-D__XC32```
		```-D__XC32__```
		```-D__XC```
		```-D__XC__```
		```-D__mips=32```
		```-D__mips__```
		```-D_mips```

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

Tips

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

See Also

`Target processor type (-target)`

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

Compiler (-compiler)

Specify the compiler that you use to build your source code

Description

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

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

Set Option

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

Command line and options file: Use the option `-compiler`. See "Command-Line Information" on page 1-25.

Why Use This Option

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

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

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

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

Settings

Default: `generic`

`generic`

Analysis allows only standard syntax.

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

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

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

`gnu3.4`

Analysis allows GCC 3.4 syntax.

`gnu4.6`

Analysis allows GCC 4.6 syntax.

`gnu4.7`

Analysis allows GCC 4.7 syntax.

For unsupported GCC extensions, see "Limitations" on page 1-23.

`gnu4.8`

Analysis allows GCC 4.8 syntax.

For unsupported GCC extensions, see "Limitations" on page 1-23.

`gnu4.9`

Analysis allows GCC 4.9 syntax.

For unsupported GCC extensions, see "Limitations" on page 1-23.

`gnu5.x`

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

If you select `gnu5.x`, the option `Target processor type (-target)` shows only a subset of targets that are allowed for a GCC based compiler. For other targets, use the option `Generic target options`.

For unsupported GCC extensions, see "Limitations" on page 1-23.

`gnu6.x`

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

If you select `gnu6.x`, the option `Target processor type (-target)` shows only a subset of targets that are allowed for a GCC based compiler. For other targets, use the option `Generic target options`.

For unsupported GCC extensions, see "Limitations" on page 1-23.

`gnu7.x`

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

If you select `gnu7.x`, the option `Target processor type (-target)` shows only a subset of targets that are allowed for a GCC based compiler. For other targets, use the option `Generic target options`.

For unsupported GCC extensions, see "Limitations" on page 1-23.

`gnu8.x`

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

If you select `gnu8.x`, the option `Target processor type (-target)` shows only a subset of targets that are allowed for a GCC based compiler. For other targets, use the option `Generic target options`.

For unsupported GCC extensions, see "Limitations" on page 1-23.

clang3.x

Analysis allows Clang 3.5, 3.6, 3.7, 3.8, and 3.9 syntax.

clang4.x

Analysis allows Clang 4.0.0, and 4.0.1 syntax.

clang5.x

Analysis allows Clang 5.0.0, 5.0.1, and 5.0.2 syntax.

visual9.0

Analysis allows Microsoft Visual C++ 2008 syntax.

visual10.0

Analysis allows Microsoft Visual C++ 2010 syntax.

This option implicitly enables the option -no-stl-stubs.

visual11.0

Analysis allows Microsoft Visual C++ 2012 syntax.

This option implicitly enables the option -no-stl-stubs.

visual12.0

Analysis allows Microsoft Visual C++ 2013 syntax.

This option implicitly enables the option -no-stl-stubs.

visual14.0

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

This option implicitly enables the option -no-stl-stubs.

visual15.x

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

This option implicitly enables the option -no-stl-stubs.

visual16.x

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

This option implicitly enables the option -no-stl-stubs.

keil

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

iar

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

armcc

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

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

armclang

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

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

codewarrior

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

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

cosmic

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

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

diab

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

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

greenhills

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

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

iar-ew

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

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

microchip

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

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

renesas

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

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

tasking

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

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

ti

Analysis allows non-ANSI C syntax and semantics associated with the Texas Instrumentscompiler.

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

cosmic

Analysis allows non-ANSI C syntax and semantics associated with the compiler used in the Cosmic software development tools.

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

Tips

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

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

- If you use a Visual Studio compiler, you must use a `Target processor type (-target)` option that sets `long long` to 64 bits. Compatible targets include: `i386`, `sparc`, `m68k`, `powerpc`, `tms320c3x`, `sharc21x61`, `mpc5xx`, `x86_64`, or `mcpu` with `long long` set to 64 (`-long-long-is-64bits` at the command line).

- If you use the option `Check JSF AV C++ rules (-jsf-coding-rules)`, select the compiler `generic`. If you use another compiler, Polyspace cannot check the JSF® coding rules that require conforming to the ISO standard. For example, AV Rule 8: "All code shall conform to ISO/IEC 14882:2002(E) standard C++."

Limitations

Polyspace does not support certain features of these compilers:

- GNU® compilers (version 4.7 or later):

  - Nested functions.

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

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

```
    return bar (a) + bar (b);
}
```

- Binary operations with vector types where one operand uses the shorthand notation for uniform vectors.

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

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

    res = 2 + a;  /* means {2,2,2,2} + a  */
    ```

- Forward declaration of function parameters.

    For instance, the parameter len is forward declared:

    ```
    void func (int len; char data[len][len], int len)
    {
       /* … */
    }
    ```

- Complex integer data types.

    However, complex floating point data types are supported.

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

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

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

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

- 128-bit variables.

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

- GNU compilers version 7.x:

    - Type names _FloatN and _FloatNx are not semantically supported. The analysis treats them as type float, double, or long double.

    - Constants of type _FloatN or _FloatNx with suffixes fN, FN, or fNx, such as 1.2f123 or 2.3F64x are not supported.

- Visual Studio compilers:

    - C++ Accelerated Massive Parallelism (AMP).

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

```
        void Buffer() restrict(amp)
        {
          ...
        }
```

- `__assume` statements.

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

  ```
  Asked for compulsory presence of absent entity : assert
  ```

- Managed Extensions for C++ (required for the .NET Framework), or its successor, C++/CLI (C++ modified for Common Language Infrastructure)

- `__declspec` keyword with attributes other than `noreturn`, `nothrow`, `selectany` or `thread`.

Command-Line Information

Parameter: `-compiler`
Value: `armcc | armclang | clang3.x | clang4.x | clang5.x | codewarrior | cosmic | diab | generic | gnu3.4 | gnu4.6 | gnu4.7 | gnu4.8 | gnu4.9 | gnu5.x | gnu6.x | gnu7.x | gnu8.x | greenhills | iar | iar-ew | keil | microchip | renesas | tasking | ti | visual10.0 | visual11.0 | visual12.0 | visual14.0 | visual15.x | visual16.x | visual9.0`
Default: `generic`
Example 1 (Bug Finder): `polyspace-bug-finder -lang c -sources "file1.c,file2.c" -compiler gnu4.6`
Example 2 (Bug Finder): `polyspace-bug-finder -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`
Example 1 (Code Prover): `polyspace-code-prover -lang c -sources "file1.c,file2.c" -lang c -compiler gnu4.6`
Example 2 (Code Prover): `polyspace-code-prover -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`
Example 1 (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources "file1.c,file2.c" -compiler gnu4.6`
Example 2 (Bug Finder Server): `polyspace-bug-finder-server -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`
Example 1 (Code Prover Server): `polyspace-code-prover-server -lang c -sources "file1.c,file2.c" -lang c -compiler gnu4.6`
Example 2 (Code Prover Server): `polyspace-code-prover-server -lang cpp -sources "file1.cpp,file2.cpp" -compiler visual9.0`

See Also

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

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

ARM v5 Compiler (`-compiler armcc`)

Specify ARM v5 compiler

Description

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

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

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

Settings

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

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

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

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

Tips

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

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

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

Introduced in R2019a

ARM v6 Compiler (`-compiler armclang`)

Specify ARM v6 compiler

Description

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

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

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

Settings

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

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

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

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

Tips

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

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

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

Introduced in R2019a

NXP CodeWarrior Compiler (`-compiler codewarrior`)

Specify NXP CodeWarrior compiler

Description

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

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

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

Settings

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

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

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

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

Tips

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

See Also

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

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

Introduced in R2018a

Cosmic Compiler (`-compiler cosmic`)

Specify Cosmic compiler

Description

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

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

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

Settings

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

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

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

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

Tips

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

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

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

Introduced in R2019b

Diab Compiler (`-compiler diab`)

Specify the Wind River Diab compiler

Description

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

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

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

Settings

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

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

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

Tips

If you encounter errors during Polyspace analysis, see "Errors Related to Diab Compiler".

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

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

See Also

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

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

Introduced in R2016b

Green Hills Compiler (`-compiler greenhills`)

Specify Green Hills compiler

Description

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

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

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

Settings

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

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

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

Tips

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

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

Command-Line Information
Parameter: `-compiler greenhills -target`

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

See Also

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

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

Introduced in R2017b

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

Specify IAR Embedded Workbench compiler

Description

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

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

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

Settings

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

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

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

Tips

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

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

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

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

Example (Bug Finder Server): `polyspace-bug-finder-server -compiler iar-ew -target rl78`
Example (Code Prover Server): `polyspace-code-prover-server -compiler iar-ew -target rl78`

See Also

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

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

Introduced in R2018a

MPLAB XC8 C Compiler (`-compiler microchip`)

Specify MPLAB XC8 C compiler

Description

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

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

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

Settings

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

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

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

Tip

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

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

See Also

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

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

Introduced in R2020a

Renesas Compiler (`-compiler renesas`)

Specify Renesas compiler

Description

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

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

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

Settings

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

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

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

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

Tips

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

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

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

Introduced in R2018b

TASKING Compiler (`-compiler tasking`)

Specify the Altium TASKING compiler

Description

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

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

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

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

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

Settings

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

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

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

Tips

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

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

- The CPU used is TC1793. If you use a different CPU, set the following analysis options in your project:

  - `Disabled preprocessor definitions (-U)`: Undefine the macro `__CPU_TC1793B__`.
  - `Preprocessor definitions (-D)`: Define the macro `__CPU__`. Enter `__CPU__=xxx`, where *xxx* is the name of your CPU.

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

Instead of manually specifying your compiler, if you trace your build command (makefile), Polyspace can detect your CPU and add the required definitions in your project.

- For some errors related to TASKING compiler-specific constructs, see solutions in "Errors Related to TASKING Compiler".

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

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

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

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

Introduced in R2017a

Texas Instruments Compiler (`-compiler ti`)

Specify Texas Instruments compiler

Description

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

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

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

Settings

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

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

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

Tips

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

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

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

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

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

See Also

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

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

Introduced in R2018a

Sfr type support (`-sfr-types`)

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

Description

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

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See "Dependency" on page 1-48 for other options you must also enable.

Command line and options file: Use the option `-sfr-types`. See "Command-Line Information" on page 1-48.

Why Use This Option

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

Settings

No Default

List each sfr name and its size in bits.

Dependency

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

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

See Also

Topics
"Specify Polyspace Analysis Options"

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

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

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

Description

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

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

Set Option

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

Command line and options file: Use the option `-div-round-down`. See "Command-Line Information" on page 1-51.

Why Use This Option

Use this option to emulate your compiler.

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

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

Settings

☑ On

> If either operand / or % is negative, the result of the / operator is the largest integer less than or equal to the algebraic quotient. The result of the % operator is deduced from `a % b = a - (a / b) * b`.
>
> *Example*: `assert(-5/3 == -2 && -5%3 == 1);` is true.

☐ Off (default)

> If either operand of / or % is negative, the result of the / operator is the smallest integer greater than or equal to the algebraic quotient. The result of the % operator is deduced from `a % b = a - (a / b) * b`.
>
> This behavior is also known as rounding towards zero.
>
> *Example*: `assert(-5/3 == -1 && -5%3 == -2);` is true.

Command-Line Information

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

See Also

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

Enum type definition (-enum-type-definition)

Specify how to represent an `enum` with a base type

Description

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

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

Set Option

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

Command line and options file: Use the option `-enum-type-definition`. See "Command-Line Information" on page 1-53.

Why Use This Option

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

To check your compiler settings:

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

```
enum { MAXSIGNEDBYTE=127 } mysmallenum_t;

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

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

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

```
#include <limits.h>

enum { MYINTMAX = INT_MAX } myintenum_t;

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

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

Settings

Default: `defined-by-compiler`

`defined-by-compiler`
   Uses the signed integer type for all compilers except gnu, clang and tasking.

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

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

`auto-signed-first`

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

`auto-unsigned-first`

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

- If enumerator values are positive: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`.

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

Command-Line Information

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

See Also

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

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

Disable Polyspace definitions for `char16_t` or `char32_t`

Description

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

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Target & Compiler** node. See "Dependencies" on page 1-54 for other options you must also enable.

Command line and options file: Use the option `-no-uliterals`. See "Command-Line Information" on page 1-54.

Why Use This Option

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

To check if your compiler defines these types, compile this code using the compiler settings that you typically use:

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

If the file compiles, it means that your compiler has already defined `char16_t` and `char32_t`. Enable this Polyspace option.

Settings

☑ On

> The analysis does not allow `char16_t` and `char32_t` types.

☐ Off (default)

> The analysis allows `char16_t` and `char32_t` types.

Dependencies

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

- `Source code language (-lang)` is set to `CPP` or `C-CPP`.
- `Compiler (-compiler)` is set to `generic` or a `gnu` version.

Command-Line Information

Parameter: `-no-uliterals`
Default: off
Example (Bug Finder): `polyspace-bug-finder -lang cpp -compiler gnu4.7 -cpp-version cpp11 -no-uliterals`

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

See Also
`Compiler (-compiler)`

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

Pack alignment value (-pack-alignment-value)

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

Description

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

Set Option

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

Command line and options file: Use the option `-pack-alignment-value`. See "Command-Line Information" on page 1-56.

Why Use This Option

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

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

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

Settings

Default: 8

You can enter one of these values:

- 1
- 2
- 4
- 8
- 16

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

Example (Code Prover Server): `polyspace-code-prover-server -compiler visual10 -pack-alignment-value 4`

See Also

Topics
"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"
"Assumptions About #pragma Directives" on page 4-28

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

Ignore `#pragma pack` directives

Description

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

Set Option

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

Command line and options file: Use the option `-ignore-pragma-pack`. See "Command-Line Information" on page 1-58.

Why Use This Option

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

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

Settings

☑ On

The analysis ignores the `#pragma` directives.

☐ Off (default)

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

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

See Also

Topics
"Specify Polyspace Analysis Options"
"Specify Target Environment and Compiler Behavior"
"Assumptions About #pragma Directives" on page 4-28

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

Specify the underlying data type of `size_t`

Description

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

Set Option

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

Command line and options file: Use the option `-size-t-type-is`. See "Command-Line Information" on page 1-60.

Why Use This Option

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

If you run into compilation errors during Polyspace analysis and trace the error to the definition of `size_t`, it is possible that you use a compiler option and change your compiler default. To probe further, compile this code with your compiler using the options that you typically use:

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

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

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

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

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

`unsigned-int`

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

`unsigned-long`

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

`unsigned-long-long`

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

1-59

`unsigned-char`

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

`unsigned-short`

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

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

See Also
`-custom-target`

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

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

Specify the underlying data type of `wchar_t`

Description

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

Set Option

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

Command line and options file: Use the option `-wchar-t-type-is`. See "Command-Line Information" on page 1-61.

Why Use This Option

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

Settings

Default: `defined-by-compiler`

`defined-by-compiler`

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

`signed-short`

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

`unsigned-short`

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

`signed-int`

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

`unsigned-int`

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

`signed-long`

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

`unsigned-long`

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

Command-Line Information
Parameter: `-wchar-t-type-is`

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

See Also

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

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

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

Description

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

This option does not modify compile-time expressions. For more details, see "Limitation" on page 1-63.

Set Option

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

Command line and options file: Use the option `-logical-signed-right-shift`. See "Command-Line Information" on page 1-64.

Why Use This Option

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

Settings

Default: `Arithmetical`

`Arithmetical`

   The sign bit remains:

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

`Logical`

   0 replaces the sign bit:

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

Limitation

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

For example, consider this right shift expression:

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

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

Command-Line Information

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

See Also

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

Preprocessor definitions (-D)

Replace macros in preprocessed code

Description

Replace macros with their definitions in preprocessed code.

Set Option

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

Command line and options file: Use the option -D. See "Command-Line Information" on page 1-66.

Why Use This Option

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

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

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

```
#ifdef _WIN32
  int env_var;
#endif

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

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

Settings

No Default

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

For example:

- `name1=name2` replaces all instances of `name1` by `name2`.

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

Tips

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

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

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

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

  `int __far* pValue;`

  is converted to:

  `int * pValue;`

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

  For instance, to allow use of `restrict` as identifier:

  - In the user interface, enter `restrict=my_restrict`.
  - On the command line, use the flag `-D restrict=my_restrict`.

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

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

Command-Line Information

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

See Also
`Disabled preprocessor definitions (-U)`

Topics
"Specify Polyspace Analysis Options"

Disabled preprocessor definitions (-U)

Undefine macros in preprocessed code

Description

Undefine macros in preprocessed code.

Set Option

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

Command line and options file: Use the option -U. See "Command-Line Information" on page 1-69.

Why Use This Option

Use this option to emulate your compiler behavior. For instance, if your compiler considers a macro _WIN32 as undefined when you build your code, it executes code in a #ifndef _WIN32 statement. If Polyspace considers that macro as defined, you must explicitly undefine the macro.

Some settings for Compiler (-compiler) enable certain macros by default. This option allows you undefine the macros.

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

```
#ifndef _WIN32
  int env_var;
#endif

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

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

Settings

No Default

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

Tips

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

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

- To undefine the macro, use this option.

Command-Line Information

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

See Also
`Preprocessor definitions (-D)`

Topics
"Specify Polyspace Analysis Options"

Source code encoding (`-sources-encoding`)

Specify the encoding of source files

Description

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

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

Set Option

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

Command line and options file: Use the option `-sources-encoding`. See "Command-Line Information" on page 1-71.

Why Use This Option

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

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

Settings

Default: `system`

`system`

> The analysis uses the default encoding of the operating system.

`shift-jis`

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

`iso-8859-1`

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

`windows-1252`

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

UTF-8

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

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

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

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

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

Command-Line Information

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

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

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

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

See Also

Topics
"Specify Polyspace Analysis Options"

Code from DOS or Windows file system (-dos)

Consider that file paths are in MS-DOS style

Description

Specify that DOS or Windows files are provided for analysis.

Set Option

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

Command line and options file: Use the option `-dos`. See "Command-Line Information" on page 1-72.

Why Use This Option

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

Settings

☑ On (default)

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

For example, with this option,

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

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

resolves to:

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

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

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

☐ Off

Characters are not controlled for files names or paths.

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

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

See Also

Topics
"Specify Polyspace Analysis Options"

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

Specify that a compilation error must stop the analysis

Description

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

Set Option

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

Command line and options file: Use the option `-stop-if-compile-error`. See "Command-Line Information" on page 1-75.

Why Use This Option

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

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

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

Settings

☑ On

> The analysis stops even if a single compilation error occurs.
>
> In the user interface of the Polyspace desktop products, you see the compilation errors on the **Output Summary** pane.

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

> For information on how to resolve the errors, see "Troubleshoot Compilation Errors".

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

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

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

☐ Off (default)

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

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

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

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

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

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

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

See Also
`File does not compile`

Topics
"Specify Polyspace Analysis Options"

Introduced in R2017a

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

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

Description

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

Set Option

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

Command line and options file: Use the option `-post-preprocessing-command`. See "Command-Line Information" on page 1-79.

Why Use This Option

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

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

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

  However, the option does not allow search and replacement involving regular expressions. For regular expressions, use a script.

- For mapping your library function to a standard library function, use the option `-code-behavior-specifications`.

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

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

Settings

No Default

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

Tips

- Your script must be designed to process the standard output from preprocessing and produce its results in accordance with that standard output.

- Your script must preserve the number of lines in the preprocessed file. In other words, it must not add or remove entire lines to or from the file.

  Adding a line or removing one can potentially result in some unpredictable behavior on the location of checks and macros in the Polyspace user interface.

- For a Perl script, in Windows, specify the full path to the Perl executable followed by the full path to the script.

  For example:

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

  - To specify a Perl script `replace_keyword.pl` that replaces all instances of a keyword, enter `polyspaceroot\sys\perl\win32\bin\perl.exe absolute_path \replace_keyword.pl`.

  Here, *polyspaceroot* is the location of the current Polyspace installation such as `C:\Program Files\Polyspace\R2019a\` and *absolute_path* is the location of the Perl script. If the paths contain spaces, use quotes to enclose the full path names.

- Use this Perl script as template. The script removes all instances of the `far` keyword.

  ```
  #!/usr/bin/perl

  binmode STDOUT;

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

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

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

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

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

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

For complete list of regular expressions, see Perl documentation.

- When you specify this option, the Compilation Assistant is automatically disabled.

Command-Line Information

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

Note that in Windows, you use the full path to the Perl executable.

See Also

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

Topics
"Specify Polyspace Analysis Options"
"Remove or Replace Keywords Before Compilation"

Include (`-include`)

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

Description

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

Set Option

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

Command line and options file: Use the option `-include`. See "Command-Line Information" on page 1-80.

Why Use This Option

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

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

Settings

No Default

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

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

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

See Also

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

Include folders (-I)

View include folders used for analysis

Description

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

View the include folders used for analysis.

Set Option

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

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

Settings

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

See Also
-I | Include (-include)

Ignore link errors (`-no-extern-c`)

Ignore certain linking errors

Description

Specify that the analysis must ignore certain linking errors.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Environment Settings** node. See "Dependency" on page 1-83 for other options that you must also enable.

Command line and options file: Use the option `-no-extern-C`. See "Command-Line Information" on page 1-83.

Why Use This Option

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

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

Settings

☑ On

   Ignore linking errors if possible.

☐ Off (default)

   Stop analysis for linkage errors.

Dependency

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

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

See Also

Topics
"Specify Polyspace Analysis Options"

Constraint setup (-data-range-specifications)

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

Description

This option applies primarily to a Code Prover analysis. In Bug Finder, you can only specify external constraints on global variables.

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

Set Option

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

Command line and options file: Use the option -data-range-specifications. See "Command-Line Information" on page 1-86.

Why Use This Option

Use this option for specifying constraints outside your code.

Polyspace uses the code that you provide to make assumptions about items such as variable ranges and allowed buffer size for pointers. Sometimes the assumptions are broader than what you expect because:

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

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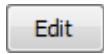
- Code Prover can consider more execution paths than those paths that occur at run time. If an operation fails along one of the execution paths, Polyspace places an orange check on the operation. If that execution path does not occur at run time, the orange check indicates a false positive.
- Bug Finder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

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

After you specify your constraints, you can save them as an XML file to use them for subsequent analyses. If your source code changes, you can update the previous constraints. You do not have to create a new constraint template.

Settings

No Default

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

For more information, see "Specify External Constraints".

Command-Line Information

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

See Also

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

Topics
"Specify Polyspace Analysis Options"
"Specify External Constraints"

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

Consider global variables as uninitialized unless explicitly initialized in code

Description

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

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

Set Option

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

Command line and options file: Use the option `-no-def-init-glob`. See "Command-Line Information" on page 1-88.

Why Use This Option

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

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

Settings

☑ On

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

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

☐ Off (default)

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

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

Tips

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

Command-Line Information

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

See Also

`Non-initialized variable`

Topics
"Specify Polyspace Analysis Options"

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

Specify functions to stub during analysis

Description

Specify functions to stub during analysis.

For specified functions, Polyspace :

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

Set Option

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

Command line and options file: Use the option `-functions-to-stub`. See "Command-Line Information" on page 1-90.

Why Use This Option

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

For instance:

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

Settings

No Default

Enter function names or choose from a list.

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

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

```
//simple function

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

```
//C++ template function

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

//C++ class method

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


//C++ template class method

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

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

Tips

- Code Prover makes assumptions about the arguments and return values of stubbed functions. For example, Polyspace assumes that the return values of stubbed functions are full range. These assumptions can affect checks in other sections of the code. See "Assumptions About Stubbed Functions" on page 4-7.

- If you stub a function, you can constrain the range of function arguments and return value. To specify constraints, use the analysis option `Constraint setup (-data-range-specifications)`.

- For C functions, these special characters are allowed:( ) < > ; _

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

  Space characters are allowed for C++, but are not allowed for C functions.

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

Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-functions-to-stub function_1,function_2`

See Also

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

Topics
"Specify Polyspace Analysis Options"

Libraries used (`-library`)

Specify libraries that you use in your program

Description

Specify libraries that you use in your program.

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

Set Option

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

Command line and options file: Use the option `-library`. See "Command-Line Information" on page 1-93.

Why Use This Option

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

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

Settings

Default: none

> The analysis uses smart stubs only for functions from the C or C++ Standard Library (if their implementations cannot be found).

autosar

> In addition to the stubbing of C or C++ Standard Library functions with missing implementations, the analysis uses smart stubs for AUTOSAR RTE API functions (even if their implementations are available).
>
> The option also triggers AUTOSAR-specific checks on function arguments. For more information, see the corresponding checkers:
>
> • Bug Finder: `Non-compliance with AUTOSAR specification`
>
>   Besides setting the option, you must also explicitly enable the above checker (or enable all checkers).

- Code Prover: `Non-compliance with AUTOSAR specification`

  Setting the option is sufficient to enable the checker.

Command-Line Information

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

See Also

Topics
"Specify Polyspace Analysis Options"

Introduced in R2021a

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

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

Description

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

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

Set Option

If you are running verification from Simulink, use the option "Stub lookup tables" on page 12-9 in Simulink Configuration Parameters, which performs the same task.

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

Command line and options file: Use the option `-stub-embedded-coder-lookup-table-functions`. See "Command-Line Information" on page 1-95.

Why Use This Option

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

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

Settings

☑ On (default)

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

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

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

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

☐ Off

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

Tips

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

Command-Line Information

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

See Also

Topics
"Specify Polyspace Analysis Options"

Introduced in R2016b

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

Specify files on which you want analysis results

Description

Specify files on which you want analysis results.

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

The option applies only to coding rule violations and code metrics. You cannot suppress Code Prover run-time checks from select source and header files.

Set Option

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

Command line and options file: Use the option `-generate-results-for`. See "Command-Line Information" on page 1-97.

Why Use This Option

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

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

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

Settings

Default: `source-headers`

`source-headers`

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

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

all-headers

> Results appear on source files and all header files. The header files can be in the same folder as source files, in subfolders of source file folders or in include folders.
>
> The source files are the files that you add to the **Source** folder of your Polyspace project (or use with the argument -sources at the command line).
>
> The include folders are the folders that you add to the **Include** folder of your Polyspace project (or use with the argument -I at the command line).

custom

> Results appear on source files and the files that you specify. If you enter a folder name, results appear on header files in that folder (and its subfolders).
>
> Click to add a field. Enter a file or folder name.

Tips

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

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

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

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

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

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

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

See Also

Topics
"Specify Polyspace Analysis Options"

Introduced in R2016a

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

Specify files on which you do not want analysis results

Description

Specify files on which you do not want analysis results.

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

The option applies only to coding rule violations, code metrics and unused global variables. You cannot suppress Code Prover run-time checks from source and header files.

Set Option

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

Command line and options file: Use the option `-do-not-generate-results-for`. See "Command-Line Information" on page 1-102.

Why Use This Option

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

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

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

Settings

Default: `include-folders`

`include-folders`

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

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

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

all-headers

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

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

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

custom

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

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

Tips

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

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

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

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

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

3 If a defect or coding rule violation involves two files and you do not generate results for one of the files, the defect or rule violation still appears. For instance, if you define two variables with similar-looking names in files `myFile.cpp` and `myFile.h`, you get a violation of the MISRA® C+

+ rule 2-10-1, even if you do not generate results for `myFile.h`. MISRA C++ rule 2-10-1 states that different identifiers must be typographically unambiguous.

The following results can involve more than one file:

MISRA C: 2004 Rules

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

MISRA C: 2012 Directives and Rules

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

MISRA C++ Rules

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

JSF C++ Rules

- JSF C++ Rule 46 — User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.

- JSF C++ Rule 48 — Identifiers will not differ by only a mixture of case, the presence/absence of the underscore character, the interchange of the letter O with the number 0 or the letter D, the interchange of the letter I with the number 1 or the letter l, the interchange of the letter S with the number 5, the interchange of the letter Z with the number 2 and the interchange of the letter n with the letter h.
- JSF C++ Rule 137 — All declarations at file scope should be static where possible.
- JSF C++ Rule 139 — External objects will not be declared in more than one file.

Polyspace Bug Finder Defects

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

4 If a global variable is never used after declaration, it appears in Code Prover results as an unused global variable. However, if it is declared in a file for which you do not want results, you do not see the unused variable in your verification results.

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

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

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

Topics
"Specify Polyspace Analysis Options"

Introduced in R2016a

No STL stubs (`-no-stl-stubs`)

Do not use Polyspace implementations of functions in the Standard Template Library

Description

Specify that the verification must not use Polyspace implementations of the Standard Template Library.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Inputs & Stubbing** node. See "Dependency" on page 1-103 for other options that you must also enable.

Command line and options file: Use the option `-no-stl-stubs`. See "Command-Line Information" on page 1-103.

Why Use This Option

When you use a class template from the Standard Template Library (STL), the Polyspace analysis, instead of verifying a full implementation of the STL template, uses stubs to improve performance and precision.

However, it might happen that your compiler redefines an STL template with an implementation that conflicts with the Polyspace implementation. If a conflict occurs, you see errors because your code uses your compiler implementation of the STL template. To avoid the errors, use this option and prevent Polyspace from using its implementations of STL templates. You must also explicitly provide the path to your compiler includes. See "C++ Standard Template Library Stubbing Errors"

Settings

☑ On

The verification does not use Polyspace implementations of the Standard Template Library.

☐ Off (default)

The verification uses efficient Polyspace implementations of the Standard Template Library.

Dependency

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

Command-Line Information
Parameter: `-no-stl-stubs`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-no-stl-stubs`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-no-stl-stubs`

See Also

Topics
"Specify Polyspace Analysis Options"

Enable automatic concurrency detection for Code Prover (`-enable-concurrency-detection`)

Automatically detect certain families of multithreading functions

Description

This option affects a Code Prover analysis only.

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

Specify whether the analysis must automatically detect POSIX®, VxWorks®, Windows, µC/OS II and other multithreading functions.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See "Dependencies" on page 1-106 for other options that you must enable or disable.

Command line and options file: Use the option `-enable-concurrency-detection`. See "Command-Line Information" on page 1-106.

Why Use This Option

If you use this option, Polyspace determines your multitasking model from your use of multithreading functions. In Bug Finder, automatic concurrency detection is enabled by default. In Code Prover, you have to explicitly enable automatic concurrency detection.

In some cases, using automatic concurrency detection can slow down the Code Prover analysis. In those cases, you can choose to not enable this option and explicitly specify your multitasking model. See "Configuring Polyspace Multitasking Analysis Manually".

Settings

☑ On

> If you use one of the supported functions for multitasking, the analysis automatically detects your multitasking model from your code.

> For a list of supported multitasking functions and limitations in auto-detection of threads, see "Auto-Detection of Thread Creation and Critical Section in Polyspace".

☐ Off (default)

> The analysis does not attempt to detect the multitasking model from your code.

> If you want to manually configure your multitasking model, see "Configuring Polyspace Multitasking Analysis Manually".

Dependencies

If you enable this option, your code must contain a `main` function. You cannot use the Code Prover options to generate a `main`.

Command-Line Information

Parameter: `-enable-concurrency-detection`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-enable-concurrency-detection`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-enable-concurrency-detection`

See Also

`Show global variable sharing and usage only (-shared-variables-mode)`

Topics
"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Auto-Detection of Thread Creation and Critical Section in Polyspace"

External multitasking configuration

Enable setup of multitasking configuration from external file definitions

Description

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

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

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

Set Option

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

Command line and options file: See "Command-Line Information" on page 1-107.

Why Use This Option

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

Settings

☑ On

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

`osek`

Look for and parse OIL files to extract multitasking description.

`autosar`

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

☐ Off (default)

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

Command-Line Information

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

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

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

Introduced in R2018a

OIL files selection (`-osek-multitasking`)

Set up multitasking configuration from OIL file definition

Description

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

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

Set Option

User interface: In the **Configuration** pane, the option is available on the **Multitasking** node. See Dependencies on page 1-113 for other options you must also enable.

Command line: and options file Use the option `-osek-multitasking`. See "Command-Line Information" on page 1-113.

Why Use This Option

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

Settings

☑ On

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

`auto`

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

`custom`

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

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

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

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

Example: Analyze Your OSEK Multitasking Project

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

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

int var1;
int var2;
int var3;

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

TASK(init) // task
{


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

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

}

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

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

void main()
{}
```

To set up your multitasking configuration and analyze the code:

1   Copy the contents of *polyspaceroot*/help/toolbox/bugfinder/examples/
    External_multitasking/OSEK or *polyspaceroot*/help/toolbox/codeprover/
    examples/External_multitasking/OSEK to your machine, for instance in
    C:\Polyspace_worskpace\OSEK.

2   Run an analysis on your OSEK project by using the command:

- Bug Finder:

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

- Code Prover:

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

- Bug Finder Server:

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

- Code Prover Server:

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

Bug Finder detects a data race on variable `var3` because of multiple read and write operation from tasks `init` and `afterinit1`. See `Data race`.

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

int var1;
int var2;
int var3;
```

There is no defect on `var2` since `afterinit1` goes to an active state (`ActivateTask()`) after `init` increments `var2`. Similarly, there is no defect on `var1` because it is protected by the `GetResource()` and `ReleaseResource()` calls.

Code Prover detects that `var3` is a potentially unprotected global variable because it is used in tasks `init` and `afterinit1` with no protection from interruption during the read and write operations. The analysis also shows that the cyclic task operation on `var4` can potentially cause an overflow. See `Potentially unprotected variable` and `Overflow`.

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

int var1;
int var2;
int var3;

...
void func()
{
  var4++;
}
```

Variable `var2` is not shared because `afterinit1` goes to an active state (`ActivateTask()`) after `init` increments `var2`. Variable `var1` is a protected variable on page 10-6 through the critical sections from the `GetResource()` and `ReleaseResource()` calls.

To see how Polyspace models the TASK, ISR, and RESOURCE definitions from your OIL files, open the **Concurrency window** from the **Dashboard** pane.

☐ Off (default)

Polyspace does not set up a multitasking configuration for your OSEK project.

Additional Considerations

- Make sure that you declare all tasks by using the DeclareTask or TASK keywords before you pass those tasks as parameters to functions or macros that expect a task. For example , if you pass task foo to ActivateTask without using DeclareTask(foo); first, Polyspace considers task foo undefined which results in a compilation error.
- The analysis ignores TerminateTask() declarations in your source code and considers that subsequent code is executed.
- Polyspace ignores syntax elements of your OIL files that do not follow the syntax defined here.

Dependencies

To enable this option in the user interface of the desktop products, first select the option External multitasking configuration.

Command-Line Information
Parameter: -osek-multitasking
Value: auto | custom='*file1 [,file2, dir1,...]*'
Default: Off
Example (Bug Finder): polyspace-bug-finder -sources *source_path* -I *include_path* -osek-multitasking custom='path\to\file1.oil, path\to\dir'
Example (Code Prover): polyspace-code-prover -sources *source_path* -I *include_path* -osek-multitasking custom='path\to\file1.oil, path\to\dir'
Example (Bug Finder Server): polyspace-bug-finder-server -sources *source_path* -I *include_path* -osek-multitasking custom='path\to\file1.oil, path\to\dir'
Example (Code Prover Server): polyspace-code-prover-server -sources *source_path* -I *include_path* -osek-multitasking custom='path\to\file1.oil, path\to\dir'

See Also
Show global variable sharing and usage only (-shared-variables-mode)

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

Introduced in R2017b

ARXML files selection (-autosar-multitasking)

Set up multitasking configuration from ARXML file definitions

Description

To detect data races in large AUTOSAR applications, use this option with Polyspace Bug Finder™.

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

Specify the ARXML files that Polyspace parses to set up the multitasking configuration of your AUTOSAR project.

Set Option

User interface: In the **Configuration** pane, the option is available on the **Multitasking** node. See Dependencies on page 1-115 for other options you must also enable.

Command line: and options file Use the option `-autosar-multitasking`. See "Command-Line Information" on page 1-113.

Why Use This Option

If your project includes ARXML files with `<ECUC-CONTAINER-VALUE>` elements, Polyspace can parse these files to set up tasks, interrupts, cyclical tasks, and critical sections. You do not have to set them up manually.

Settings

☑ On

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

When you select this option, the software assumes that you use the OSEK multitasking API in your source code to declare and define tasks and interrupts. Polyspace supports these OSEK multitasking keywords:

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

Polyspace parses the ARXML files that you provide for `OsTask`, `OsIsr`, `OsResource`, `OsAlarm`, and `OsEvent` definitions. The analysis uses these definitions and the supported multitasking keywords to configure tasks, interrupts, cyclical tasks, and critical sections.

To see how Polyspace models the `OsTask`, `OsIsr`, and `OsResource` definitions from your ARXML files, open the **Concurrency window** from the **Dashboard** pane. In that window, under the **Entry points** column, the names of the elements are extracted from their `<SHORT-NAME>` values in the ARXML files.

☐ Off (default)

Polyspace does not set up a multitasking configuration for your AUTOSAR project.

Additional Considerations

- The analysis ignores `TerminateTask()` declarations in your source code and considers that subsequent code is executed.
- Polyspace supports multitasking configuration only from ARXML files for AUTOSAR specification version 4.0 and later.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `External multitasking configuration`.

Command-Line Information

Parameter: `-autosar-multitasking`
Value: *file1 [,file2, dir1,...]*
Default: Off
Example (Bug Finder): polyspace-bug-finder -sources *source_path* -I *include_path* -autosar-multitasking C:\Polyspace_Workspace\AUTOSAR\myFile.arxml
Example (Bug Finder Server): polyspace-bug-finder-server -sources *source_path* -I *include_path* -autosar-multitasking C:\Polyspace_Workspace\AUTOSAR \myFile.arxml

See Also

Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)|External multitasking configuration|OIL files selection (-osek-multitasking)|Show global variable sharing and usage only (-shared-variables-mode)

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

Introduced in R2018a

Configure multitasking manually

Consider that code is intended for multitasking

Description

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

Specify whether your code is a multitasking application. This option allows you to manually configure the multitasking structure for Polyspace.

Set Option

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

Command line and options file: See "Command-Line Information" on page 1-117.

Why Use This Option

By default, Bug Finder determines your multitasking model from your use of multithreading functions. In Code Prover, you have to enable automatic concurrency detection with the option Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection). However, in some cases, using automatic concurrency detection can slow down the Code Prover analysis.

In cases where automatic concurrency detection is not supported, you can explicitly specify your multitasking model by using this option. Once you select this option, you can explicitly specify your entry point functions, cyclic tasks, interrupts and protection mechanisms for shared variables, such as critical section details.

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.

  See "Global Variables".
- Whether a run-time error can occur.

  For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For more information, see "Concurrency Defects".

Settings

☑ On

  The code is intended for a multitasking application.

  You have to explicitly specify your multitasking configuration using other Polyspace options. See "Configuring Polyspace Multitasking Analysis Manually".

☐ Off (default)

> The code is not intended for a multitasking application.
>
> Disabling the option has this additional effect in Code Prover:

- If a `main` exists, Code Prover verifies only those functions that are called by the `main`.
- If a `main` does not exist, Polyspace verifies the functions that you specify. To verify the functions, Polyspace generates a `main` function and calls functions from the generated `main` in a sequence that you specify. For more information, see `Verify module or library (-main-generator)`.

Tips

If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

Command-Line Information

There is no single command-line option to turn on multitasking analysis. By using any of the options `Tasks (-entry-points)`, `Cyclic tasks (-cyclic-tasks)` or `Interrupts (-interrupts)`, you turn on multitasking analysis.

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Critical section details (-critical-section-begin -critical-section-end)` | `Cyclic tasks (-cyclic-tasks)` | `Tasks (-entry-points)` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics
"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"

Tasks (`-entry-points`)

Specify functions that serve as tasks to your multitasking application

Description

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

Specify functions that serve as tasks to your code. If the function does not exist, the verification warns you and continues the verification.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See "Dependencies" on page 1-119 for other options you must also enable.

Command line and options file: Use the option `-entry-points`. See "Command-Line Information" on page 1-119.

Why Use This Option

Use this option when your code is intended for multitasking.

To specify cyclic tasks and interrupts, use the options `Cyclic tasks (-cyclic-tasks)` and `Interrupts (-interrupts)`. Use this option to specify other tasks.

A Code Prover analysis uses your specifications to determine:

- Whether a global variable is shared.

  See "Global Variables".
- Whether a run-time error can occur.

  For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For more information, see "Concurrency Defects".

Settings

No Default

Enter function names or choose from a list.

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

Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

Tips

- In Code Prover, the functions representing entry points must have the form

  `void functionName (void)`

- If a function `func` takes arguments or returns a value, you cannot use it directly as an entry point. To use `func` as an entry point:, call `func` from a wrapper `void-void` function and specify the wrapper as an entry point. See "Configuring Polyspace Multitasking Analysis Manually".

- If you specify a function as a task, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:

  `task func_name must be a userdef function without parameters`

  A Bug Finder analysis continues but does not consider the function as an entry point.

- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

- The Polyspace multitasking analysis assumes that a task cannot interrupt itself.

Command-Line Information
Parameter: `-entry-points`
No Default
Value: *function1*`[,`*function2*`[,...]]`
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-entry-points func_1,func_2`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-entry-points func_1,func_2`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-entry-points func_1,func_2`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-entry-points func_1,func_2`

See Also
`-non-preemptable-tasks` | `-preemptable-interrupts` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Show global variable sharing and usage only (-shared-variables-mode)`

Topics
"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"

Cyclic tasks (`-cyclic-tasks`)

Specify functions that represent cyclic tasks

Description

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

Specify functions that represent cyclic tasks. The analysis assumes that operations in the function body:

- Can execute any number of times.
- Can be interrupted by noncyclic tasks, other cyclic tasks and interrupts. Noncyclic tasks are specified with the option `Tasks (-entry-points)` and interrupts are specified with the option `Interrupts (-interrupts)`.

  To model a cyclic task that cannot be interrupted by other cyclic tasks, specify the task as nonpreemptable. See `-non-preemptable-tasks`. For examples, see "Define Preemptable Interrupts and Nonpreemptable Tasks".

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See "Dependencies" on page 1-121 for other options you must also enable.

Command line and options file: Use the option `-cyclic-tasks`. See "Command-Line Information" on page 1-121.

Why Use This Option

Use this option to specify cyclic tasks in your multitasking code. The functions that you specify must have the prototype:

```
void function_name(void);
```

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.

  See "Global Variables".
- Whether a run-time error can occur.

  For instance, if the operation `var++` occurs in the body of a cyclic task and you do not impose a limit on `var`, the operation can overflow. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For the `Data race` defect, the software establishes the following relations between preemptable tasks and other tasks.

- *Data race between two preemptable tasks*:

  Unless protected, two operations in different preemptable tasks can interfere with each other. If the operations use the same shared variable without protection, a data race can occur.

If both operations are atomic, to see the defect, you have to enable the checker **Data race including atomic operations**.

- *Data race between a preemptable task and a nonpreemptable task or interrupt*:

  - An atomic operation in a preemptable task cannot interfere with an operation in a nonpreemptable task or an interrupt. Even if the operations use the same shared variable without protection, a data race cannot occur.

  - A nonatomic operation in a preemptable task also cannot interfere with an operation in a nonpreemptable task or an interrupt. However, the latter operation can interrupt the former. Therefore, if the operations use the same shared variable without protection, a data race can occur.

For more information, see "Concurrency Defects".

Settings

No Default

Enter function names or choose from a list.

- Click ⊞ to add a field and enter the function name.

- Click ⊡ to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

Tips

- In Code Prover, the functions representing cyclic tasks must have the form

  `void functionName (void)`

- If a function `func` takes arguments or returns a value, you cannot use it directly as a cyclic task. To use `func` as a cyclic task:, call `func` from a wrapper `void-void` function and specify the wrapper as a cyclic task. See "Configuring Polyspace Multitasking Analysis Manually".

- If you specify a function as a cyclic task, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:

  `task func_name must be a userdef function without parameters`

  A Bug Finder analysis continues but does not consider the function as a cyclic task.

- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

- The Polyspace multitasking analysis assumes that a task cannot interrupt itself.

Command-Line Information
Parameter: `-cyclic-tasks`

No Default
Value: *function1*[,*function2*[,...]]
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-cyclic-tasks func_1,func_2`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-cyclic-tasks func_1,func_2`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-cyclic-tasks func_1,func_2`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-cyclic-tasks func_1,func_2`

See Also

`-non-preemptable-tasks`|`-preemptable-interrupts`|`Interrupts (-interrupts)`|`Show global variable sharing and usage only (-shared-variables-mode)`|`Tasks (-entry-points)`

Topics
"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"
"Define Preemptable Interrupts and Nonpreemptable Tasks"

Introduced in R2016b

Interrupts (`-interrupts`)

Specify functions that represent nonpreemptable interrupts

Description

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

Specify functions that represent nonpreemptable interrupts. The analysis assumes that operations in the function body:

- Can execute any number of times.
- Cannot be interrupted by noncyclic tasks, cyclic tasks or other interrupts. Noncyclic tasks are specified with the option `Tasks (-entry-points)` and cyclic tasks are specified with the option `Cyclic tasks (-cyclic-tasks)`.

  To model an interrupt that can be interrupted by other interrupts, specify the interrupt as preemptable. See `-preemptable-interrupts`. For examples, see "Define Preemptable Interrupts and Nonpreemptable Tasks".

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See "Dependencies" on page 1-124 for other options you must also enable.

Command line and options file: Use the option `-interrupts`. See "Command-Line Information" on page 1-124.

Why Use This Option

Use this option to specify interrupts in your multitasking code. The functions that you specify must have the prototype:

void *function_name*(void);

A Code Prover verification uses your specifications to determine:

- Whether a global variable is shared.

  See "Global Variables".
- Whether a run-time error can occur.

  For instance, if the operation `var=INT_MAX;` occurs in an interrupt and `var++` occurs in the body of a task, an overflow can occur if the interrupt excepts before the operation in the task. The analysis detects the possible overflow.

A Bug Finder analysis uses your specifications to look for concurrency defects. For the `Data race` defect, the analysis establishes the following relations between interrupts and other tasks:

- *Data race between two interrupts*:

Two operations in different interrupts cannot interfere with each other (unless one of the interrupts is preemptable). Even if the operations use the same shared variable without protection, a data race cannot occur.

- *Data race between an interrupt and another task*:

  - An operation in an interrupt cannot interfere with an atomic operation in any other task. Even if the operations use the same shared variable without protection, a data race cannot occur.

  - An operation in an interrupt can interfere with a nonatomic operation in any other task unless the other task is also a nonpreemptable interrupt. Therefore, if the operations use the same shared variable without protection, a data race can occur.

See "Concurrency Defects".

Settings

No Default

Enter function names or choose from a list.

- Click ![plus icon] to add a field and enter the function name.

- Click ![list icon] to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

Tips

- In Code Prover, the functions representing interrupts must have the form

  ```
  void functionName (void)
  ```

- If a function `func` takes arguments or returns a value, you cannot use it directly as an interrupt. To use `func` as an interrupt, call `func` from a wrapper `void-void` function and specify the wrapper as an interrupt. See "Configuring Polyspace Multitasking Analysis Manually".

- If you specify a function as an interrupt, you must provide its definition. Otherwise, a Code Prover verification stops with the error message:

  ```
  task func_name must be a userdef function without parameters
  ```

  A Bug Finder analysis continues but does not consider the function as an interrupt.

- If you run a file by file verification in Code Prover, your multitasking options are ignored. See `Verify files independently (-unit-by-unit)`.

- The Polyspace multitasking analysis assumes that an interrupt cannot interrupt itself.

Command-Line Information
Parameter: `-interrupts`
No Default

Value: *function1*[,*function2*[,...]]
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-interrupts func_1,func_2`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-interrupts func_1,func_2`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-interrupts func_1,func_2`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-interrupts func_1,func_2`

See Also

`-non-preemptable-tasks`|`-preemptable-interrupts`|`Cyclic tasks (-cyclic-tasks)`|`Show global variable sharing and usage only (-shared-variables-mode)`|`Tasks (-entry-points)`

Topics
"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"
"Define Preemptable Interrupts and Nonpreemptable Tasks"

Introduced in R2016b

Critical section details (`-critical-section-begin -critical-section-end`)

Specify functions that begin and end critical sections

Description

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

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock function and an unlock function.

```
lock();
/* Critical section code */
unlock();
```

Specify the lock and unlock function names for your critical sections (for instance, `lock()` and `unlock()` in above example).

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See "Dependencies" on page 1-127 for other options you must also enable.

Command line and options file: Use the option `-critical-section-begin` and `-critical-section-end`. See "Command-Line Information" on page 1-128.

Why Use This Option

When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait till `my_task` calls the corresponding unlock function. Therefore, critical section operations in the other tasks cannot interrupt critical section operations in `my_task`.

For instance, the operation `var++` in `my_task1` and `my_task2` cannot interrupt each other.

```
int var;

void my_task1() {
    my_lock();
    var++;
    my_unlock();
}

void my_task2() {
    my_lock();
    var++;
    my_unlock();
}
```

Using your specifications, a Code Prover verification checks if your placement of lock and unlock functions protects all shared variables from concurrent access. When determining values of those variables, the verification accounts for the fact that critical sections in different tasks do not interrupt each other.

A Bug Finder analysis uses the critical section information to look for concurrency defects such as data race and deadlock.

Settings

No Default

Click ![icon] to add a field.

- In **Starting routine**, enter name of lock function.
- In **Ending routine**, enter name of unlock function.

Enter function names or choose from a list.

- Click ![icon] to add a field and enter the function name.
- Click ![icon] to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

Tips

- You can also use primitives such as the POSIX functions `pthread_mutex_lock` and `pthread_mutex_unlock` to begin and end critical sections. For a list of primitives that Polyspace can detect automatically, see "Auto-Detection of Thread Creation and Critical Section in Polyspace".
- For function calls that begin and end critical sections, Polyspace ignores the function arguments.

  For instance, Polyspace treats the two code sections below as the same critical section.

Starting routine: my_lock
Ending routine: my_unlock

```void my_task1() {```
`   my_lock(1);`
`   /* Critical section code */`
`   my_unlock(1);`
`}`
	```void my_task2() {```
	`   my_lock(2);`
	`   /* Critical section code */`
	`   my_unlock(2);`
	`}`

  To work around the limitation, see "Define Critical Sections with Functions That Take Arguments".

- The functions that begin and end critical sections must be functions. For instance, if you define a function-like macro:

  ```
  #define init() num_locks++
  ```

  You cannot use the macro `init()` to begin or end a critical section.

- When you use multiple critical sections, you can run into issues such as:

- Deadlock: A sequence of calls to lock functions causes two tasks to block each other.
- Double lock: A lock function is called twice in a task without an intermediate call to an unlock function.

Use Polyspace Bug Finder to detect such issues. See "Concurrency Defects".

Then, use Polyspace Code Prover™ to detect if your placement of lock and unlock functions actually protects all shared variables from concurrent access. See "Global Variables".

- When considering possible values of shared variables, a Code Prover verification takes into account your specifications for critical sections.

However, if the shared variable is a pointer or array, the software uses the specifications only to determine if the variable is a shared protected global variable. For run-time error checking, the software does not take your specifications into account and considers that the variable can be concurrently accessed.

Command-Line Information
Parameter: `-critical-section-begin|-critical-section-end`
No Default
Value: `function1:cs1[,function2:cs2[,...]]`
Example (Bug Finder): `polyspace-bug_finder -sources file_name -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`
Example (Code Prover): `polyspace-code-prover -sources file_name -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`
Example (Bug Finder Server): `polyspace-bug_finder-server -sources file_name -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`

See Also
`-non-preemptable-tasks|-preemptable-interrupts|Cyclic tasks (-cyclic-tasks)|Interrupts (-interrupts)|Tasks (-entry-points)|Temporally exclusive tasks (-temporal-exclusions-file)`

Topics
"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"
"Define Atomic Operations in Multitasking Code"
"Define Critical Sections with Functions That Take Arguments"
"Concurrency Defects"
"Global Variables"

Temporally exclusive tasks (`-temporal-exclusions-file`)

Specify entry point functions that cannot execute concurrently

Description

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

Specify entry point functions that cannot execute concurrently. The execution of the functions cannot overlap with each other.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Multitasking** node. See "Dependencies" on page 1-129 for other options you must also enable.

Command line and options file: Use the option `-temporal-exclusions-file`. See "Command-Line Information" on page 1-130.

Why Use This Option

Use this option to implement temporal exclusion in multitasking code.

A Code Prover verification checks if specifying certain tasks as temporally exclusive protects all shared variables from concurrent access. When determining possible values of those shared variables, the verification accounts for the fact that temporally exclusive tasks do not interrupt each other. See "Global Variables".

A Bug Finder analysis uses the temporal exclusion information to look for concurrency defects such as data race. See "Concurrency Defects".

Settings

No Default

Click to add a field. In each field, enter a space-separated list of functions. Polyspace considers that the functions in the list cannot execute concurrently.

Enter the function names manually or choose from a list.

- Click to add a field and enter the function names.

- Click to list functions in your code. Choose functions from the list.

Dependencies

To enable this option in the user interface of the desktop products, first select the option `Configure multitasking manually`.

Tips

When considering possible values of shared variables, a Code Prover verification takes into account your specifications for temporally exclusive tasks.

However, if the shared variable is a pointer or array, the software uses the specifications only to determine if the variable is a shared protected global variable. For run-time error checking in Code Prover, the software does not take your specifications into account and considers that the variable can be concurrently accessed.

Command-Line Information

For the command-line option, create a temporal exclusions file in the following format:

- On each line, enter one group of temporally excluded tasks.
- Within a line, the tasks are separated by spaces.

To enter comments, begin with #. For an example, see the file *polyspaceroot*\polyspace
\examples\cxx\Code_Prover_Example\sources\temporal_exclusions.txt. Here,
polyspaceroot is the Polyspace installation folder, for example `C:\Program Files\Polyspace`
`\R2019a`.
Parameter: `-temporal-exclusions-file`
No Default
Value: Name of temporal exclusions file
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-temporal-`
`exclusions-file "C:\exclusions_file.txt"`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-temporal-`
`exclusions-file "C:\exclusions_file.txt"`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-`
`temporal-exclusions-file "C:\exclusions_file.txt"`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-`
`temporal-exclusions-file "C:\exclusions_file.txt"`

See Also

`-non-preemptable-tasks|-preemptable-interrupts|Critical section details (-`
`critical-section-begin -critical-section-end)|Cyclic tasks (-cyclic-tasks)|`
`Interrupts (-interrupts)|Tasks (-entry-points)`

Topics
"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"
"Define Atomic Operations in Multitasking Code"
"Concurrency Defects"
"Global Variables"

Set checkers by file (-checkers-selection-file)

Define a custom set of coding standards checks for your analysis

Description

Specify the full path of a configuration XML file where you define custom selections of coding standards checkers. In the same XML file, you can specify a custom selection of checkers for each of these coding standards:

- MISRA C:2004
- MISRA C:2012
- MISRA C++
- JSF AV C++
- AUTOSAR C++14 *(Bug Finder only)*
- CERT® C *(Bug Finder only)*
- CERT C++ *(Bug Finder only)*
- ISO/IEC TS 17961 *(Bug Finder only)*
- Polyspace Guidelines *(Bug Finder only)*

You can also define custom rules to match identifiers in your code to text patterns you specify.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

Command line and options file: Use the option `-checkers-selection-file`. See "Command-Line Information" on page 1-134.

When you enable this option, set the coding standards that you select to `from-file` to use the specified configuration file.

Why Use This Option

Use this option to define a selection of coding standard checkers specific to your organization. The configuration of different coding standards is consolidated in a single XML file that you can reuse across projects to enforce common coding standards.

Settings

☑ On

Polyspace checks your code against the selection of coding standard checkers, or the custom rules, defined in the configuration file that you specify.

To create a configuration file by using the Polyspace Desktop, in the **Configuration**, select **Coding Standards & Code Metrics**. To open the **Checkers selection** interface, click the folder

( ) on the right pane. Choose the coding standards that you want to configure in the left pane, and then select the rules that you want to activate in the right pane.

To create a configuration file by using Polyspace As you Code IDE plugins, refer to the documentation of your specific plugin.

To use or update an existing file, enter the full path to the file in the in the **Select file** field of the **Checkers selection** dialog box. Alternatively, click **Browse** in the **Findings selection** window and browse to the existing file.

V Checkers selection

MISRA C:2004

Select file

MISRA C:2004 (132/132)	Select rules in category: ☑ All ☑ required ☑ a
1 Environment	
2 Language extensions	Status Category Name
3 Documentation	⊞ ☑ 1 Environment
4 Character sets	⊞ ☑ 2 Language extensions
5 Identifiers	⊞ ☑ 3 Documentation
6 Types	⊞ ☑ 4 Character sets
7 Constants	⊞ ☑ 5 Identifiers
8 Declarations and definitions	⊞ ☑ 6 Types
9 Initialization	⊞ ☑ 7 Constants
10 Arithmetic type conversions	⊞ ☑ 8 Declarations and definitions
11 Pointer type conversions	⊞ ☑ 9 Initialization
12 Expressions	⊞ ☑ 10 Arithmetic type conversions
13 Control statement expressions	⊞ ☑ 11 Pointer type conversions
14 Control flow	⊞ ☑ 12 Expressions
15 Switch statements	⊞ ☑ 13 Control statement expressions
16 Functions	⊞ ☑ 14 Control flow
17 Pointers and arrays	⊞ ☑ 15 Switch statements
18 Structures and unions	⊞ ☑ 16 Functions
19 Preprocessing directives	⊞ ☑ 17 Pointers and arrays
20 Standard libraries	⊞ ☑ 18 Structures and unions
21 Run-time failures	⊞ ☑ 19 Preprocessing directives
⊞ MISRA AC AGC (130/130)	⊞ ☑ 20 Standard libraries
⊞ MISRA C:2012 (173/173)	⊞ ☑ 21 Run-time failures
⊞ MISRA C++:2008 (211/211)	
⊞ JSF AV C++ (160/160)	
⊞ SEI CERT C (204/204)	
⊞ SEI CERT C++ (134/134)	
ISO/IEC TS 17961 (46/46)	
⊞ AUTOSAR C++14 (327/327)	
⊞ Polyspace Guidelines (18/18)	
⊞ Custom (43/43)	

☐ Off (default)

Polyspace does not check your code against the selection of coding standard checkers, or the custom rules, defined in the configuration file you specify.

Tips

- For the Polyspace desktop products, specify the coding standard configuration in the Polyspace User Interface. When you save the configuration, an XML file is created for use in the current and other projects.

- For the Polyspace Server products, you have to create a coding standard XML. Depending on the standard that you want to enable, make a writeable copy of one of the files in *polyspaceserverroot*\help\toolbox\polyspace_bug_finder_server\examples \coding_standards_XML. Turn off rules by using entries in the XML file (all rules from a standard are enabled in the template). Here, *polyspaceserverroot* is the root installation folder for the Polyspace Server products, for instance, C:\Program Files\Polyspace Server \R2019a.

  For instance, to turn off MISRA C:2012 rule 8.1, in the file misra_c_2012_rules.xml, use this entry:

  ```
  <standard name="MISRA C:2012">
    ...
    <section name="8 Declarations and definitions">
        ...
        <check id="8.1" state="off">
        </check>
        ...
    </section>
    ...
  </standard>
  ```

  For a full list of rule IDs and section names, see:

  -
  -
  -
  -
  - "Custom Coding Rules"
  - "JSF C++ Rules"
  - "MISRA C:2004 Rules"
  - "MISRA C:2012 Directives and Rules"
  - "MISRA C++:2008 Rules"
  -

  Note The XML format of the checker configuration file might change in future releases.

Command-Line Information

Parameter: -checkers-selection-file
Value: Full path of XML configuration file
Default: Off
Example (Bug Finder): polyspace-bug-finder -sources *file_name* -checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file

Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-checkers-selection-file "C:\Standards\custom_config.xml" -misra3 from-file`

See Also

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

Topics
"Specify Polyspace Analysis Options"
"Check for Coding Standard Violations"

Check MISRA C:2004 (`-misra2`)

Check for violation of MISRA C:2004 rules

Description

Specify whether to check for violation of MISRA C:2004 rules. Each value of the option corresponds to a subset of rules to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See "Dependencies" on page 1-137 for other options that you must also enable.

Command line and options file: Use the option `-misra2`. See "Command-Line Information" on page 1-137.

Why Use This Option

Use this option to specify the subset of MISRA C:2004 rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

Settings

Default: `required-rules`

`required-rules`

> Check required coding rules.

`single-unit-rules`

> Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

`system-decidable-rules`

> Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

`all-rules`

> Check required and advisory coding rules.

`SQO-subset1`

> Check only a subset of MISRA C rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see "Software Quality Objective Subsets (C:2004)".

SQO-subset2

Check a subset of rules including `SQO-subset1` and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see "Software Quality Objective Subsets (C:2004)".

from-file

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click Edit , then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Dependencies

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

  For projects with mixed C and C++ code, the MISRA C:2004 checker analyzes only `.c` files.

- If you set `Source code language (-lang)` to `C-CPP`, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:

  1 Find coding rule violations in `SQO-subset1`. Fix your code to address the violations and rerun verification.

  2 Find coding rule violations in `SQO-subset2`. Fix your code to address the violations and rerun verification.

- If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see "Coding Rule Subsets Checked Early in Analysis".

Command-Line Information

Parameter: `-misra2`
Value: `required-rules` | `all-rules` | `SQO-subset1` | `SQO-subset2` | `single-unit-rules` | `system-decidable-rules` | `from-file`
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-misra2 all-rules`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-misra2 all-rules`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-misra2 all-rules`

Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-misra2 all-rules`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file (only XML format supported)
Not recommended starting in R2019a

Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

<u>**Desktop interface:**</u>

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click ⬜. In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename*`.xml`, where *filename* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

<u>**Command-line:**</u>

If you do not have access to a Polyspace desktop interface, use the file `misra_c_2004_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *polyspaceroot*`\help\toolbox` `\polyspace_bug_finder_server\examples\coding_standards_XML`. Here, *polyspaceroot* is the root installation folder for the Polyspace products, for instance, `C:\Program Files` `\Polyspace\R2019a`. To update your script, see this table

Option	Use Instead
`-misra2 "custom_standard.conf"`	`-checkers-selection-file` `misra_c_2004_rules.xml -misra2 from-file`

.

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C:2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

•

•

•

- "Custom Coding Rules"
- "JSF C++ Rules"
- "MISRA C:2004 Rules"
- "MISRA C:2012 Directives and Rules"
- "MISRA C++:2008 Rules"

See Also

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

Topics
"Specify Polyspace Analysis Options"
"Check for Coding Standard Violations"
"MISRA C:2004 Rules"

Check MISRA AC AGC (`-misra-ac-agc`)

Check for violation of MISRA AC AGC rules

Description

Specify whether to check for violation of rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation.* Each value of the option corresponds to a subset of rules to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See "Dependencies" on page 1-141 for other options that you must also enable.

Command line and options file: Use the option `-misra-ac-agc`. See "Command-Line Information" on page 1-141.

Why Use This Option

Use this option to specify the subset of MISRA C:2004 AC AGC rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

Settings

Default: `OBL-rules`

`OBL-rules`

   Check required coding rules.

`OBL-REC-rules`

   Check required and recommended rules.

`single-unit-rules`

   Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

`system-decidable-rules`

   Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

`all-rules`

   Check required, recommended and readability-related rules.

`SQO-subset1`

   Check a subset of rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see "Software Quality Objective Subsets (AC AGC)".

SQO-subset2

Check a subset of rules including SQO-subset1 and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see "Software Quality Objective Subsets (AC AGC)".

from-file

Specify an XML file where you configure a custom selection of checkers for this coding standard.

To create a configuration file, click [Edit], then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to from-file, enable Set checkers by file (-checkers-selection-file).

Dependencies

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

  For projects with mixed C and C++ code, the MISRA AC AGC checker analyzes only .c files.

- If you set Source code language (-lang) to C-CPP, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:

  1 Find coding rule violations in SQO-subset1. Fix your code to address the violations and rerun verification.

  2 Find coding rule violations in SQO-subset2. Fix your code to address the violations and rerun verification.

- If you select the option single-unit-rules or system-decidable-rules and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see "Coding Rule Subsets Checked Early in Analysis".

Command-Line Information

Parameter: -misra-ac-agc
Value: OBL-rules | OBL-REC-rules | single-unit-rules | system-decidable-rules | all-rules | SQO-subset1 | SQO-subset2 | from-file
Example (Bug Finder): polyspace-bug-finder -sources *file_name* -misra-ac-agc all-rules
Example (Code Prover): polyspace-code-prover -sources *file_name* -misra-ac-agc all-rules
Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -misra-ac-agc all-rules

Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -misra-ac-agc all-rules`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file (only XML format supported)
Not recommended starting in R2019a

Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

<u>Desktop interface:</u>

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename*.xml, where *filename* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

<u>Command-line:</u>

If you do not have access to a Polyspace desktop interface, use the file `misra_ac_agc_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *polyspaceroot*\help\toolbox \polyspace_bug_finder_server\examples\coding_standards_XML. Here, *polyspaceroot* is the root installation folder for the Polyspace products, for instance, `C:\Program Files \Polyspace\R2019a`. To update your script, see this table

Option	Use Instead
`-misra-ac-agc "custom_standard.conf"`	`-checkers-selection-file misra_ac_agc_rules.xml -misra-ac-agc from-file`

.

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C:2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

•

•

•

• "Custom Coding Rules"

• "JSF C++ Rules"

• "MISRA C:2004 Rules"

• "MISRA C:2012 Directives and Rules"

• "MISRA C++:2008 Rules"

See Also
Do not generate results for (`-do-not-generate-results-for`)

Topics
"Specify Polyspace Analysis Options"
"Check for Coding Standard Violations"
"MISRA C:2004 Rules"

Check MISRA C:2012 (`-misra3`)

Check for violations of MISRA C:2012 rules and directives

Description

Specify whether to check for violations of MISRA C:2012 guidelines. Each value of the option corresponds to a subset of guidelines to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See "Dependencies" on page 1-145 for other options that you must also enable.

Command line and options file: Use the option `-misra3`. See "Command-Line Information" on page 1-146.

Why Use This Option

Use this option to specify the subset of MISRA C:2012 rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

Settings

Default: `mandatory-required`

`mandatory`

Check for mandatory guidelines.

`mandatory-required`

Check for mandatory and required guidelines.

- Mandatory guidelines: Your code must comply with these guidelines.

- Required guidelines: You may deviate from these guidelines. However, you must complete a formal deviation record, and your deviation must be authorized.

    See Section 5.4 of the MISRA C:2012 guidelines. For an example of a deviation record, see Appendix I of the MISRA C:2012 guidelines.

Note To turn off some required guidelines, instead of `mandatory-required` select `custom`. To clear specific guidelines, click Edit . In the **Comment** column, enter your rationale for disabling a guideline. For instance, you can enter the Deviation ID that refers to a deviation record for the guideline. The rationale appears in your generated report.

---

single-unit-rules

> Check a subset of rules that apply only to single translation units. These rules are checked in the compilation phase of the analysis.

system-decidable-rules

> Check rules in the single-unit-rules subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. These rules are checked in the compilation and linking phases of the analysis.

all

> Check for mandatory, required, and advisory guidelines.

SQO-subset1
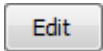
> Check for only a subset of guidelines. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see "Software Quality Objective Subsets (C:2012)".

SQO-subset2

> Check for the subset SQO-subset1, plus some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see "Software Quality Objective Subsets (C:2012)".

from-file

> Specify an XML file where you configure a custom selection of checkers for this coding standard.
>
> To create a configuration file, click **Edit**, then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.
>
> To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.
>
> If you set the option to from-file, enable Set checkers by file (-checkers-selection-file).

Dependencies

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

  For projects with mixed C and C++ code, the MISRA C:2012 checker analyzes only .c files.

- If you set Source code language (-lang) to C-CPP, you can activate a C coding rule checker **and** a C++ coding rule checker. When you have both C and C++ coding rule checkers active, to avoid duplicate results, Polyspace does not produce the C coding rules found in the linking phase (such as MISRA C:2012 Rule 8.3).

Tips

- To reduce unproven results in Polyspace Code Prover:

  1 Find coding rule violations in SQO-subset1. Fix your code to address the violations and rerun verification.

  2 Find coding rule violations in SQO-subset2. Fix your code to address the violations and rerun verification.

- If you select the option `single-unit-rules` or `system-decidable-rules` and choose to detect coding rule violations only, the analysis can complete quicker than checking other rules. For more information, see "Coding Rule Subsets Checked Early in Analysis".

- Polyspace Code Prover does not support checking of the following:

  - MISRA C:2012 Directive 4.13 and 4.14
  - MISRA C:2012 Rule 21.13, 21.14, and 21.17 - 21.20
  - MISRA C:2012 Rule 22.1 - 22.4 and 22.6 - 22.10

  For support of all MISRA C:2012 rules including the security guidelines in Amendment 1, use Polyspace Bug Finder.

- In code generated by using Embedded Coder®, there are known deviations from MISRA C:2012. See "Deviations Rationale for MISRA C:2012 Compliance" (Embedded Coder).

Command-Line Information

Parameter: `-misra3`
Value: `mandatory` | `mandatory-required` | `single-unit-rules` | `system-decidable-rules` | `all` | `SQO-subset1` | `SQO-subset2` | `from-file`
Example (Bug Finder): `polyspace-bug-finder -lang c -sources` *`file_name`* `-misra3 mandatory-required`
Example (Code Prover): `polyspace-code-prover -lang c -sources` *`file_name`* `-misra3 mandatory-required`
Example (Bug Finder Server): `polyspace-bug-finder-server -lang c -sources` *`file_name`* `-misra3 mandatory-required`
Example (Code Prover Server): `polyspace-code-prover-server -lang c -sources` *`file_name`* `-misra3 mandatory-required`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file (only XML format supported)
Not recommended starting in R2019a

Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as

filename.xml, where *filename* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `misra_c_2012_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *polyspaceroot*`\help\toolbox`
`\polyspace_bug_finder_server\examples\coding_standards_XML`. Here, *polyspaceroot* is the root installation folder for the Polyspace products, for instance, `C:\Program Files`
`\Polyspace\R2019a`. To update your script, see this table

Option	Use Instead

`-misra3 "custom_standard.conf"`	`-checkers-selection-file`
	`misra_c_2012_rules.xml -misra3 from-`
	`file`

.

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C:2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

•

•

•

• "Custom Coding Rules"

• "JSF C++ Rules"

• "MISRA C:2004 Rules"

• "MISRA C:2012 Directives and Rules"

• "MISRA C++:2008 Rules"

See Also
`Do not generate results for (-do-not-generate-results-for)`

Topics

Use generated code requirements (`-misra3-agc-mode`)

Check for violations of MISRA C:2012 rules and directives that apply to generated code

Description

Specify whether to use the MISRA C:2012 categories for automatically generated code. This option changes which rules are mandatory, required, or advisory.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See "Dependency" on page 1-150 for other options that you must also enable.

Command line and options file: Use the option `-misra3-agc-mode`. See "Command-Line Information" on page 1-150.

Why Use This Option

Use this option to specify that you are checking for MISRA C:2012 rules in generated code. The option modifies the MISRA C:2012 subsets so that they are tailored for generated code.

Settings

☐ Off (default)

Use the normal categories (mandatory, required, advisory) for MISRA C:2012 coding guideline checking.

☑ On (default for analyses from Simulink)

Use the generated code categories (mandatory, required, advisory, readability) for MISRA C:2012 coding guideline checking.

For analyses started from the Simulink plug-in, this option is the default value.

Category changed to `Advisory`

These rules are changed to advisory:

- 5.3
- 7.1
- 8.4, 8.5, 8.14
- 10.1, 10.2, 10.3, 10.4, 10.6, 10.7, 10.8
- 14.1, 14.4
- 15.2, 15.3
- 16.1, 16.2, 16.3, 16.4, 16.5, 16.6, 16.7

- 20.8

Category changed to Readability

These guidelines are changed to readability:

- Dir 4.5
- 2.3, 2.4, 2.5, 2.6, 2.7
- 5.9
- 7.2, 7.3
- 9.2, 9.3, 9.5
- 11.9
- 13.3
- 14.2
- 15.7
- 17.5, 17.7, 17.8
- 18.5
- 20.5

Dependency

To use this option, first select the `Check MISRA C:2012 (-misra3)` option.

Command-Line Information
Parameter: `-misra3-agc-mode`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-misra3 all -misra3-agc-mode`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-misra3 all -misra3-agc-mode`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-misra3 all -misra3-agc-mode`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-misra3 all -misra3-agc-mode`

See Also
`Check MISRA C:2012 (-misra3)|Do not generate results for (-do-not-generate-results-for)`

Topics
"Specify Polyspace Analysis Options"
"Check for Coding Standard Violations"
"MISRA C:2012 Directives and Rules"

Check custom rules (`-custom-rules`)

Follow naming conventions for identifiers

Description

Define naming conventions for identifiers and check your code against them.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

Command line and options file: Use the option `-custom-rules`. See "Command-Line Information" on page 1-153.

Why Use This Option

Use this option to impose naming conventions on identifiers. Using a naming convention allows you to easily determine the nature of an identifier from its name. For instance, if you define a naming convention for structures, you can easily tell whether an identifier represents a structured variable or not.

After analysis, the **Results List** pane lists violations of the naming conventions. On the **Source** pane, for every violation, Polyspace assigns a ▼ symbol to the keyword or identifier relevant to the violation.

For the full list of types on which you can apply naming conventions, see "Custom Coding Rules".

Settings

☑ On

Polyspace matches identifiers in your code against text patterns you define. Define the text patterns in a custom coding rules file. To create a coding rules file,

- Use the custom rules wizard:

    1 Click [Edit]. A **Findings selection** window opens.

    2 The **Custom** node in the left pane is highlighted. Expand the nodes in the right pane to select custom rule you want to check.

    3 For every custom rule you want to check:

       a Select **On** ☑.

       b In the **Convention** column, enter the error message you want to display if the rule is violated.

       For example, for rule 4.3, **All struct fields must follow the specified pattern**, you can enter `All struct fields must begin with s_`. This message appears on the **Result Details** pane if:

- You specify the **Pattern** as `s_[A-Za-z0-9_]+`.
- A structure field in your code does not begin with `s_`.

c In the **Pattern** column, enter the text pattern.

For example, for rule 4.3, **All struct fields must follow the specified pattern**, you can enter `s_[A-Za-z0-9_]+`. Polyspace reports violation of rule 4.3 if a structure field does not begin with `s_`.

You can use Perl regular expressions to define patterns. For instance, you can use the following expressions.

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

For frequent patterns, you can use the following regular expressions:

- `(?!__)[a-z0-9_]+(?!__)`, matches a text pattern that does not start and end with two underscores.

  ```
  int __text; //Does not match
  int _text_; //Matches
  ```
- `[a-z0-9_]+_(u8|u16|u32|s8|s16|s32)` , matches a text pattern that ends with a specific suffix.

  ```
  int _text_; //Does not match
  int _text_s16; //Matches
  int _text_s33; // Does not match
  ```
- `[a-z0-9_]+_(u8|u16|u32|s8|s16|s32)(_b3|_b8)?` , matches a text pattern that ends with a specific suffix and an optional second suffix.

  ```
  int _text_s16; //Matches
  int _text_s16_b8; //Matches
  ```

For a complete list of regular expressions, see Perl documentation.

To use or update an existing coding rules file, click Edit to open the **Findings selection** window then do one of the following:

- Enter the full path to the file in the field provided
- Click **Browse** and navigate to the file location.

☐ Off (default)

Polyspace does not check your code against custom naming conventions.

Command-Line Information

Parameter: `-custom-rules`
Value: `from-file`, specify the file using `Set checkers by file (-checkers-selection-file)`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-custom-rules from-file -checkers-selection-file "C:\Standards\custom_config.xml"`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file (only XML format supported)
Not recommended starting in R2019a

Starting in R2019a, the file where you define custom coding rules uses the XML format. You can save selections for custom coding rules and all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your selection for each coding standard and custom coding rules in separate text files. Polyspace will stop supporting custom coding rule files in text format in a future release.

Desktop user interface:

If you have a project that contains custom coding rules and coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click ▢. In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *`filename`*`.xml`, where *`filename`* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `custom_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *`polyspaceroot`*`\help\toolbox\polyspace_bug_finder_server`

\examples\coding_standards_XML. Here, *polyspaceroot* is the root installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace\R2019a`. To update your script, replace reference to the old file format with the new XML file format .

Example of Configuration File in XML Format

To turn on and define custom coding rule 8.1, use this entry:

```
<standard name="CUSTOM RULES">
  ...
  <section name="8 Constants">
      ...
      <check id="8.1" state="on">
       </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

*
*
*
* "Custom Coding Rules"
* "JSF C++ Rules"
* "MISRA C:2004 Rules"
* "MISRA C:2012 Directives and Rules"
* "MISRA C++:2008 Rules"

See Also

Topics
"Specify Polyspace Analysis Options"
"Check for Coding Standard Violations"
"Create Custom Coding Rules"
"Custom Coding Rules"

Effective boolean types (`-boolean-types`)

Specify data types that coding rule checker must treat as effectively Boolean

Description

Specify data types that the coding rule checker must treat as effectively Boolean. You can specify a data type as effectively Boolean only if you have defined it through an `enum` or `typedef` statement in your source code.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See "Dependencies" on page 1-156 for other options that you must also enable.

Command line and options file: Use the option `-boolean-types`. See "Command-Line Information" on page 1-156.

Why Use This Option

Use this option to allow Polyspace to check the following coding rules:

- MISRA C: 2004 and MISRA AC AGC

Rule Number	Rule Statement
12.6	Operands of logical operators, &&, ||, and !, should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to other operators.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
15.4	A `switch` expression should not represent a value that is effectively Boolean.

- MISRA C: 2012

Rule Number	Rule Statement
10.1	Operands shall not be of an inappropriate essential type
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category
10.5	The value of an expression should not be cast to an inappropriate essential type
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.
16.7	A switch-expression shall not have essentially Boolean type.

For example, in the following code, unless you specify `myBool` as effectively Boolean, Polyspace detects a violation of MISRA C: 2012 rule 14.4.

```
typedef int myBool;

void func1(void);
void func2(void);

void func(myBool flag) {
    if(flag)
        func1();
    else
        func2();
}
```

Settings

No Default

Click ⊕ to add a field. Enter a type name that you want Polyspace to treat as Boolean.

Dependencies

This option is enabled only if you select one of these options:

- `Check MISRA C:2004 (-misra2)`
- `Check MISRA AC AGC (-misra-ac-agc).`
- `Check MISRA C:2012 (-misra3)`

Command-Line Information
Parameter: `-boolean-types`
Value: *type1*`[,`*type2*`[,...]]`
No Default
Example (Bug Finder): `polyspace-bug-finder -sources` *filename* `-misra2 required-rules -boolean-types boolean1_t,boolean2_t`
Example (Code Prover): `polyspace-code-prover -sources` *filename* `-misra2 required-rules -boolean-types boolean1_t,boolean2_t`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *filename* `-misra2 required-rules -boolean-types boolean1_t,boolean2_t`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *filename* `-misra2 required-rules -boolean-types boolean1_t,boolean2_t`

See Also
`Check MISRA AC AGC (-misra-ac-agc)`|`Check MISRA C:2004 (-misra2)`|`Check MISRA C:2012 (-misra3)`

Topics
"Specify Polyspace Analysis Options"
"Check for Coding Standard Violations"
"MISRA C:2004 Rules"
"MISRA C:2012 Directives and Rules"

Allowed pragmas (`-allowed-pragmas`)

Specify pragma directives that are documented

Description

Specify pragma directives that must not be flagged by MISRA C:2004 rule 3.4 or MISRA C++ rule 16-6-1. These rules require that you document all pragma directives.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See "Dependencies" on page 1-157 for other options that you must also enable.

Command line and options file: Use the option `-allowed-pragmas`. See "Command-Line Information" on page 1-158.

Why Use This Option

MISRA C:2004/MISRA AC AGC rule 3.4 and MISRA C++ rule 16-6-1 require that all pragma directives are documented within the documentation of the compiler. If you list a pragma as documented using this analysis option, Polyspace does not flag use of the pragma as a violation of these rules.

Settings

No Default

Click ➕ to add a field. Enter the pragma name that you want Polyspace to ignore during coding rule checking .

Dependencies

This option is enabled only if you select one of these options:

- `Check MISRA C:2004 (-misra2)`
- `Check MISRA AC AGC (-misra-ac-agc)`.
- `Check MISRA C++:2008 (-misra-cpp)`

Tips

Enter the name of the pragma only excluding any argument. For instance, if you use the pragma `pack`:

`#pragma pack(`*n*`)`

Enter only the name `pack` for this option.

Command-Line Information

Parameter: `-allowed-pragmas`
Value: *pragma1*`[,`*pragma2*`[,...]]`
Example (Bug Finder): `polyspace-bug-finder -sources` *filename* `-misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`
Example (Code Prover): `polyspace-code-prover -sources` *filename* `-misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *filename* `- misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *filename* `- misra-cpp required-rules -allowed-pragmas pragma_01,pragma_02`

See Also

`Check MISRA AC AGC (-misra-ac-agc)`|`Check MISRA C++:2008 (-misra-cpp)`|`Check MISRA C:2004 (-misra2)`

Topics
"Specify Polyspace Analysis Options"
"Check for Coding Standard Violations"
"MISRA C:2004 Rules"
"MISRA C++:2008 Rules"

Calculate code metrics (`-code-metrics`)

Compute and display code complexity metrics

Description

Specify that Polyspace must compute and display code complexity metrics for your source code. The metrics include file metrics such as number of lines and function metrics such as cyclomatic complexity and estimated size of local variables.

For more information, see "Compute Code Complexity Metrics".

To maintain an acceptable level of software complexity during the development cycle, use the software complexity checkers. See .

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node.

Command line and options file: Use the option `-code-metrics`. See "Command-Line Information" on page 1-160.

Why Use This Option

By default, Polyspace does not calculate code complexity metrics. If you want these metrics in your analysis results, before running analysis, set this option.

High values of code complexity metrics can lead to obscure code and increase chances of coding errors. Additionally, if you run a Code Prover verification on your source code, you might benefit from checking your code complexity metrics first. If a function is too complex, attempts to verify the function can lead to a lot of unproven code. For information on how to cap your code complexity metrics, see "Compute Code Complexity Metrics".

Settings

☑ On

Polyspace computes and displays code complexity metrics on the **Results List** pane.

☐ Off (default)

Polyspace does not compute complexity metrics.

Tips

If you want to compute only the code complexity metrics for your code:

- In Bug Finder, disable checking of defects. See `Find defects (-checkers)`.
- In Code Prover, run verification up to the `Source Compliance Checking` phase. See `Verification level (-to)`.

A Code Prover analysis computes the stack usage metrics after the source compliance checking phase. If you stop a Code Prover verification before source compliance checking, the stack usage metrics are not reported.

Command-Line Information

Parameter: `-code-metrics`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-code-metrics`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-code-metrics`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-code-metrics`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-code-metrics`

See Also

Topics
"Compute Code Complexity Metrics"
"Code Metrics"

Check MISRA C++:2008 (`-misra-cpp`)

Check for violations of MISRA C++ rules

Description

Specify whether to check for violation of MISRA C++ rules. Each value of the option corresponds to a subset of rules to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See "Dependency" on page 1-162 for other options that you must also enable.

Command line and options file: Use the option `-misra-cpp`. See "Command-Line Information" on page 1-162.

Why Use This Option

Use this option to specify the subset of MISRA C++ rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

Settings

Default: `required-rules`

`required-rules`

   Check required coding rules.

`all-rules`

   Check required and advisory coding rules.
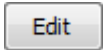
`SQO-subset1`

   Check only a subset of MISRA C++ rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results. For more information, see "Software Quality Objective Subsets (C++)".

`SQO-subset2`

   Check a subset of rules including `SQO-subset1` and some additional rules. In Polyspace Code Prover, observing these rules can further reduce the number of unproven results. For more information, see "Software Quality Objective Subsets (C++)"

`from-file`

   Specify an XML file where you configure a custom selection of checkers for this coding standard.

   To create a configuration file, click [Edit], then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Dependency

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

For projects with mixed C and C++ code, the MISRA C++ checker analyzes only `.cpp` files.

Command-Line Information
Parameter: `-misra-cpp`
Value: `required-rules` | `all-rules` | `SQO-subset1` | `SQO-subset2` | `from-file`
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-misra-cpp all-rules`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-misra-cpp all-rules`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-misra-cpp all-rules`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-misra-cpp all-rules`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file (only XML format supported)
Not recommended starting in R2019a

Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

Desktop interface:

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics** node of the **Configuration** pane, click . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename*`.xml`, where *filename* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

Command-line:

If you do not have access to a Polyspace desktop interface, use the file `misra_cpp_2008_rules.xml` as a template to create the XML file where you define a custom selection of coding standard checkers. This template file is in *polyspaceroot*\help\toolbox \polyspace_bug_finder_server\examples\coding_standards_XML. Here, *polyspaceroot* is the root installation folder for the Polyspace products, for instance, `C:\Program Files \Polyspace\R2019a`. To update your script, see this table

Option	Use Instead
`-misra-cpp "custom_standard.conf"`	`-checkers-selection-file misra_cpp_2008_rules.xml -misra-cpp from-file`

.

Note The XML format of the checker configuration file can change in future releases.

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

•

•

•

• "Custom Coding Rules"

• "JSF C++ Rules"

• "MISRA C:2004 Rules"

• "MISRA C:2012 Directives and Rules"

• "MISRA C++:2008 Rules"

See Also

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

Topics
"Specify Polyspace Analysis Options"
"Check for Coding Standard Violations"
"MISRA C++:2008 Rules"

Check JSF AV C++ rules (`-jsf-coding-rules`)

Check for violations of JSF C++ rules

Description

Specify whether to check for violation of JSF AV C++ rules (JSF++:2005). Each value of the option corresponds to a subset of rules to check.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Coding Standards & Code Metrics** node. See "Dependency" on page 1-165 for other options that you must also enable.

Command line and options file: Use the option `-jsf-coding-rules`. See "Command-Line Information" on page 1-165.

Why Use This Option

Use this option to specify the subset of JSF C++ rules to check for.

After analysis, the **Results List** pane lists the coding standard violations. On the **Source** pane, for every coding rule violation, Polyspace assigns a ▽ symbol to the keyword or identifier relevant to the violation.

Settings

Default: `shall-rules`

`shall-rules`

   Check all **Shall** rules. **Shall** rules are mandatory requirements and require verification.

`shall-will-rules`

   Check all **Shall** and **Will** rules. **Will** rules are intended to be mandatory requirements but do not require verification.

`all-rules`

   Check all **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.

`from-file`

   Specify an XML file where you configure a custom selection of checkers for this coding standard.

   To create a configuration file, click [Edit], then select the rules and recommendations you want to check for this coding standard from the right pane of the **Findings selection** window. Save the file.

   To use or update an existing configuration file, in the **Findings selection** window, enter the full path to the file in the field provided or click **Browse**.

   If you set the option to `from-file`, enable `Set checkers by file (-checkers-selection-file)`.

Tips

- If your project uses a setting other than `generic` for `Compiler (-compiler)`, some rules might not be completely checked. For example, AV Rule 8: "All code shall conform to ISO/IEC 14882:2002(E) standard C++."

Dependency

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

For projects with mixed C and C++ code, the JSF C++ checker analyzes only `.cpp` files.

Command-Line Information

Parameter: `-jsf-coding-rules`
Value: `shall-rules` | `shall-will-rules` | `all-rules` | `from-file`
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-jsf-coding-rules all-rules`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-jsf-coding-rules all-rules`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-jsf-coding-rules all-rules`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-jsf-coding-rules all-rules`

Compatibility Considerations

Polyspace will no longer support text format for coding rules file (only XML format supported)
Not recommended starting in R2019a

Starting in R2019a, the file where you define a custom selection of coding standard checkers uses the XML format. You can save custom selections for all the coding standards that Polyspace supports in the same file.

In previous releases, you saved your custom selection for each coding standard in separate text files. Polyspace will stop supporting custom coding standard files in text format in a future release.

<u>**Desktop interface:**</u>

If you have a project that contains custom coding standard selection files in text format, Polyspace automatically updates and consolidates those files into a single XML file. If your project has conflicting configurations that refer to the same custom selection file, the software saves the consolidated coding standard selection for each configuration to separate XML files.

To update your text files to the XML format manually, in the **Coding Standards & Code Metrics**

node of the **Configuration** pane, click . In the **Findings selection** window, select the files then click **Save Changes**. Polyspace consolidates the files into a single XML files, and saves this file as *filename*`.xml`, where *filename* is the name of the first selected file alphabetically. For instance, if you select `foo.conf` and `bar.conf`, they are saved as `bar.conf.xml`.

<u>**Command-line:**</u>

If you do not have access to a Polyspace desktop interface, use the file
`StandardsConfiguration.xml` as a template to create the XML file where you define a custom
selection of coding standard checkers. This template file is in *polyspaceserverroot*\polyspace
\examples\cxx\Bug_Finder_Example\sources or *polyspaceserverroot*\polyspace
\examples\cxx\Code_Prover_Example\sources. Here, *polyspaceserverroot* is the root
installation folder for the Polyspace products, for instance, `C:\Program Files\Polyspace`
`\R2019a`. To update your script, see this table

Option	Use Instead

`-jsf-coding-rules`
`"custom_standard.conf"`
	`-checkers-selection-file`
	`"custom_standard.conf.xml" -jsf-`
	`coding-rules from-file`

.

Example of Configuration File in XML Format

To turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
      ...
      <check id="8.1" state="on">
      </check>
      ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

•

•

•

• "Custom Coding Rules"

• "JSF C++ Rules"

• "MISRA C:2004 Rules"

• "MISRA C:2012 Directives and Rules"

• "MISRA C++:2008 Rules"

See Also
`Do not generate results for (-do-not-generate-results-for)`

Topics
"Specify Polyspace Analysis Options"
"Check for Coding Standard Violations"
"JSF C++ Rules"

Verify whole application

Stop verification if sources files are incomplete and do not contain a `main` function

Description

This option affects a Code Prover analysis only.

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

Specify that Polyspace verification must stop if a `main` function is not present in the source files.

If you select a Visual C++ setting for `Compiler (-compiler)`, you can specify which function must be considered as `main`. See `Main entry point (-main)`.

Set Option

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

Command line and options file: There is no corresponding command-line option. See "Command-Line Information" on page 1-167.

Settings

◉ On

Polyspace verification stops if it does not find a `main` function in the source files.

◯ Off (default)

Polyspace continues verification even when a `main` function is not present in the source files. If a `main` is not present, it generates a file `__polyspace_main.c` that contains a `main` function.

Tips

If you use this option, your code must contain a `main` function. Otherwise you see the error:

`Error: required main procedure not found`

If your code does not contain a `main` function, use the option `Verify module or library (-main-generator)` to generate a `main` function.

Command-Line Information

Unlike the user interface, by default, a verification from the command line stops if it does not find a `main` function in the source files. If you specify the option `-main-generator`, Polyspace generates a `main` if it cannot find one in the source files.

See Also

`Show global variable sharing and usage only (-shared-variables-mode)` | `Verify module or library (-main-generator)`

Topics
"Specify Polyspace Analysis Options"
"Verify C Application Without main Function"
"Verify C++ Classes"

Show global variable sharing and usage only (`-shared-variables-mode`)

Compute global variable sharing and usage without running full analysis

Description

This option affects a Code Prover analysis only.

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

Specify this option to run a less extensive analysis that computes the global variable sharing and usage in your entire application. The analysis does not verify your code for run-time errors. The analysis results also include coding standards violations if you enable coding standards checking, and code metrics if you enable code metrics computation.

Set Option

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

Command line and options file: Use the option `-shared-variables-mode`. See "Command-Line Information" on page 1-170.

Why Use This Option

You can see global variable sharing and usage without running a full analysis on your entire application that includes run-time error detection. Run-time error detection on an entire application can take a long time.

Settings

☑ On

> Polyspace computes global variable sharing and usage but does not verify your code for run-time errors.

◉ Off (default)

> Polyspace runs a full analysis on your code, including run-time error detection.

Dependencies

- You can use this option only if your program contains a `main` function and you enable the option `Verify whole application` (implicitly set by default at command line).

- When you enable this option, you must also enable at least one of these options.

  - `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`
  - `Tasks (-entry-points)`

- Cyclic tasks (-cyclic-tasks)
- Interrupts (-interrupts)
- ARXML files selection (-autosar-multitasking)
- OIL files selection (-osek-multitasking)

Tips

- After you analyze your complete application to see global variable sharing and usage, run a component-by-component Code Prover analysis to detect run-time errors.
- In the desktop product, you can see all read and write operations on global variables in the "Variable Access" pane.
- In this less extensive analysis mode, the analysis checks for most but not all coding standards violations, and computes most but not all code metrics.

Command-Line Information

Parameter: `-shared-variables-mode`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-shared-variables-mode -enable-concurrency-detection`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-shared-variables-mode -enable-concurrency-detection`

See Also

Topics
"Specify Polyspace Analysis Options"

Introduced in R2019b

Verify initialization section of code only (`-init-only-mode`)

Check initialization code alone for run-time errors and other issues

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must check only the section of code marked as initialization code for run-time errors and other issues.

To indicate the end of initialization code, you enter the line

```
#pragma polyspace_end_of_init
```

in the `main` function (only once). The initialization code starts from the beginning of `main` and continues up to this pragma.

Since compilers ignore unrecognized pragmas, the presence of this pragma does not affect program execution.

Set Option

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

Command line and options file: Use the option `-init-only-mode`. See "Command-Line Information" on page 1-173.

Why Use This Option

Often, issues in the initialization code can invalidate the analysis of the remaining code. You can use this option to check the initialization code alone and fix the issues, and then disable this option to verify the remaining program.

For instance, in this example:

```
#include <limits.h>

int aVar;
const int aConst = INT_MAX;
int anotherVar;

int main() {
      aVar = aConst + 1;
#pragma polyspace_end_of_init
      anotherVar = aVar - 1;
      return 0;
}
```

the overflow in the line `aVar = aConst+1` must be fixed first before the value of `aVar` is used in subsequent code.

Settings

☑ On

> Polyspace checks the code from the beginning of `main` and continues up to the pragma `polyspace_end_of_init`.

◉ Off (default)

> Polyspace checks the complete application beginning from the `main` function.

Dependencies

You can use this option and designate a section of code as initialization code only if:

- Your program contains a `main` function and you use the option `Verify whole application` (implicitly set by default at command line).
- You set `Source code language (-lang)` to `C`.

Note that the pragma must appear only once in the `main` function. The pragma can appear before or after variable declarations but must appear after type definitions (`typedef`-s).

You cannot use this option with the following options:

- `Verify files independently (-unit-by-unit)`
- `Show global variable sharing and usage only (-shared-variables-mode)`

Tips

- Use this option along with the option `Check that global variables are initialized after warm reboot (-check-globals-init)` to thoroughly check the initialization code before checking the remaining program. If you use both options, the verification checks for the following:

  - Definite or possible run-time errors in the initialization code.
  - Whether all non-`const` global variables are initialized along all execution paths through the initialization code.

- Multitasking options are disabled if you check initialization code only because the initialization of global variables is expected to happen before the tasks (threads) begin. As a result, task bodies are not verified.

  See also "Multitasking".

- If you check initialization code only, the analysis truncates execution paths containing the pragma at the location of the pragma but continues to check other execution paths.

  For instance, in this example, the `pragma` appears in an `if` block. A red non-initialized variable check appears on the line `int a = var` because the path containing the initialization stops at the location of the pragma. On the only other remaining path that bypasses the `if` block, the variable `var` is not initialized.

  ```
  int var;

  int func();
  ```

```
int main() {
    int err = func();
    if(err) {
        var = 0;
 #pragma polyspace_end_of_init
    }
    int a = var;
    return 0;
}
```

To avoid these situations, try to place the pragma outside a block. See other suggestions for placement of the pragma in the reference for `Check that global variables are initialized after warm reboot (-check-globals-init)`.

• To determine the initialization of a structure, a regular Code Prover analysis only considers fields that are used.

If you check initialization code only using this option, the analysis covers only a portion of the code and cannot determine if a variable is used beyond this portion. Therefore, the checks for initialization consider all structure fields, whether used or not.

Command-Line Information
Parameter: `-init-only-mode`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-init-only-mode`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-init-only-mode`

See Also
`Check that global variables are initialized after warm reboot (-check-globals-init)`|`Global variable not assigned a value in initialization code`

Topics
"Specify Polyspace Analysis Options"
"Assumptions About Global Variable Initialization" on page 4-15

Introduced in R2020a

Verify module or library (`-main-generator`)

Generate a `main` function if source files are modules or libraries that do not contain a `main`

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must generate a `main` function if it does not find one in the source files.

Set Option

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

Command line and options file: Use the option `-main-generator`. See "Command-Line Information" on page 1-175.

For the analogous option for model generated code, see `Verify model generated code (-main-generator)`.

Why Use This Option

Use this option if you are verifying a module or library. A Code Prover analysis requires a `main` function. When verifying a module or library, your code might not have a `main`.

When you use this option, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Settings

◉ On (default)

Polyspace generates a `main` function if it does not find one in the source files. The generated `main`:

1. Initializes variables specified by `Variables to initialize (-main-generator-writes-variables)`.
2. Before calling other functions, calls the functions specified by `Initialization functions (-functions-called-before-main)`.
3. In all possible orders, calls the functions specified by `Functions to call (-main-generator-calls)`.
4. (C++ only) Calls class methods specified by `Class (-class-analyzer)` and `Functions to call within the specified classes (-class-analyzer-calls)`.

If you do not specify the function and variable options above, the generated `main`:

- Initializes all global variables except those declared with keywords `const` and `static`.
- In all possible orders, calls all functions that are not called anywhere in the source files. Polyspace considers that global variables can be written between two consecutive function calls. Therefore, in each called function, global variables initially have the full range of values allowed by their type.

◉ Off

Polyspace stops if a `main` function is not present in the source files.

Tips

- If a `main` function is present in your source files, the verification uses that `main` function, irrespective of whether you enable or disable this option.

  The option is relevant only if a `main` function is not present in your source files.

- If you use the option `Verify whole application` (default on the command line), your code must contain a `main` function. Otherwise you see the error:

  `Error: required main procedure not found`

  If your code does not contain a `main` function, use this option to generate a `main` function.

- If you specify multitasking options, the verification ignores your specifications for `main` generation. Instead, the verification introduces an empty `main` function.

  For more information on the multitasking options, see "Configuring Polyspace Multitasking Analysis Manually".

Command-Line Information

Parameter: `-main-generator`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-main-generator`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-main-generator`

See Also

`Class (-class-analyzer)`|`Functions to call (-main-generator-calls)`|`Functions to call within the specified classes (-class-analyzer-calls)`|`Initialization functions (-functions-called-before-main)`|`Variables to initialize (-main-generator-writes-variables)`|`Verify whole application`

Topics
"Specify Polyspace Analysis Options"
"Verify C Application Without main Function"

Main entry point (`-main`)

Specify a Microsoft Visual C++ extensions of `main`

Description

This option affects a Code Prover analysis only.

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

Specify the function that you want to use as `main`. If the function does not exist, the verification stops with an error message. Use this option to specify Microsoft Visual C++ extensions of `main`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See "Dependencies" on page 1-176 for other options that you must also enable.

Command line and options file: Use the option `-main`. See "Command-Line Information" on page 1-177.

Settings

Default: `_tmain`

`_tmain`
   Use `_tmain` as entry point to your code.

`wmain`
   Use `wmain` as entry point to your code.

`_tWinMain`
   Use `_tWinMain` as entry point to your code.

`wWinMain`
   Use `wWinMain` as entry point to your code.

`WinMain`
   Use `WinMain` as entry point to your code.

`DllMain`
   Use `DllMain` as entry point to your code.

Dependencies

This option is enabled only if you:

- Set `Source code language (-lang)` to CPP.
- Select `Verify whole application`.

Command-Line Information

Parameter: -main
Value: _tmain | wmain | _tWinMain | wWinMain | WinMain | DllMain
Example (Code Prover): polyspace-code-prover -sources *file_name* -compiler visual14.0 -main _tmain
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -compiler visual14.0 -main _tmain

See Also

Verify module or library (-main-generator) | Verify whole application

Topics
"Specify Polyspace Analysis Options"

Variables to initialize (`-main-generator-writes-variables`)

Specify global variables that you want the generated `main` to initialize

Description

This option affects a Code Prover analysis only.

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

Specify global variables that you want the generated `main` to initialize. Polyspace considers these variables to have any value allowed by their type.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See "Dependencies" on page 1-179 for other options that you must also enable.

Command line and options file: Use the option `-main-generator-writes-variables`. See "Command-Line Information" on page 1-179.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

A Code Prover analysis of a module without a `main` function makes some default assumptions about global variable initialization. The analysis assumes that global variables that are not explicitly initialized can have the full range of values allowed by their data types upon each entry into an uncalled function. For instance, in the example below, which does not have a `main` function, the variable `glob` is assumed to have *all* possible `int` values both in `foo` and `bar` (despite the modification in `foo`). The assumption is a conservative one since the call context of `foo` and `bar`, including which function gets called earlier, is not known.

```
int glob;

int foo() {
    int locFoo = glob;
    glob++;
    return locFoo;
}

int bar() {
    int locBar = glob;
    return locBar;
}
```

To implement this assumption, the generation `main` initializes such global variables to full-range values before calling each otherwise uncalled function. Use this option to modify this default assumption and implement a different initialization strategy for global variables.

Settings

Default:

- C code — `public`
- C++ Code — `uninit`

`uninit`

   C++ Only

   The generated `main` only initializes global variables that you have not initialized during declaration.

`none`

   The generated `main` does not initialize global variables.

   Global variables are initialized according to the C/C+ standard. For instance, `int` or `char` variables are initialized to 0, `float` variables to 0.0, and so on.

`public`

   The generated `main` initializes all global variables except those declared with keywords `static` and `const`.

`all`

   The generated `main` initializes all global variables except those declared with keyword `const`.

`custom`

   The generated `main` only initializes global variables that you specify. Click ![plus icon] to add a field. Enter a global variable name.

Dependencies

You can use this option only if the following are true:

- Your code does not contain a `main` function.
- `Verify module or library (-main-generator)` is selected.

The option is disabled if you enable the option `Ignore default initialization of global variables (-no-def-init-glob)`. Global variables are considered as uninitialized until you explicitly initialize them in the code.

Tips

This option only affects global variables that are defined in the project. If a global variable is declared as `extern`, the analysis considers that the variable can have any value allowed by its data type, irrespective of the value of this option.

Command-Line Information

Parameter: `-main-generator-writes-variables`
Value: `uninit` | `none`|`public`|`all`|`custom=`*variable1*`[,`*variable2*`[,...]]`
Default: (C) `public` | (C++) `uninit`

Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-main-generator -main-generator-writes-variables all`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-main-generator -main-generator-writes-variables all`

See Also

`Verify module or library (-main-generator)`

Topics
"Specify Polyspace Analysis Options"
"Verify C Application Without main Function"

Initialization functions (-functions-called-before-main)

Specify functions that you want the generated `main` to call ahead of other functions

Description

This option affects a Code Prover analysis only.

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

Specify functions that you want the generated `main` to call ahead of other functions.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See "Dependencies" on page 1-182 for other options that you must also enable.

Command line and options file: Use the option `-functions-called-before-main`. See "Command-Line Information" on page 1-182.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option along with the option `Functions to call (-main-generator-calls)` to specify which functions the generated `main` must call. Unless a function is called directly or indirectly from `main`, the software does not analyze the function.

Settings

No Default

Enter function names or choose from a list.

- Click to add a field and enter the function name.

- Click to list functions in your code. Choose functions from the list.

If the function or method is not overloaded, specify the function name. Otherwise, specify the function prototype with arguments. For instance, in the following code, you must specify the prototypes `func(int)` and `func(double)`.

```
int func(int x) {
 return(x * 2);
}
double func(double x) {
 return(x * 2);
}
```

For C++, if the function is:

- A class method: The generated `main` calls the class constructor before calling this function.
- Not a class method: The generated `main` calls this function before calling class methods.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::init(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::init()`.

Dependencies

This option is enabled only if you select **Verify module or library** under **Code Prover Verification** and your code does not contain a `main` function.

Tips

Although these functions are called ahead of other functions, they can be called in arbitrary order. If you want to call your initialization functions in a specific order, manually write a `main` function to call them.

Command-Line Information
Parameter: `-functions-called-before-main`
Value: *function1*[,*function2*[,...]]
No Default
Example 1 (Code Prover): `polyspace-code-prover -sources` *file_name* `-main-generator -functions-called-before-main myfunc`
Example 2 (Code Prover): `polyspace-code-prover -sources` *file_name* `-main-generator -functions-called-before-main myClass::init(int)`
Example 1 (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-main-generator -functions-called-before-main myfunc`
Example 2 (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-main-generator -functions-called-before-main myClass::init(int)`

See Also
`Class (-class-analyzer)|Functions to call (-main-generator-calls)|Functions to call within the specified classes (-class-analyzer-calls)|Variables to initialize (-main-generator-writes-variables)|Verify module or library (-main-generator)`

Topics
"Specify Polyspace Analysis Options"
"Verify C Application Without main Function"
"Verify C++ Classes"

Functions to call (`-main-generator-calls`)

Specify functions that you want the generated `main` to call after the initialization functions

Description

This option affects a Code Prover analysis only.

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

Specify functions that you want the generated `main` to call. The `main` calls these functions after the ones you specify through the option `Initialization functions (-functions-called-before-main)`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See "Dependencies" on page 1-184 for other options that you must also enable.

Command line and options file: Use the option `-main-generator-calls`. See "Command-Line Information" on page 1-184.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option along with the option `Initialization functions (-functions-called-before-main)` to specify which functions the generated `main` must call. Unless a function is called directly or indirectly from `main`, the software does not analyze the function.

Settings

Default: `unused`

`none`

> The generated `main` does not call any function.

`unused`

> The generated `main` calls only those functions that are not called in the source code. It does not call inlined functions.

`all`

> The generated `main` calls all functions except inlined ones.

`custom`

> The generated `main` calls functions that you specify.
>
> Enter function names or choose from a list.
>
> - Click to add a field and enter the function name.

- Click to list functions in your code. Choose functions from the list.

  If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Dependencies

This option is available only if you select `Verify module or library (-main-generator)`.

Tips

- Select `unused` when you use **Code Prover Verification** > **Verify files independently**.
- If you want the generated `main` to call an inlined function, select `custom` and specify the name of the function.
- To verify a multitasking application without a main, select `none`.
- The generated `main` can call the functions in arbitrary order. If you want to call your functions in a specific order, manually write a `main` function to call them.
- To specify instantiations of templates as arguments, run analysis once with the option argument `all`. Search for the template name in the analysis log and use the template name as it appears in the analysis log for the option argument.

  For instance, to specify this template function instantiation as option argument:

  ```
  template <class T>
  T GetMax (T a, T b) {
    T result;
    result = (a>b)? a : b;
    return (result);
  }
  template int GetMax<int>(int, int); // explicit instantiation
  ```

  Run an analysis with the option `-main-generator-calls all`. Search for `getMax` in the analysis log. You see the function format:

  ```
  T1 getMax<int>(T1, T1)
  ```

  To call only this template instantiation, remove the space between the arguments and use the option:

  ```
  -main-generator-calls custom="T1 getMax<int>(T1,T1)"
  ```

Command-Line Information
Parameter: `-main-generator-calls`
Value: none | unused | all | custom=*function1*[,*function2*[,...]]
Default: unused
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-main-generator -main-generator-calls all`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-main-generator -main-generator-calls all`

See Also
`Class (-class-analyzer)`|`Functions to call within the specified classes (-class-analyzer-calls)`|`Initialization functions (-functions-called-before-main)`|`Verify module or library (-main-generator)`

Topics
"Specify Polyspace Analysis Options"
"Verify C Application Without main Function"

Verify files independently (`-unit-by-unit`)

Verify each source file independently of other source files

Description

This option affects a Code Prover analysis only.

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

Specify that each source file must be verified independently of other source files. Each file is verified individually, independent of other files in the module. Verification results can be viewed for the entire project or for individual files.

After you open the verification result for one file, in the user interface of the Polyspace desktop products, you can see a summary of results for all files on the **Dashboard** pane. You can open the results for each file directly from this summary table.

Each result file (with name `ps_results.pscp`) is saved in a subfolder of the results folder. The subfolder has the same name as the source file being analyzed.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See "Dependencies" on page 1-186 for other options that you must also enable.

Command line and options file: Use the option `-unit-by-unit`. See "Command-Line Information" on page 1-187.

Why Use This Option

There are many reasons you might want to verify each source file independently of other files.

For instance, if verification of a project takes very long, you can perform a file by file verification to identify which file is slowing the verification.

Settings

☑ On

   Polyspace creates a separate verification job for each source file.

☐ Off (default)

   Polyspace creates a single verification job for all source files in a module.

Dependencies

This option is enabled only if you select `Verify module or library (-main-generator)`.

Tips

- Code Prover requires a `main` function as the starting point of verification. In the file-by-file mode, because most files do not have a `main`, Code Prover generates a `main` function when required. By default, the generated `main` calls uncalled functions (uncalled non-private methods and out-of-class functions in C++). For more information, see:

  - "Verify C Application Without main Function"
  - "Verify C++ Classes"

- If you perform a file by file verification, you cannot specify multitasking options.

- If your verification for the entire project takes very long, perform a file by file verification. After the verification is complete for a file, you can view the results while other files are still being verified.

- You can generate a report of the verification results for each file or for all the files together. To generate a single report for all files, perform the report generation after verification (and not along with verification using analysis options).

  To generate a single report for all the files in the Polyspace user interface (desktop product only):

  1. Open the results for one file.

  2. Select **Reporting > Run Report**. Before generating the report, select the option **Generate a single report including all unit results**.

- When you perform a file-by-file verification, you can see many instances of unused variables. Some of these variables might be used in other files but show as unused in a file-by-file verification.

  If you want to ignore these results, use a review scope (named set of filters) that filters out unused variables. See "Filter and Group Results in Polyspace Desktop User Interface".

Command-Line Information

Parameter: `-unit-by-unit`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-unit-by-unit`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-unit-by-unit`

See Also

`Common source files (-unit-by-unit-common-source)`

Topics
"Specify Polyspace Analysis Options"

Common source files (`-unit-by-unit-common-source`)

Specify files that you want to include with each source file during a file by file verification

Description

This option affects a Code Prover analysis only.

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

For a file by file verification, specify files that you want to include with each source file verification. These files are compiled once, and then linked to each verification.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See "Dependencies" on page 1-188 for other options that you must also enable.

Command line and options file: Use the option `-unit-by-unit-common-source`. See "Command-Line Information" on page 1-189.

Why Use This Option

There are many reasons you might want to verify each source file independently of other files. For instance, if verification of a project takes very long, you can perform a file by file verification to identify which file is slowing the verification.

If you perform a file by file verification, some of your files might be missing information present in the other files. Place the missing information in a common file and use this option to specify the file for verification. For instance, if multiple source files call the same function, use this option to specify a file that contains the function definition or a function stub. Otherwise, Polyspace uses its own stubs for functions that are called but not defined in the source files. The assumptions behind the Polyspace stubs can be broader than what you want, leading to orange checks.

Settings

No Default

Click to add a field. Enter the full path to a file. Otherwise, use the button to navigate to the file location.

Dependencies

This option is enabled only if you select `Verify files independently (-unit-by-unit)`.

Command-Line Information

Parameter: `-unit-by-unit-common-source`
Value: `file1`[`,file2`[`,...`]]
No Default
Example (Code Prover): `polyspace-code-prover -sources` `file_name` `-unit-by-unit -`
`unit-by-unit-common-source definitions.c`
Example (Code Prover Server): `polyspace-code-prover-server -sources` `file_name` `-`
`unit-by-unit -unit-by-unit-common-source definitions.c`

See Also

`Verify files independently (-unit-by-unit)`

Topics
"Specify Polyspace Analysis Options"

Verify model generated code (`-main-generator`)

Specify that a `main` function must be generated if it is not present in source files

Description

In Bug Finder, use this option only for code generated from MATLAB code or Simulink models.

Specify that Polyspace must generate a `main` function if it does not find one in the source files.

Set Option

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

Command line and options file: Use the option `-main-generator`. See "Command-Line Information" on page 1-190.

Settings

This option is always enabled for code generated from models.

Polyspace generates a `main` function for the analysis. The generated `main` contains cyclic code that executes in a loop. The loop can run an unspecified number of times.

The `main` performs the following functions before the loop begins:

- Initializes variables specified by Parameters (`-variables-written-before-loop`).
- Calls the functions specified by Initialization functions (`-functions-called-before-loop`).

The `main` then performs the following functions in the loop:

- Calls the functions specified by Step functions (`-functions-called-in-loop`).
- Writes to variables specified by Inputs (`-variables-written-in-loop`).

Finally, the `main` calls the functions specified by Termination functions (`-functions-called-after-loop`).

Command-Line Information
Parameter: `-main-generator`
Default: On
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-main-generator ...`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-main-generator ...`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-main-generator ...`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-main-generator ...`

See Also

Initialization functions (`-functions-called-before-loop`)|Inputs (`-variables-written-in-loop`)|Parameters (`-variables-written-before-loop`)|Step functions (`-functions-called-in-loop`)|Termination functions (`-functions-called-after-loop`)|Verify model generated code (`-main-generator`)

Topics
"Configure Advanced Polyspace Options in Simulink"
"How Polyspace Analysis of Generated Code Works"

Parameters (`-variables-written-before-loop`)

Specify variables that the generated `main` must initialize before the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify variables that the generated `main` must initialize before the cyclic code loop begins. Before the loop begins, Polyspace considers these variables to have any value allowed by their type.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node. You see this option only if you open a project configuration from Simulink.

Command line and options file: Use the option `-variables-written-before-loop`. See "Command-Line Information" on page 1-192.

Settings

Default: `none`

`none`

> The generated `main` does not initialize variables.

`all`

> The generated `main` initializes all variables except those declared with keyword `const`.

`custom`

> The generated `main` only initializes variables that you specify. Click to add a field. Enter variable name. For C++ class members, use the syntax `className::variableName`.

Command-Line Information
Parameter: `-variables-written-before-loop`
Value: `none` | `all` | `custom=`*variable1*`[,`*variable2*`[,...]]`
Default: `none`
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-main-generator -variables-written-before-loop all`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-main-generator -variables-written-before-loop all`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-main-generator -variables-written-before-loop all`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-main-generator -variables-written-before-loop all`

See Also
`Inputs (-variables-written-in-loop)`|`Verify model generated code (-main-generator)`

Topics
"Configure Advanced Polyspace Options in Simulink"
"How Polyspace Analysis of Generated Code Works"

Inputs (-variables-written-in-loop)

Specify variables that the generated `main` must initialize in the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify variables that the generated `main` must initialize at the beginning of every iteration of the cyclic code loop. At the beginning of every loop iteration, Polyspace considers these variables to have any value allowed by their type.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node. You see this option only if you open a project configuration from Simulink.

Command line and options file: Use the option `-variables-written-in-loop`. See "Command-Line Information" on page 1-194.

Settings

Default: `none`

`none`

> The generated `main` does not initialize variables.

`all`

> The generated `main` initializes all variables except those declared with keyword `const`.

`custom`

> The generated `main` only initializes variables that you specify. Click to add a field. Enter variable name. For C++ class members, use the syntax `className::variableName`.

Command-Line Information
Parameter: `-variables-written-in-loop`
Value: `none` | `all` | `custom=`*variable1*`[,`*variable2*`[,...]]`
Default: `none`
Example (Bug Finder): `polyspace-bug-finder -sources `*file_name*` -main-generator -variables-written-in-loop all`
Example (Code Prover): `polyspace-code-prover -sources `*file_name*` -main-generator -variables-written-in-loop all`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources `*file_name*` -main-generator -variables-written-in-loop all`
Example (Code Prover Server): `polyspace-code-prover-server -sources `*file_name*` -main-generator -variables-written-in-loop all`

See Also

`Parameters (-variables-written-before-loop)`|`Verify model generated code (-main-generator)`

Topics
"Configure Advanced Polyspace Options in Simulink"
"How Polyspace Analysis of Generated Code Works"

Initialization functions (`-functions-called-before-loop`)

Specify functions that the generated `main` must call before the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify functions that the generated `main` must call before the cyclic code begins.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node. You see this option only if you open a project configuration from Simulink.

Command line and options file: Use the option `-functions-called-before-loop`. See "Command-Line Information" on page 1-196.

Settings

No Default

Click to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::init(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::init()`.

Tips

- If you specify a function for the option `Termination functions (-functions-called-after-loop)`, you cannot specify it for this option.

Command-Line Information
Parameter: `-functions-called-before-loop`
No Default
Value: *function1*[,*function2*[,...]]
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-main-generator -functions-called-before-loop myfunc`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-main-generator -functions-called-before-loop myfunc`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-main-generator -functions-called-before-loop myfunc`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-main-generator -functions-called-before-loop myfunc`

See Also

Step functions (`-functions-called-in-loop`)|Termination functions (`-functions-called-after-loop`)|Verify model generated code (`-main-generator`)

Topics

"Configure Advanced Polyspace Options in Simulink"
"How Polyspace Analysis of Generated Code Works"

Step functions (`-functions-called-in-loop`)

Specify functions that the generated `main` must call in the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify functions that the generated `main` must call in each cycle of the cyclic code.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node. You see this option only if you open a project configuration from Simulink.

Command line and options file: Use the option `-functions-called-in-loop`. See "Command-Line Information" on page 1-198.

Settings

Default: none

The generated `main` does not call functions in the cyclic code.

all

The generated `main` calls all functions except inlined ones. If you specify certain functions for the options **Initialization functions** or **Termination functions**, the generated `main` does not call those functions in the cyclic code.

custom

The generated `main` calls functions that you specify. Click ➕ to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Tips

If you have specified a function for the option `Initialization functions (-functions-called-before-loop)` or `Termination functions (-functions-called-after-loop)`, to call it inside the cyclic code, use `custom` and specify the function name.

Command-Line Information
Parameter: `-functions-called-in-loop`
Value: none | all | custom=*function1*[,*function2*[,...]]
Default: none

Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-main-generator -functions-called-in-loop all`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-main-generator -functions-called-in-loop all`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-main-generator -functions-called-in-loop all`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-main-generator -functions-called-in-loop all`

See Also

`Initialization functions (-functions-called-before-loop)|Termination functions (-functions-called-after-loop)|Verify model generated code (-main-generator)`

Topics
"Configure Advanced Polyspace Options in Simulink"
"How Polyspace Analysis of Generated Code Works"

Termination functions (`-functions-called-after-loop`)

Specify functions that the generated `main` must call after the cyclic code loop

Description

Use this option only for code generated from MATLAB code or Simulink models.

Specify functions that the generated `main` must call after the cyclic code ends.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Code Prover Verification** node. You see this option only if you open a project configuration from Simulink.

Command line and options file: Use the option `-functions-called-after-loop`. See "Command-Line Information" on page 1-200.

Settings

No Default

Click to add a field. Enter function name.

If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Tips

- If you specify a function for the option `Initialization functions (-functions-called-before-loop)`, you cannot specify it for this option.

Command-Line Information
Parameter: `-functions-called-after-loop`
No Default
Value: *function1*[,*function2*[,...]]
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-main-generator -functions-called-after-loop myfunc`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-main-generator -functions-called-after-loop myfunc`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-main-generator -functions-called-after-loop myfunc`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-main-generator -functions-called-after-loop myfunc`

See Also

Initialization functions (-functions-called-before-loop)|Step functions (-functions-called-in-loop)|Verify model generated code (-main-generator)

Topics

"Configure Advanced Polyspace Options in Simulink"

"How Polyspace Analysis of Generated Code Works"

Class (-class-analyzer)

Specify classes that you want to verify

Description

This option affects a Code Prover analysis only.

Specify classes that Polyspace uses to generate a `main`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See "Dependencies" on page 1-202 for other options that you must also enable.

Command line and options file: Use the option `-class-analyzer`. See "Command-Line Information" on page 1-203.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option and the option `Functions to call within the specified classes (-class-analyzer-calls)` to specify the class methods that the generated `main` must call. Unless a class method is called directly or indirectly from `main`, the software does not analyze the method.

Settings

Default: `all`

`all`

> Polyspace can use all classes to generate a `main`. The generated `main` calls methods that you specify using **Functions to call within the specified classes**.

`none`

> The generated `main` cannot call any class method.

`custom`

> Polyspace can use classes that you specify to generate a `main`. The generated `main` calls methods from classes that you specify using **Functions to call within the specified classes**.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.
- `Source code language (-lang)` is set to `CPP` or `C-CPP`.
- `Verify module or library (-main-generator)` is selected.

Tips

If you select none for this option, Polyspace will not verify class methods that you do not call explicitly in your code.

Command-Line Information

Parameter: -class-analyzer
Value: all | none | custom=*class1*[,*class2*,...]
Default: all
Example (Code Prover): polyspace-code-prover -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2

See Also

Analyze class contents only (-class-only) | Functions to call within the specified classes (-class-analyzer-calls) | Skip member initialization check (-no-constructors-init-check) | Verify module or library (-main-generator)

Topics
"Specify Polyspace Analysis Options"
"Verify C++ Classes"

Functions to call within the specified classes (-class-analyzer-calls)

Specify class methods that you want to verify

Description

This option affects a Code Prover analysis only.

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

Specify class methods that Polyspace uses to generate a `main`. The generated `main` can call static, public and protected methods in classes that you specify using the **Class** option.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See "Dependencies" on page 1-205 for other options that you must also enable.

Command line and options file: Use the option `-class-analyzer-calls`. See "Command-Line Information" on page 1-205.

Why Use This Option

If you are verifying a module or library, Code Prover generates a `main` function if one does not exist. If a `main` exists, the analysis uses the existing `main`.

Use this option and the option `Class (-class-analyzer)` to specify the class methods that the generated `main` must call. Unless a class method is called directly or indirectly from `main`, the software does not analyze the method.

Settings

Default: unused

`all`

> The generated `main` calls all public and protected methods. It does not call methods inherited from a parent class.

`all-public`

> The generated `main` calls all public methods. It does not call methods inherited from a parent class.

`inherited-all`

> The generated `main` calls all public and protected methods including those inherited from a parent class.

`inherited-all-public`

> The generated `main` calls all public methods including those inherited from a parent class.

unused

    The generated `main` calls public and protected methods that are not called in the code.

unused-public

    The generated `main` calls public methods that are not called in the code. It does not call methods inherited from a parent class.

inherited-unused

    The generated `main` calls public and protected methods that are not called in the code including those inherited from a parent class.

inherited-unused-public

    The generated `main` calls public methods that are not called in the code including those inherited from a parent class.

custom

    The generated `main` calls the methods that you specify.

    Enter function names or choose from a list.

- Click to add a field and enter the function name.

- Click to list functions in your code. Choose functions from the list.

    If you use the scope resolution operator to specify the function from a particular namespace, enter the fully qualified name, for instance, `myClass::myMethod(int)`. If the function does not have a parameter, use an empty parenthesis, for instance, `myClass::myMethod()`.

Dependencies

You can use this option only if:

- `Source code language (-lang)` is set to CPP or C-CPP.
- `Verify module or library (-main-generator)` is selected.

Command-Line Information

Parameter: `-class-analyzer-calls`
Value: all | all-public | inherited-all | inherited-all-public | unused | unused-public | inherited-unused | inherited-unused-public | custom=*method1*[,*method2*,...]
Default: unused
Example (Code Prover): polyspace-code-prover -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public

See Also

Class (-class-analyzer) | Verify module or library (-main-generator)

Topics
"Specify Polyspace Analysis Options"

"Verify C++ Classes"

Analyze class contents only (`-class-only`)

Do not analyze code other than class methods

Description

This option affects a Code Prover analysis only.

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

Specify that Polyspace must verify only methods of classes that you specify using the option `Class (-class-analyzer)`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See "Dependencies" on page 1-207 for other options that you must also enable.

Command line and options file: Use the option `-class-only`. See "Command-Line Information" on page 1-208.

Why Use This Option

Use this option to restrict the analysis to certain class methods only.

You specify these methods through the options:

- `Class (-class-analyzer)`
- `Functions to call within the specified classes (-class-analyzer-calls)`

When you analyze a module or library, Code Prover generates a `main` function if one does not exist. The `main` function calls class methods using these two options and functions that are not class methods using other options. Code Prover analyzes these methods and functions for robustness to all inputs. If you use this option, Code Prover analyzes the methods only.

Settings

☑ On

> Polyspace verifies the class methods only. It stubs functions out of class scope even if the functions are defined in your code.

☐ Off (default)

> Polyspace verifies functions out of class scope in addition to class methods.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.

- `Source code language (-lang)` is set to CPP or C-CPP.
- `Verify module or library (-main-generator)` is selected.

If you select this option, you must specify the classes using the `Class (-class-analyzer)` option.

Tips

Use this option:

- For robustness verification of class methods. Unless you use this option, Polyspace verifies methods that you call in your code only for your input combinations.
- In case of scaling.

Command-Line Information

Parameter: `-class-only`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -class-only`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `- main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public -class-only`

See Also

`Class (-class-analyzer)|Functions to call within the specified classes (-class-analyzer-calls)|Verify module or library (-main-generator)`

Topics
"Specify Polyspace Analysis Options"
"Verify C++ Classes"

Skip member initialization check (`-no-constructors-init-check`)

Do not check if class constructor initializes class members

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must not check whether each class constructor initializes all class members.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Code Prover Verification** node. See "Dependencies" on page 1-209 for other options that you must also enable.

Command line and options file: Use the option `-no-constructors-init-check`. See "Command-Line Information" on page 1-210.

Why Use This Option

Use this option to disable checks for initialization of class members in constructors.

Settings

☑ On

Polyspace does not check whether each class constructor initializes all class members.

☐ Off (default)

Polyspace checks whether each class constructor initializes all class members. It uses the functions `check_NIV()` and `check_NIP()` in the generated `main` to perform these checks. It checks for initialization of:

- Integer types such as `int`, `char` and `enum`, both `signed` or `unsigned`.
- Floating-point types such as `float` and `double`.
- Pointers.

Dependencies

You can use this option only if all of the following are true:

- Your code does not contain a `main` function.
- `Source code language (-lang)` is set to `CPP` or `C-CPP`.
- `Verify module or library (-main-generator)` is selected.

If you select this option, you must specify the classes using the `Class (-class-analyzer)` option.

Command-Line Information

Parameter: `-no-constructors-init-check`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-main-generator`
`-class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public`
`-no-constructors-init-check`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-`
`main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls`
`unused-public -no-constructors-init-check`

See Also

`Class (-class-analyzer)|Verify module or library (-main-generator)`

Topics
"Specify Polyspace Analysis Options"
"Verify C++ Classes"

Respect types in fields (`-respect-types-in-fields`)

Do not cast nonpointer fields of a structure to pointers

Description

This option affects a Code Prover analysis only.

Specify that structure fields not declared initially as pointers will not be cast to pointers later.

Set Option

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

Command line and options file: Use the option `-respect-types-in-fields`. See "Command-Line Information" on page 1-212.

Why Use This Option

Use this option to identify and forbid casts from nonpointer structure fields to pointers.

Settings

☑ On

The verification assumes that structure fields not declared initially as pointers will not be cast to pointers later.

Code with option off	Code with option on

``c
struct {
    unsigned int x1;
    unsigned int x2;
} S;
 	
void funct(void) {
    int var, *tmp;
    S.x1 = &var;
    tmp = (int*)S.x1;
    *tmp = 1;
    assert(var==1);
}
``
	``c
	struct {
	    unsigned int x1;
	    unsigned int x2;
	} S;
	
	void funct(void) {
	    int var, *tmp;
	    S.x1 = &var;
	    tmp = (int*)S.x1;
	    *tmp = 1;
	    assert(var==1);
	}
	``

In this example, the fields of S are declared as integers but S.x1 is cast to a pointer. With the option turned off, Polyspace allows the cast.	In this example, the fields of S are declared as integers but S.x1 is cast to a pointer. With the option turned on, Polyspace ignores the cast. Therefore, it ignores the initialization of var through the pointer (int*)S.x1 and produces a red **Non-initialized local variable** error when var is read.

☐ Off (default)

> The verification assumes that structure fields can be cast to pointers even when they are not declared as pointers.

Command-Line Information

Parameter: `-respect-types-in-fields`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-respect-types-in-fields`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-respect-types-in-fields`

See Also

`Non-initialized local variable` | `Respect types in global variables (-respect-types-in-globals)`

Topics
"Specify Polyspace Analysis Options"

Respect types in global variables (`-respect-types-in-globals`)

Do not cast nonpointer global variables to pointers

Description

This option affects a Code Prover analysis only.

Specify that global variables not declared initially as pointers will not be cast to pointers later.

Set Option

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

Command line and options file: Use the option `-respect-types-in-globals`. See "Command-Line Information" on page 1-214.

Why Use This Option

Use this option to identify and forbid casts from nonpointer global variables to pointers.

Settings

☑ On

The verification assumes that global variables not declared initially as pointers will not be cast to pointers later.

☐ Off (default)

The verification assumes that global variables can be cast to pointers even when they are not declared as pointers.

Tips

If you select this option, the number of checks in your code can change. You can use this option and the change in results to identify cases where you cast nonpointer variables to pointers.

For instance, in the following example, when you select the option, the results have one less orange check and one more red check.

Code with option off	Code with option on
```int global;``` ```void main(void) {``` ``` int local;``` ``` global = (int)&local;``` ``` *(int*)global = 5;``` ``` assert(local==5);``` ```}``` In this example, `global` is declared as an `int` variable but cast to a pointer. With the option turned off, Polyspace allows the cast.	```int global;``` ```void main(void) {``` ``` int local;``` ``` global = (int)&local;``` ``` *(int*)global = 5;``` ``` assert(local==5);``` ```}``` In this example, `global` is declared as an `int` variable but cast to a pointer. With the option turned on, Polyspace ignores the cast. Therefore, it ignores the initialization of `local` through the pointer `(int*)global` and produces a red **Non-initialized local variable** error when `local` is read.

Command-Line Information

Parameter: `-respect-types-in-globals`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-respect-types-in-globals`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-respect-types-in-globals`

See Also

`Non-initialized local variable` | `Respect types in fields (-respect-types-in-fields)`

Topics
"Specify Polyspace Analysis Options"

Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)

Specify that environment pointers can be unsafe to dereference unless constrained otherwise

Description

This option affects a Code Prover analysis only.

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

Specify that the verification must consider environment pointers as unsafe unless otherwise constrained. Environment pointers are pointers that can be assigned values outside your code.

Environment pointers include:

- Global or `extern` pointers.
- Pointers returned from stubbed functions.

  A function is stubbed if your code does not contain the function definition or you override a function definition by using the option `Functions to stub (-functions-to-stub)`.

- Pointer parameters of functions whose calls are generated by the software.

  A function call is generated if you verify a module or library and the module or library does not have an explicit call to the function. You can also force a function call to be generated with the option `Functions to call (-main-generator-calls)`.

Set Option

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

Command line and options file: Use the option `-stubbed-pointers-are-unsafe`. See "Command-Line Information" on page 1-217.

Why Use This Option

Use this option so that the verification makes more conservative assumptions about pointers from external sources.

If you specify this option, the verification considers that environment pointers can have a `NULL` value. If you read an environment pointer without checking for `NULL`, the **Illegally dereferenced pointer** check shows a potential error in orange. The message associated with the orange check shows the pointer can be `NULL`.

Settings

☑ On

   The verification considers that environment pointers can have a `NULL` value.

☐ Off (default)

The verification considers that environment pointers:

- Cannot have a `NULL` value.
- Points within allowed bounds.

Tips

- Enable this option during the integration phase. In this phase, you provide complete code for verification. Even if an orange check originates from external sources, you are likely to place protections against unsafe pointers from such sources. For instance, if you obtain a pointer from an unknown source, you check the pointer for `NULL` value.

  Disable this option during the unit testing phase. In this phase, you focus on errors originating from your unit.

- If you are verifying code implementation of AUTOSAR runnables, Code Prover assumes that pointer arguments to runnables and pointers returned from `Rte_` functions are not `NULL`. You cannot use this option to change the assumption. See "Run Polyspace on AUTOSAR Code with Conservative Assumptions".

- If you enable this option, the number of orange checks in your code might increase.

Environment Pointers Safe	Environment Pointers Unsafe
The **Illegally dereferenced pointer** check is green. The verification assumes that `env_ptr` is not NULL and any dereference is within allowed bounds. The verification assumes that the result of the dereference is full range. For instance, in this case, the return value has the full range of type `int`. ```\nint func (int *env_ptr) {\n return *env_ptr;\n}\n```	The **Illegally dereferenced pointer** check is orange. The verification assumes that `env_ptr` can be NULL. ```\nint func (int *env_ptr) {\n return *env_ptr;\n}\n```

If you enable this option, the number of gray checks might decrease.

Environment Pointers Safe	Environment Pointers Unsafe
The verification assumes that `env_ptr` is not NULL. The `if` condition is always true and the `else` block is unreachable. ```\n#include <stdlib.h>\nint func (int *env_ptr) {\n if(env_ptr!=NULL)\n return *env_ptr;\n else\n return 0;\n}\n```	The verification assumes that `env_ptr` can be NULL. The `if` condition is not always true and the `else` block can be reachable. ```\n#include <stdlib.h>\nint func (int *env_ptr) {\n if(env_ptr!=NULL)\n return *env_ptr;\n else\n return 0;\n}\n```

- Instead of considering all environment pointers as safe or unsafe, you can individually constrain some of the environment pointers. See the description of **Initialize Pointer** in "External Constraints for Polyspace Analysis".

When you individually constrain a pointer, you first specify an **Init Mode**, and then specify through the **Initialize Pointer** option whether the pointer is `Null`, `Not Null`, or `Maybe Null`. Depending on the **Init Mode**, you can either override the global specification for all environment pointers or not.

- If you set the **Init Mode** of the pointer to `INIT` or `PERMANENT`, your selection for **Initialize Pointer** overrides your specification for this option. For instance, if you specify `Not NULL` for an environment pointer `ptr`, the verification assumes that `ptr` is not NULL even if you specify that environment pointers must be considered unsafe.

- If you set the **Init Mode** to `MAIN GENERATOR`, the verification uses your specification for this option.

  For pointers returned from stubbed functions, the option `MAIN GENERATOR` is not available. If you override the global specification for such a pointer through the **Initialize Pointer** option in constraints, you cannot toggle back to the global specification without changing the **Initialize Pointer** option too.

- If you disable this option, the verification considers that dereferences at all pointer depths are valid.

  For instance, all the dereferences are considered valid in this code:

  ```
  int*** stub(void);

  void func2() {
          int ***ptr = stub();
          int **ptr2 = *ptr;
          int *ptr3 = *ptr2;
  }
  ```

Command-Line Information
Parameter: `-stubbed-pointers-are-unsafe`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-stubbed-pointers-are-unsafe`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-stubbed-pointers-are-unsafe`

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

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

Introduced in R2016b

Consider volatile qualifier on fields (`-consider-volatile-qualifier-on-fields`)

Assume that `volatile` qualified structure fields can have all possible values at any point in code

Description

This option affects a Code Prover analysis only.

Specify that the verification must take into account the `volatile` qualifier on fields of a structure.

Set Option

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

Command line and options file: Use the option `-consider-volatile-qualifier-on-fields`. See "Command-Line Information" on page 1-220.

Why Use This Option

The `volatile` qualifier on a variable indicates that the variable value can change between successive operations even if you do not explicitly change it in your code. For instance, if `var` is a `volatile` variable, the consecutive operations `res = var; res =var;` can result in two different values of `var` being read into `res`.

Use this option so that the verification emulates the `volatile` qualifier for structure fields. If you select this option, the software assumes that a `volatile` structure field has a full range of values at any point in the code. The range is determined only by the data type of the structure field.

Settings

☑ On

The verification considers the `volatile` qualifier on fields of a structure.

In the following example, the verification considers that the field `val1` can have all values allowed for the `int` type at any point in the code.

```
struct myStruct {
    volatile int val1;
    int val2;
};
```

Even if you write a specific value to `val1` and read the variable in the next operation, the variable read results in any possible value.

```
struct myStruct myStructInstance;
myStructInstance.val1 = 1;
assert (myStructInstance.val1 == 1); // Assertion can fail
```

☐ Off (default)

The verification ignores the `volatile` qualifier on fields of a structure.

In the following example, the verification ignores the qualifier on field `val1`.

```
struct myStruct {
    volatile int val1;
    int val2;
};
```

If you write a specific value to `val1` and read the variable in the next operation, the variable read results in that specific value.

```
struct myStruct myStructInstance;
myStructInstance.val1 = 1;
assert (myStructInstance.val1 == 1); // Assertion passes
```

Tips

- If your volatile fields do not represent values read from hardware and you do not expect their values to change between successive operations, disable this option. You are using the `volatile` qualifier for some other reason and the verification does not need to consider full range for the field values.

- If you enable this option, the number of red, gray, and green checks in your code can decrease. The number of orange checks can increase.

In the following example, a red or green check changes to orange or a gray check goes away when the option is used. Considering the `volatile` qualifier changes the check color. These examples use the following structure definition:

```
struct myStruct {
    volatile int field1;
    int field2;
};
```

Color Without Option	Result Without Option	Result With Option
Green	```void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 == 1); }```	```void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 ==1); }```
Red	```void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 != 1); }```	```void main(){ struct myStruct structVal; structVal.field1 = 1; assert(structVal.field1 !=1); }```

Color Without Option	Result Without Option	Result With Option
Gray	```void main(){ struct myStruct structVal; structVal.field1 = 1; if (structVal.field1 != 1) { /* Perform operation */ } }```	```void main(){ struct myStruct structVal; structVal.field1 = 1; if (structVal.field1 != 1) { /* Perform operation */ } }```

- In C++ code, the option also applies to class members.

Command-Line Information

Parameter: `-consider-volatile-qualifier-on-fields`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-consider-volatile-qualifier-on-fields`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-consider-volatile-qualifier-on-fields`

See Also

Topics
"Specify Polyspace Analysis Options"

Introduced in R2016b

Float rounding mode (-float-rounding-mode)

Specify rounding modes to consider when determining the results of floating point arithmetic

Description

This option affects a Code Prover analysis only.

Specify the rounding modes to consider when determining the results of floating-point arithmetic.

Set Option

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

Command line and options file: Use the option -float-rounding-mode. See "Command-Line Information" on page 1-223.

Why Use This Option

The default verification uses the round-to-nearest mode.

Use the rounding mode all if your code contains routines such as fesetround to specify a rounding mode other than round-to-nearest. Although the verification ignores the fesetround specification, it considers all rounding modes including the rounding mode that you specified. Alternatively, for targets that can use extended precision (for instance, using the flag -mfpmath=387), use the rounding mode all. However, for your Polyspace analysis results to agree with run-time behavior, you must prevent use of extended precision through a flag such as -ffloat-store.

Otherwise, continue to use the default rounding mode to-nearest. Because all rounding modes are considered when you specify all, you can have many orange **Overflow** checks resulting from overapproximation.

Settings

Default: to-nearest

to-nearest

  The verification assumes the round-to-nearest mode.

all

  The verification assumes all rounding modes for each operation involving floating-point variables. The following rounding modes are considered: round-to-nearest, round-towards-zero, round-towards-positive-infinity, and round-towards-negative-infinity.

Tips

• The Polyspace analysis uses floating-point arithmetic that conforms to the IEEE® 754 standard. For instance, the arithmetic uses floating point instructions present in the SSE instruction set. The GNU C flag -mfpmath=sse enforces use of this instruction set. If you use the GNU C compiler

with this flag to compile your code, your Polyspace analysis results agree with your run-time behavior.

However, if your code uses extended precision, for instance using the GNU C flag `-mfpmath=387`, your Polyspace analysis results might not agree with your run-time behavior in some corner cases. See some examples of these corner cases in `codeprover_limitations.pdf` in *polyspaceroot* `\polyspace\verifier\code_prover_desktop`. Here, *polyspaceroot* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

To prevent use of extended precision, on targets without SSE support, you can use a flag such as `-ffloat-store`. For your Polyspace analysis, use `all` for rounding mode to account for double rounding.

- The **Overflow** check uses the rounding modes that you specify. For instance, the following table shows the difference in the result of the check when you change your rounding modes.

Rounding mode: `to-nearest`	Rounding mode: `all`

If results of floating-point operations are rounded to nearest values:
 	
• In the first addition operation, `eps1` is just large enough that the value nearest to `FLT_MAX + eps1` is greater than `FLT_MAX`. The **Overflow** check is red.
 	
• In the second addition operation, `eps2` is just small enough that the value nearest to `FLT_MAX + eps2` is `FLT_MAX`. The **Overflow** check is green.
 	
<pre>#include <float.h>
#define eps1 0x1p103
#define eps2 0x0.FFFFFFp103
 	
float func(int ch) {
    float left_op = FLT_MAX;
    float right_op_1 = eps1, \
right_op_2 = eps2;
    switch(ch) {
    case 1:
        return (left_op +\
right_op_1);
    case 2:
        return (left_op +\
right_op_2);
    default:
        return 0;
    }
}</pre>
	Besides to-nearest mode, the **Overflow** check also considers other rounding modes.
	
	• In the first addition operation, in to-nearest mode, the value nearest to `FLT_MAX + eps1` is greater than `FLT_MAX`, so the addition overflows. But if rounded towards negative infinity, the result is `FLT_MAX`, so the addition does not overflow. Combining these two rounding modes, the **Overflow** check is orange.
	
	• In the second addition operation, in to-nearest mode, the value nearest to `FLT_MAX + eps2` is `FLT_MAX`, so the addition does not overflow. But if rounded towards positive infinity, the result is greater than `FLT_MAX`, so the addition overflows. Combining these two rounding modes, the **Overflow** check is orange.
	
	<pre>#include <float.h>
	#define eps1 0x1p103
	#define eps2 0x0.FFFFFFp103
	
	float func(int ch) {
	    float left_op = FLT_MAX;
	     float right_op_1 = eps1, \
	 right_op_2 = eps2;
	    switch(ch) {
	    case 1:
	        return (left_op +\
	right_op_1);
	    case 2:
	        return (left_op +\
	right_op_2);
	    default:
	        return 0;
	    }
	}</pre>

If you set the rounding mode to `all` and obtain an orange **Overflow** check, to determine how the overflow can occur, consider all rounding modes.

Command-Line Information

Parameter: `-float-rounding-mode`
Value: `to-nearest` | `all`
Default: `to-nearest`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-float-rounding-mode all`

Example (Code Prover Server): `polyspace-code-prover-server -sources `*`file_name`*` -float-rounding-mode all`

See Also
`Overflow`

Topics
"Specify Polyspace Analysis Options"

Introduced in R2016a

Allow negative operand for left shifts (`-allow-negative-operand-in-shift`)

Allow left shift operations on a negative number

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow left shift operations on a negative number.

Set Option

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

Command line and options file: Use the option `-allow-negative-operand-in-shift`. See "Command-Line Information" on page 1-225.

Why Use This Option

According to the C99 standard (sec 6.5.7), the result of a left shift operation on a negative number is undefined. Following the standard, the verification produces a red check on left shifts of negative numbers.

If your compiler has a well-defined behavior for left shifts of negative numbers, set this option. Note that allowing left shifts of negative numbers can reduce the cross-compiler portability of your code.

Settings

☑ On

    The verification allows shift operations on a negative number, for instance, `-2 << 2`.

☐ Off (default)

    If a shift operation is performed on a negative number, the verification generates an error.

Command-Line Information

Parameter: `-allow-negative-operand-in-shift`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-allow-negative-operand-in-shift`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-allow-negative-operand-in-shift`

See Also
`Invalid shift operations`

Topics
"Specify Polyspace Analysis Options"

"Modify or Disable Code Prover Run-Time Checks"

Overflow mode for signed integer (`-signed-integer-overflows`)

Specify whether result of overflow is wrapped around or truncated

Description

This option affects a Code Prover analysis only.

Specify whether Polyspace flags signed integer overflows and whether the analysis wraps the result of an overflow or restricts it to its extremum value.

Set Option

User interface (desktop products only): In the **Configuration** pane, the option is on the **Check Behavior** node under **Code Prover Verification**.

Command line and options file: Use the option `-signed-integer-overflows`. See "Command-Line Information" on page 1-230.

Why Use This Option

Use this option to specify whether to check for signed integer overflows and to specify the assumptions the analysis makes following an overflow.

Settings

Default: `forbid`

`forbid`

Polyspace flags signed integer overflows. If the **Overflow** check on an operation is:

- Red, Polyspace does not analyze the remaining code in the current scope.
- Orange, Polyspace analyzes the remaining code in the current scope. Polyspace considers that:

  - After a positive **Overflow**, the result of the operation has an upper bound. This upper bound is the maximum value allowed by the type of the result.
  - After a negative **Overflow**, the result of the operation has a lower bound. This lower bound is the minimum value allowed by the type of the result.

This behavior conforms to the ANSI C (ISO C++) standard.

In the following code, j has values in the range $[1..2^{31}-1]$ before the orange overflow. Polyspace considers that j has even values in the range $[2 \ .. \ 2147483646]$ after the overflow. Polyspace does not analyze the `printf()` statement after the red overflow.

```
#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // Result of * operation overflows
    i = i * 2;
    // Remaing code in current scope not analyzed
    printf("%d", i);
}
void func2()
{

    int j = getVal();
    if (j > 0) {
        // Range of j: [1..2³¹-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [2 .. 2147483646]
        printf("%d", j);
    }
}
```

Note that tooltips on operations with signed integers show (`result is truncated`) to indicate the analysis mode. The message appears even if the **Overflow** check is green.

allow

Polyspace does not flag signed integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow.

In this code, the analysis does not flag any overflow in the code. However, the range of j wraps around to even values in the range [-2³¹..2] or [2..2³¹-2] and the value of i wraps around to -2³¹.

```
#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // i = 2³⁰
    i = i * 2;
    // i = -2³¹
    printf("%d", i);
}
void func2()
{

    int j = getVal();
    if (j > 0) {
        // Range of j: [1..2³¹-1]
        j = j * 2;
        // Range of j: even values in [-2³¹..2] or [2..2³¹-2]
        printf("%d", j);
    }
}
```

Note that tooltips on operations with signed integers show (`result is wrapped`) to indicate the analysis mode. The message appears even if the analysis in this mode does not flag signed integer overflows.

warn-with-wrap-around

Polyspace flags signed integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow.

In the following code, j has values in the range $[1..2^{31}\text{-}1]$ before the orange overflow. Polyspace considers that j has even values in the range $[\text{-}2^{31}..2]$ or $[2..2^{31}\text{-}2]$ after the overflow.

Similarly, i has value 2^{30} before the red overflow and value $\text{-}2^{31}$ after it .

```
#include<stdio.h>

int getVal();

void func1()
{
    int i = 1;
    i = i << 30;
    // i = 2^30
    // Result of * operation overflows
    i = i * 2;
    // i = -2^31
    printf("%d", i);
}
void func2()
{

    int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^31-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [-2^31..2] or [2..2^31-2]
        printf("%d", j);
    }
}
```

Note that tooltips on operations with signed integers show (`result is wrapped`) to indicate the analysis mode. The message appears even if the **Overflow** check is green.

In wrap-around mode, an overflowing value propagates and can lead to a similar overflow several lines later. By default, Code Prover shows only the first of similar overflows. To see all overflows, use the option `-show-similar-overflows`.

Tips

- To check for overflows on conversions from unsigned to signed integers of the same size, set **Overflow mode for unsigned integer** to `forbid` or `warn-with-wrap-around`. If you allow unsigned integer overflows, Polyspace does not flag overflows on conversions and wraps the result of an overflow, even if you check for signed integer overflows.

- In Polyspace Code Prover, overflowing signed constants are wrapped around. This behavior cannot be changed by using the options. If you want to detect overflows with signed constants, use the Polyspace Bug Finder checker `Integer constant overflow`.

Command-Line Information

Parameter: `-signed-integer-overflows`
Value: `forbid` | `allow` | `warn-with-wrap-around`
Default: `forbid`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-signed-integer-overflows allow`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-signed-integer-overflows allow`

See Also
`-show-similar-overflows` | `Overflow` | `Overflow mode for unsigned integer (-unsigned-integer-overflows)`

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"

Introduced in R2018b

Overflow mode for unsigned integer (`-unsigned-integer-overflows`)

Specify whether result of overflow is wrapped around or truncated

Description

This option affects a Code Prover analysis only.

Specify whether Polyspace flags unsigned integer overflows and whether the analysis wraps the result of an overflow or restricts it to its extremum value.

Set Option

User interface (desktop products only): In the **Configuration** pane, the option is on the **Check Behavior** node under **Code Prover Verification**.

Command line and options file: Use the option `-unsigned-integer-overflows`. See "Command-Line Information" on page 1-235.

Why Use This Option

Use this option to specify whether to check for unsigned integer overflows and to specify the assumptions the analysis makes following an overflow.

Settings

Default: `allow`

`forbid`

Polyspace flags unsigned integer overflows. If the **Overflow** check on an operation is:

- Red, Polyspace does not analyze the remaining code in the current scope.
- Orange, Polyspace analyzes the remaining code in the current scope. Polyspace considers that:

  - After a positive **Overflow**, the result of the operation has an upper bound. This upper bound is the maximum value allowed by the type of the result.
  - After a negative **Overflow**, the result of the operation has a lower bound. This lower bound is the minimum value allowed by the type of the result.

In the following code, j has values in the range $[1..2^{32}-1]$ before the orange overflow. Polyspace considers that j has even values in the range $[2 \ .. \ 4294967294]$ after the overflow. Polyspace does not analyze the `printf()` statement after the red overflow.

```
#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // Result of * operation overflows
    i = i * 2;
    // Remaing code in current scope not analyzed
    printf("%u", i);
}
void func2()
{

    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^32-1]
        // Result of * operation may overflow
        j = j * 2;
        // Range of j: even values in [2 .. 4294967294]
        printf("%u", j);
    }
}
```

Note that tooltips on operations with unsigned integers show (`result is truncated`) to indicate the analysis mode. The message appears even if the **Overflow** check is green.

allow

Polyspace does not flag unsigned integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow. For instance, `MAX_INT + 1` wraps to `MIN_INT`. This behavior conforms to the ANSI C (ISO C++) standard.

In this code, the analysis does not flag any overflow in the code. However, the range of `j` wraps around to even values in the range `[0..2^32-2]]` and the value of `i` wraps around to `0`.

```
#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // i = 2^31
    i = i * 2;
    // i = 0
    printf("%u", i);
}
void func2()
{

    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^32-1]
        j = j * 2;
        // Range of j: even values in [0 .. 4294967294]
        printf("%u", j);
    }
}
```

Note that tooltips on operations with unsigned integers show (`result is wrapped`) to indicate the analysis mode. The message appears even if the analysis does not flag unsigned integer overflows.

`warn-with-wrap-around`

Polyspace flags unsigned integer overflows. If an operation results in an overflow, Polyspace analyzes the remaining code but wraps the result of the overflow. For instance, `INT_MAX + 1` wraps to `0`.

In the following code, j has values in the range $[1..2^{32}-1]$ before the orange overflow. Polyspace considers that j has even values in the range `[0 .. 4294967294]` after the overflow.

Similarly, i has value 2^{31} before the red overflow and value `0` after it.

```
#include<stdio.h>

unsigned int getVal();

void func1()
{
    unsigned int i = 1;
    i = i << 31;
    // i = 2^31
    i = i * 2;
    // i = 0
    printf("%u", i);
}
void func2()
{

    unsigned int j = getVal();
    if (j > 0) {
        // Range of j: [1..2^32-1]
        j = j * 2;
        // Range of j: even values in [0 .. 4294967294]
        printf("%u", j);
    }
}
```

Note that tooltips on operations with unsigned integers show `(result is wrapped)` to indicate the analysis mode. The message appears even if the **Overflow** check is green.

In wrap-around mode, an overflowing value propagates and can lead to a similar overflow several lines later. By default, Code Prover shows only the first of similar overflows. To see all overflows, use the option `-show-similar-overflows`.

Tips

- To check for overflows on conversions from unsigned to signed integers of the same size, set **Overflow mode for unsigned integer** to `forbid` or `warn-with-wrap-around`. If you allow unsigned integer overflows, Polyspace does not flag overflows on conversions and wraps the result of an overflow, even if you check for signed integer overflows.

- In Polyspace Code Prover, overflowing unsigned constants are wrapped around. This behavior cannot be changed by using the options. If you want to detect overflows with unsigned constants, use the Polyspace Bug Finder checker `Unsigned integer constant overflow`.

- Code Prover does not show an overflow on bitwise operations on unsigned variables, for instance, in this example:

```
volatile unsigned char Y;
Y = ~Y;
```

The verification considers that such bitwise operations are deliberate on your part and you intend an automatic wrap-around in case the result of the operation overflows.

Command-Line Information
Parameter: `-unsigned-integer-overflows`
Value: `forbid` | `allow` | `warn-with-wrap-around`

Default: `allow`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-unsigned-integer-overflows allow`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-unsigned-integer-overflows allow`

See Also

`-show-similar-overflows`|`Overflow`|`Overflow mode for signed integer (-signed-integer-overflows)`

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"

Introduced in R2018b

Disable checks for non-initialization (`-disable-initialization-checks`)

Disable checks for non-initialized variables and pointers

Description

This option affects a Code Prover analysis only.

Specify that Polyspace Code Prover must not check for non-initialization in your code.

Set Option

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

Command line and options file: Use the option `-disable-initialization-checks`. See "Command-Line Information" on page 1-238.

Why Use This Option

Use this option if you do not want to detect instances of non-initialized variables.

Settings

☑ On

Polyspace Code Prover does not perform the following checks:

- `Non-initialized local variable`: Local variable is not initialized before being read.
- `Non-initialized variable`: Variable other than local variable is not initialized before being read.
- `Non-initialized pointer`: Pointer is not initialized before being read.
- `Return value not initialized`: C function does not return value when expected.

Polyspace assumes that, at declaration:

- Variables have full-range of values allowed by their type.
- Pointers can be `NULL`-valued or point to a memory block at an unknown offset.

☐ Off (default)

Polyspace Code Prover checks for non-initialization in your code. The software displays red checks if, for instance, a variable is not initialized and orange checks if a variable is initialized only on some execution paths.

Tips

- If you select this option, the software does not report most violations of MISRA C:2004 rule 9.1, and `MISRA C:2012 Rule 9.1`.

- If you select this option, the number and type of orange checks in your code can change.

  For instance, the following table shows an additional orange check with the option enabled.

Checks for Non-initialization Enabled	Checks for Non-initialization Disabled
| ```
void func(int flag) {
    int var1,var2;
    if( flag==0) {
        var1=var2;
    }
    else {
        var1=0;
    }
    var2=var1 + 1;
}
``` | ```
void func(int flag) {
    int var1,var2;
    if( flag==0) {
        var1=var2;
    }
    else {
        var1=0;
    }
    var2=var1 + 1;
}
``` |
| In this example, the software produces: | In this example, the software: |
| A red **Non-initialized local variable** check on `var2` in the `if` branch. The verification continues as if only the `else` branch of the `if` statement exists. A green **Non-initialized local variable** check on `var1` in the last statement. `var1` has the assigned value 0. A green **Overflow** check on the `+` operation. | Does not produce **Non-initialized local variable** checks. At initialization, the software assumes that `var2` has full range of `int` values. Following the `if` statement, because the software considers both `if` branches, it assumes that `var1` also has full range of `int` values. Produces an orange **Overflow** check on the `+` operation. For instance, if `var1` has the maximum `int` value, adding 1 to it can cause an overflow. |

Command-Line Information

Parameter: `-disable-initialization-checks`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-disable-initialization-checks`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-disable-initialization-checks`

See Also

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"

Check that global variables are initialized after warm reboot (-check-globals-init)

Check that global variables are assigned values in designed initialization code

Description

This option affects a Code Prover analysis only.

Specify that Polyspace must check whether all non-const global variables (and local static variables) are explicitly initialized at declaration or within a section of code marked as initialization code.

To indicate the end of initialization code, you enter the line

#pragma polyspace_end_of_init

in the `main` function (only once). The initialization code starts from the beginning of `main` and continues up to this pragma.

Since compilers ignore unrecognized pragmas, the presence of this pragma does not affect program execution.

Set Option

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

Command line and options file: Use the option `-check-globals-init`. See "Command-Line Information" on page 1-242.

Why Use This Option

In a warm reboot, to save time, the bss segment of a program, which might hold variable values from a previous state, is not loaded. Instead, the program is supposed to explicitly initialize all non-const variables without default values before execution. You can use this option to delimit the initialization code and verify that all non-const global variables are indeed initialized in a warm reboot.

For instance, in this simple example, the global variable `aVar` is initialized in the initialization code section but the variable `anotherVar` is not.

```
int aVar;
const int aConst = -1;
int anotherVar;

int main() {
     aVar = aConst;
#pragma polyspace_end_of_init
     return 0;
}
```

Settings

☑ On

>Polyspace checks whether all global variables are initialized in the designated initialization code. The initialization code starts from the beginning from `main` and continues up to the pragma `polyspace_end_of_init`.
>
>The results are reported using the check `Global variable not assigned a value in initialization code`.

⦿ Off (default)

>Polyspace does not check for initialization of global variables in a designated code section.
>
>However, the verification continues to check if a variable is initialized at the time of use. The results are reported using the check `Non-initialized variable`.

Dependencies

You can use this option and designate a section of code as initialization code only if:

- Your program contains a `main` function and you use the option `Verify whole application` (implicitly set by default at command line).
- You set `Source code language (-lang)` to C.

Note that the pragma must appear only once in the `main` function. The pragma can appear before or after variable declarations but must appear after type definitions (`typedef`-s).

You cannot use this option with the following options:

- `Disable checks for non-initialization (-disable-initialization-checks)`
- `Verify files independently (-unit-by-unit)`
- `Show global variable sharing and usage only (-shared-variables-mode)`

Tips

- You can use this option along with the option `Verify initialization section of code only (-init-only-mode)` to check the initialization code before checking the remaining program.

  This approach has the following benefits compared to checking the entire code in one run:

  - Run-time errors in the initialization code can invalidate analysis of the remaining code. You can run a comparatively quicker check on the initialization code before checking the remaining program.
  - You can review results of the checker `Global variable not assigned a value in initialization code` relatively easily.

    Consider this example. There is an orange check on `var` because `var` might remain uninitialized when the `if` and `else if` statements are skipped.

    ```
    int var;
    ```

```
int checkSomething(void);
int checkSomethingElse(void);

int main() {
    int local_var;
    if(checkSomething())
    {
        var=0;
    }
    else if(checkSomethingElse()) {
        var=1;
    }
    #pragma polyspace_end_of_init
    var=2;
    local_var = var;
    return 0;
}
```

To review this check and understand when x might be non-initialized, you have to browse through all instances of x on the **Variable Access** pane. If you check the initialization code alone, only the code in bold gets checked and you have to browse through only the instances in the initialization code.

- The check is only as good as your placement of the pragma `polyspace_end_of_init`. For instance:

  - Place the pragma only after initialization code ends.

    Otherwise, a variable might appear falsely uninitialized.

  - Try to place the pragma directly in the `main` function, that is, outside a block. If you place the pragma in a block, the check considers only those paths that end in the block.

    All paths that end in the block might have a variable initialized but paths that skip the block might let the variable go uninitialized. If you do place the pragma in a block, make sure that it is okay if a variable stays uninitialized outside the block.

    For instance, in this example, the variable `var` is initialized on all paths that end at the location of the pragma. The check is green despite the fact that the `if` block might be skipped, letting the variable go uninitialized.

    ```
    int var;

    int func();

    int main() {
        int err = func();
        if(err) {
            var = 0;
     #pragma polyspace_end_of_init
        }
        int a = var;
        return 0;
    }
    ```

    The issue is detected by the checker if you place the pragma after the `if` block ends.

- Do not place the pragma in a loop.

  If you place the pragma in a loop, you can see results that are difficult to interpret. For instance, in this example, both `aVar` and `anotherVar` are initialized in one iteration of the loop. However, the pragma only considers the first iteration of the loop when it shows a green check for initialization. If a variable is initialized on a later iteration, the check is orange.

  ```
  int aVar;
  int anotherVar;

  void main() {
      for(int i=0; i<=1; i++) {
          if(i == 0)
              aVar = 0;
          else
              anotherVar = 0;
          #pragma polyspace_end_of_init
      }
  }
  ```

  The check is red if you verify initialization code alone and do not initialize a variable in the first loop iteration. To avoid these incorrect red or orange checks, do not place the pragma in a loop.

- To determine the initialization of a structure, a regular Code Prover analysis only considers fields that are used.

  If you check initialization code only using the option `Verify initialization section of code only (-init-only-mode)`, the analysis covers only a portion of the code and cannot determine if a variable is used beyond this portion. Therefore, the checks for initialization consider all structure fields, whether used or not.

Command-Line Information
Parameter: `-check-globals-init`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources file_name -check-globals-init`
Example (Code Prover Server): `polyspace-code-prover-server -sources file_name -check-globals-init`

See Also
`Global variable not assigned a value in initialization code` | `Verify initialization section of code only (-init-only-mode)`

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"
"Assumptions About Global Variable Initialization" on page 4-15

Introduced in R2020a

Detect stack pointer dereference outside scope (-detect-pointer-escape)

Find cases where a function returns a pointer to one of its local variables

Description

This option affects a Code Prover analysis only.

Specify that the verification must detect cases where you access a variable outside its scope via dangling pointers. Such an access can happen, for example, when a function returns a pointer to a local variable and you dereference the pointer outside the function. The dereference causes undefined behavior because the local variable that the pointer points to does not live outside the function.

Set Option

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

Command line and options file: Use the option `-detect-pointer-escape`. See "Command-Line Information" on page 1-244.

Why Use This Option

Use this option to enable detection of pointer escape.

Settings

☑ On

The **Illegally dereferenced pointer** check performs an additional task, besides its usual specifications. When you dereference a pointer, the check also determines if you are accessing a variable outside its scope through the pointer. The check is:

- Red, if all the variables that the pointer points to are accessed outside their scope.

  For instance, you dereference a pointer `ptr` in a function `func` that is called twice in your code. In both calls, when you perform the dereference `*ptr`, `ptr` is pointing to variables outside their scope. Therefore, the **Illegally dereferenced pointer** check is red.

- Orange, if only some of the variables that the pointer points to are accessed outside their scope.

- Green, if none of the variables that the pointer points to are accessed outside their scope, and other requirements of the check are also satisfied.

In the following code, if you enable this option, Polyspace Code Prover produces a red **Illegally dereferenced pointer** check on *ptr. Otherwise, the **Illegally dereferenced pointer** check on *ptr is green.

```
void func2(int *ptr) {
    *ptr = 0;
```

```
    }

    int* func1(void) {
        int ret = 0;
        return &ret ;
    }
    void main(void) {
        int* ptr = func1() ;
        func2(ptr) ;
    }
```

The **Result Details** pane displays a message indicating that `ret` is accessed outside its scope.

> **ID 1: Illegally dereferenced pointer**
> Error: pointer is outside its bounds
>   This check may be a path-related issue, which is not dependent on input values
> Dereference of parameter 'ptr' (pointer to int 32, size: 32 bits):
>   Pointer is not null.
>   Points to 4 bytes at offset 0 in buffer of 4 bytes, so is within bounds (if memory is allocated).
>   Pointer may point to variable or field of variable:
>   'ret', local to function 'func1'. 'ret' is accessed outside its scope.

☐ Off (default)

When you dereference a pointer, the **Illegally dereferenced pointer** check does not check for whether you are accessing a variable outside its scope. The check is green even if the pointer dereference is outside the variable scope, as long as it satisfies these requirements:

- The pointer is not NULL.
- The pointer points within the memory buffer.

Tips

The detection of stack pointer deference outside scope does not apply to certain types of pointers. For specific limitations, see "Limitations of Polyspace Verification" on page 4-41.

Command-Line Information
Parameter: `-detect-pointer-escape`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-detect-pointer-escape`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-detect-pointer-escape`

See Also
`Illegally dereferenced pointer`

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"

Introduced in R2015a

Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)

Allow arithmetic on pointer to a structure field so that it points to another field

Description

This option affects a Code Prover analysis only.

Specify that a pointer assigned to a structure field can point outside its bounds as long as it points within the structure.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See "Dependency" on page 1-246 for other options you must also enable.

Command line and options file: Use the option -allow-ptr-arith-on-struct. See "Command-Line Information" on page 1-246.

Why Use This Option

Use this option to relax the check for illegally dereferenced pointers. Once you assign a pointer to a structure field, you can perform pointer arithmetic and use the result to access another structure field.

Settings

☑ On

A pointer assigned to a structure field can point outside the bounds imposed by the field as long as it points within the structure. For instance, in the following code, unless you use this option, the verification will produce a red Illegally dereferenced pointer check:

```
void main(void) {
struct S {char a; char b; int c;} x;
char *ptr = &x.b;
ptr ++;
*ptr = 1; // Red on the dereference, because ptr points outside x.b
}
```

☐ Off (default)

A pointer assigned to a structure field can point only within the bounds imposed by the field.

Tips

- The verification does not allow a pointer with negative offset values. This behavior occurs irrespective of whether you choose the option **Enable pointer arithmetic across fields**.

- Using this option can slightly increase the number of orange checks. The option relaxes the constraint that a pointer to a structure field cannot point to other fields of the structure. In

1-245

exchange for relaxing this constraint, the verification loses precision on the boundary of fields within a structure and treats the structure as a whole. Pointer dereferences that were previously green can now turn orange.

Use this option if you follow a policy of reviewing red checks only and you need to work around red checks from pointer arithmetic within a structure.

- Before using this option, consider the costs of using pointer arithmetic across different fields of a structure.

Unlike an array, members of a structure can have different data types. For efficient storage, structures use padding to accommodate this difference. When you increment a pointer pointing to a structure member, you might not point to the next member. When you dereference this pointer, you cannot rely on what you are reading or writing to.

Dependency

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

Command-Line Information
Parameter: `-allow-ptr-arith-on-struct`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-allow-ptr-arith-on-struct`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-allow-ptr-arith-on-struct`

See Also
`Allow incomplete or partial allocation of structures (-size-in-bytes)` | `Illegally dereferenced pointer`

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"

Allow incomplete or partial allocation of structures (`-size-in-bytes`)

Allow a pointer with insufficient memory buffer to point to a structure

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow dereferencing a pointer that points to a structure but has a sufficient buffer for only some of the structure's fields.

This type of pointer results when a pointer to a smaller structure is cast to a pointer to a larger structure. The pointer resulting from the cast has sufficient buffer for only some fields of the larger structure.

Set Option

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

Command line and options file: Use the option `-size-in-bytes`. See "Command-Line Information" on page 1-248.

Why Use This Option

Use this option to relax the check for illegally dereferenced pointers. You can point to a structure even when the buffer allowed for the pointer is not sufficient for all the structure fields.

Settings

☑ On

When a pointer with insufficient buffer is dereferenced, Polyspace does not produce an **Illegally dereferenced pointer** error, as long as the dereference occurs within allowed buffer.

For instance, in the following code, the pointer p has sufficient buffer for the first two fields of the structure BIG. Therefore, with the option on, Polyspace considers that the first two dereferences are valid. The third dereference takes p outside its allowed buffer. Therefore, Polyspace produces an **Illegally dereferenced pointer** error on the third dereference.

```
#include <stdlib.h>

typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;

void main(void) {
   BIG *p = malloc(sizeof(LITTLE));

   if (p!= ((void *) 0) ) {
      p->a = 0 ;
      p->b = 0 ;
```

```
      p->c = 0 ;    // Red IDP check
    }
  }
```

☐ Off (default)

Polyspace does not allow dereferencing a pointer to a structure if the pointer does not have sufficient buffer for all fields of the structure. It produces an **Illegally dereferenced pointer** error the first time you dereference the pointer.

For instance, in the following code, even though the pointer p has sufficient buffer for the first two fields of the structure BIG, Polyspace considers that dereferencing p is invalid.

```
#include <stdlib.h>

typedef struct _little { int a; int b; } LITTLE;
typedef struct _big { int a; int b; int c; } BIG;

void main(void) {
   BIG *p = malloc(sizeof(LITTLE));

   if (p!= ((void *) 0) ) {
      p->a = 0 ;    // Red IDP check
      p->b = 0 ;
      p->c = 0 ;
   }
}
```

Tips

*   If you do not turn on this option, you cannot point to the field of a partially allocated structure.

    For instance, in the preceding example, if you do not turn on the option and perform the assignment

    ```
    int *ptr = &(p->a);
    ```

    Polyspace considers that the assignment is invalid. If you dereference ptr, it produces an **Illegally dereferenced pointer** error.

*   Using this option can slightly increase the number of orange checks.

Command-Line Information

Parameter: `-size-in-bytes`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-size-in-bytes`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-size-in-bytes`

See Also

`Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)` |
`Illegally dereferenced pointer`

Topics
"Specify Polyspace Analysis Options"

"Modify or Disable Code Prover Run-Time Checks"

Permissive function pointer calls (`-permissive-function-pointer`)

Allow type mismatch between function pointers and the functions they point to

Description

This option affects a Code Prover analysis only.

Specify that the verification must allow function pointer calls where the type of the function pointer does not match the type of the function.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See "Dependency" on page 1-252 for other options you must also enable.

Command line and options file: Use the option `-permissive-function-pointer`. See "Command-Line Information" on page 1-252.

Why Use This Option

By default, Code Prover does not recognize calls through function pointers when a type mismatch occurs. Fix the type mismatch whenever possible.

Use this option if:

- You cannot fix the type mismatch, and
- The analysis does not cover a significant portion of your code because calls via function pointers are not recognized.

Settings

☑ On

The verification must allow function pointer calls where the type of the function pointer does not match the type of the function. For instance, a function declared as `int f(int*)` can be called by a function pointer declared as `int (*fptr)(void*)`.

Only type mismatches between pointer types are allowed. Type mismatches between nonpointer types cause compilation errors. For instance, a function declared as `int f(int)` cannot be called by a function pointer declared as `int (*fptr)(double)`.

☐ Off (default)

The verification must require that the argument and return types of a function pointer and the function it calls are identical.

Type mismatches are detected with the check `Correctness condition`.

Tips

- With sources that use function pointers extensively, enabling this option can cause loss in performance. This loss occurs because the verification has to consider more execution paths.

- Using this option can increase the number of orange checks. Some of these orange checks can reveal a real issue with the code.

  Consider these examples where a type mismatch occurs between the function pointer type and the function that it points to:

  - In this example, the function pointer `obj_fptr` has an argument that is a pointer to a three-element array. However, it points to a function whose corresponding argument is a pointer to a four-element array. In the body of `foo`, four array elements are read and incremented. The fourth element does not exist and the `++` operation reads a meaningless value.

    ```
    typedef int array_three_elements[3];
    typedef void (*fptr)(array_three_elements*);

    typedef int array_four_elements[4];
    void foo(array_four_elements*);

    void main() {
     array_three_elements arr[3] = {0,0,0};
     array_three_elements *ptr;
     fptr obj_fptr;

     ptr = &arr;
     obj_fptr = &foo;

     //Call via function pointer
     obj_fptr(&ptr);
    }

    void foo(array_four_elements* x) {
        int i = 0;
        int *current_pos;

        for(i = 0; i< 4; i++) {
          current_pos = (*x) + i;
          (*current_pos)++;
        }
    }
    ```

    Without this option, an orange `Correctness condition` check appears on the call `obj_fptr(&ptr)` and the function `foo` is not verified. If you use this option, the body of `foo` contains several orange checks. Review the checks carefully and make sure that the type mismatch does not cause issues.

  - In this example, the function pointer has an argument that is a pointer to a structure with three `float` members. However, the corresponding function argument is a pointer to an unrelated structure with one array member. In the function body, the `strlen` function is used assuming the array member. Instead the `strlen` call reads the `float` members and can read meaningless values, for instance, values stored in the structure padding.

```
#include <string.h>
struct point {
  float x;
  float y;
  float z;
};
struct message {
  char msg[10] ;
};
void foo(struct message*);

void main() {
  struct point pt = {3.14, 2048.0, -1.0} ;
  void (*obj_fptr)(struct point *) ;

  obj_fptr = &foo;

  //Call via function pointer
  obj_fptr(&pt);
}

void foo(struct message* x) {
  int y = strlen(x->msg) ;
}
```

Without this option, an orange `Correctness condition` check appears on the call `obj_fptr(&pt)` and the function `foo` is not verified. If you use this option, the function contains an orange check on the `strlen` call. Review the check carefully and make sure that the type mismatch does not cause issues.

Dependency

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

Command-Line Information
Parameter: `-permissive-function-pointer`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-lang c -permissive-function-pointer`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-lang c -permissive-function-pointer`

See Also
`Correctness condition`

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"

Consider non finite floats (`-allow-non-finite-floats`)

Enable an analysis mode that incorporates infinities and NaNs

Description

Enable an analysis mode that incorporates infinities and NaNs for floating point operations.

Set Option

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

Command line and options file: Use the option `-allow-non-finite-floats`. See "Command-Line Information" on page 1-255.

Why Use This Option

Code Prover

By default, the analysis does not incorporate infinities and NaNs. For instance, the analysis terminates the execution thread where a division by zero occurs and does not consider that the result could be infinite.

If you use functions such as `isinf` or `isnan` and account for infinities and NaNs in your code, set this option. When you set this option and a division by zero occurs for instance, the execution thread continues with infinity as the result of the division.

Set this option alone if you are sure that you have accounted for infinities and NaNs in your code. Using the option alone effectively disables many numerical checks on floating point operations. If you have generally accounted for infinities and NaNs, but you are not sure that you have considered all situations, set these additional options:

- Infinities (`-check-infinite`): Use `warn-first`.
- NaNs (`-check-nan`): Use `warn-first`.

Bug Finder

If the analysis flags comparisons using `isinf` or `isnan` as dead code, use this option. By default, a Bug Finder analysis does not incorporate infinities and NaNs.

Settings

☑ On

The analysis allows infinities and NaNs. For instance, in this mode:

- The analysis assumes that floating-point operations can produce results such as infinities and NaNs.

By using options `Infinities (-check-infinite)` and `NaNs (-check-nan)`, you can choose to highlight operations that produce nonfinite results and stop the execution threads where the nonfinite results occur. These options are not available for a Bug Finder analysis.

- The analysis assumes that floating-point variables with unknown values can have any value allowed by their type, including infinite or NaN. Floating-point variables with unknown values include volatile variables and return values of stubbed functions.

☐ Off (default)

The analysis does not allow infinities and NaNs. For instance, in this mode:

- The Code Prover analysis produces a red check on a floating-point operation that produces an infinity or a NaN as the only possible result on all execution paths. The verification produces an orange check on a floating-point operation that can potentially produce an infinity or NaN.
- The Code Prover analysis assumes that floating-point variables with unknown values are full-range but finite.
- The Bug Finder analysis shows comparisons with infinity using `isinf` as dead code.

Tips

- The IEEE 754 Standard allows special quantities such as infinities and `NaN` so that you can handle certain numerical exceptions without aborting the code. Some implementations of the C standard support infinities and `NaN`.

  - If your compiler supports infinities and `NaN`s and you account for them explicitly in your code, use this option so that the verification also allows them.

    For instance, if a division results in infinity, in your code, you specify an alternative action. Therefore, you do not want the verification to highlight division operations that result in infinity.

  - If your compiler supports infinities and `NaN`s but you are not sure if you account for them explicitly in your code, use this option so that the verification incorporates infinities and `NaN`s. Use the options `-check-nan` and `-check-infinite` with argument `warn` so that the verification highlights operations that result in infinities and `NaN`s, but does not stop the execution thread. These options are not available for a Bug Finder analysis.

- If you run a Code Prover analysis and use this option, checkers for overflow, division by zero and other numerical run-time errors are disabled. See "Numerical Checks".

  If you run a Bug Finder analysis and use this option:

  - The checkers for overflow and division by zero are disabled. See "Numerical Defects".
  - The checker `Floating point comparison with equality operators` can show false positives.

- If you select this option, the number and type of Code Prover checks in your code can change.

  For instance, in the following example, when you select the option, the results have one less red check and three more green checks.

Infinities and NaNs Not Allowed	Infinities and NaNs Allowed
Code Prover produces a **Division by zero** error and stops verification. ```double func(void) { double x=1.0/0.0; double y=1.0/x; double z=x-x; return z;}```	If you select this option, Code Prover does not check for a **Division by zero** error. ```double func(void) { double x=1.0/0.0; double y=1.0/x; double z=x-x; return z;}``` The analysis assumes that dividing by zero results in: • Value of x equal to Inf • Value of y equal to 0.0 • Value of z equal to NaN In your analysis results in the Polyspace user interface, if you place your cursor on y and z, you can see the nonfinite values Inf and NaN respectively in the tooltip.

Command-Line Information

Parameter: `-allow-non-finite-floats`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-allow-non-finite-floats`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-allow-non-finite-floats`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-allow-non-finite-floats`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-allow-non-finite-floats`

See Also

"Numerical Defects" | "Numerical Checks" | `Infinities (-check-infinite)` | `NaNs (-check-nan)`

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"

Introduced in R2016a

Infinities (-check-infinite)

Specify how to handle floating-point operations that result in infinity

Description

This option affects a Code Prover analysis only.

Specify how the analysis must handle floating-point operations that result in infinities.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See "Dependencies" on page 1-257 for other options you must also enable.

Command line and options file: Use the option `-check-infinite`. See "Command-Line Information" on page 1-257.

Why Use This Option

Use this option to enable detection of floating-point operations that result in infinities.

If you specify that the analysis must consider nonfinite floats, by default, the analysis does not flag these operations. Use this option to detect these operations while still incorporating nonfinite floats.

Settings

Default: `allow`

`allow`

The verification does not produce a check on the operation.

For instance, in the following code, there is no **Overflow** check.

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

`warn-first`

The verification produces a check on the operation. The check determines if the result of the operation is infinite when the operands themselves are not infinite. The verification does not terminate the execution thread that produces infinity.

If the verification detects an operation that produces infinity as the only possible result on all execution paths and the operands themselves are never infinite, the check is red. If the operation can potentially result in infinity, the check is orange.

For instance, in the following code, there is a nonblocking **Overflow** check for infinity.

```
double func(void) {
    double x=1.0/0.0;
```

```
        return x;
    }
```

Even though the **Overflow** check on the / operation is red, the verification continues. For instance, a green **Non-initialized local variable** check appears on x in the `return` statement.

forbid

The verification produces a check on the operation and terminates the execution thread that produces infinity.

If the check is red, the verification does not continue for the remaining code in the same scope as the check. If the check is orange, the verification continues but removes from consideration the variable values that produced infinity.

For instance, in the following code, there is a blocking **Overflow** check for infinity.

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

The verification stops because the **Overflow** check on the / operation is red. For instance, a **Non-initialized local variable** check does not appear on x in the `return` statement.

Dependencies

To use this option, you must enable the verification mode that incorporates infinities and NaNs. See Consider non finite floats (-allow-non-finite-floats).

Command-Line Information

Parameter: -check-infinite
Value: allow | warn-first | forbid
Default: allow
Example (Code Prover): polyspace-code-prover -sources *file_name* -check-infinite forbid
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -check-infinite forbid

See Also

Polyspace Analysis Options
Consider non finite floats (-allow-non-finite-floats) | NaNs (-check-nan)

Polyspace Results
Overflow

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"

Introduced in R2016a

NaNs (-check-nan)

Specify how to handle floating-point operations that result in NaN

Description

This option affects a Code Prover analysis only.

Specify how the analysis must handle floating-point operations that result in NaN.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Check Behavior** node. See "Dependencies" on page 1-259 for other options you must also enable.

Command line and options file: Use the option `-check-nan`. See "Command-Line Information" on page 1-259.

Why Use This Option

Use this option to enable detection of floating-point operations that result in NaN-s.

If you specify that the analysis must consider nonfinite floats, by default, the analysis does not flag these operations. Use this option to detect these operations while still incorporating nonfinite floats.

Settings

Default: `allow`

`allow`

> The verification does not produce a check on the operation.
>
> For instance, in the following code, there is no **Invalid operation on floats** check.
>
> ```
> double func(void) {
>     double x=1.0/0.0;
>     double y=x-x;
>     return y;
> }
> ```

`warn-first`

> The verification produces a check on the operation. The check determines if the result of the operation is NaN when the operands themselves are not NaN. For instance, the check flags the operation `val1 + val2` only if the result can be NaN when *both* `val1` and `val2` are not NaN. The verification does not terminate the execution thread that produces NaN.
>
> If the verification detects an operation that produces NaN as the only possible result on all execution paths and the operands themselves are never NaN, the check is red. If the operation can potentially result in NaN, the check is orange.
>
> For instance, in the following code, there is a nonblocking **Invalid operation on floats** check for NaN.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

Even though the **Invalid operation on floats** check on the - operation is red, the verification continues. For instance, a green **Non-initialized local variable** check appears on y in the `return` statement.

forbid

The verification produces a check on the operation and terminates the execution thread that produces NaN.

If the check is red, the verification does not continue for the remaining code in the same scope as the check. If the check is orange, the verification continues but removes from consideration the variable values that produced a NaN.

For instance, in the following code, there is a blocking **Invalid operation on floats** check for NaN.

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

The verification stops because the **Invalid operation on floats** check on the - operation is red. For instance, a **Non-initialized local variable** check does not appear on y in the `return` statement.

The **Invalid operation on floats** check for NaN also appears on the / operation and is green.

Dependencies

To use this option, you must enable the verification mode that incorporates infinities and NaNs. See Consider non finite floats (-allow-non-finite-floats).

Command-Line Information
Parameter: -check-nan
Value: allow|warn-first|forbid
Default: allow
Example (Code Prover): polyspace-code-prover -sources *file_name* -check-nan forbid
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -check-nan forbid

See Also

Polyspace Analysis Options
Consider non finite floats (-allow-non-finite-floats)|Infinities (-check-infinite)

Polyspace Results
Invalid operation on floats

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"

Introduced in R2016a

Subnormal detection mode (`-check-subnormal`)

Detect operations that result in subnormal floating-point values

Description

This option affects a Code Prover analysis only.

Specify that the verification must check floating-point operations for subnormal results.

Set Option

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

Command line and options file: Use the option `-check-subnormal`. See "Command-Line Information" on page 1-263.

Why Use This Option

Use this option to detect floating-point operations that result in subnormal values.

Subnormal numbers have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the significand. The presence of subnormal numbers indicates loss of significant digits. This loss can accumulate over subsequent operations and eventually result in unexpected values. Subnormal numbers can also slow down the execution on targets without hardware support.

Settings

Default: `allow`

`allow`

The verification does not check operations for subnormal results.

`forbid`

The verification checks for subnormal results.

The verification stops the execution path with the subnormal result and prevents subnormal values from propagating further. Therefore, in practice, you see only the first occurrence of the subnormal value.

`warn-all`

The verification checks for subnormal results and highlights all occurrences of subnormal values. Even if a subnormal result comes from previous subnormal values, the result is highlighted.

The verification continues even if the check is red.

`warn-first`

The verification checks for subnormal results but only highlights first occurrences of subnormal values. If a subnormal value propagates to further subnormal results, those subsequent results are not highlighted.

The verification continues even if the check is red.

For details of the result colors in each mode, see `Subnormal float`.

Tips

- If you want to see only those operations where a subnormal value originates from non-subnormal operands, use the `warn-first` mode.

  For instance, in the following code, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`. This assumption can lead to subnormal results from certain operations. If you use the `warn-first` mode, the first operation causing the subnormal result is highlighted.

warn-all	warn-first
``void func (double arg1, double arg2)
{
double difference1 = arg1 - arg2;
double difference2 = arg1 - arg2;
double val1 = difference1 * 2;
double val2 = difference2 * 2;
}``` In this example, all four operations can have subnormal results. The four checks for subnormal results are orange.	```void func (double arg1, double arg2)
{
double difference1 = arg1 - arg2;
double difference2 = arg1 - arg2;
double val1 = difference1 * 2;
double val2 = difference2 * 2;
}``` In this example, `difference1` and `difference2` can be subnormal if `arg1` and `arg2` are sufficiently close. The first two checks for subnormal results are orange. `val1` and `val2` cannot be subnormal unless `difference1` and `difference2` are subnormal. The last two checks for subnormal results are green. Through red/orange checks, you see only the first instance where a subnormal value appears. You do not see red/orange checks from those subnormal values propagating to subsequent operations.

- If you want to see where a subnormal value originates and do not want to see subnormal results arising from the same cause more than once, use the `forbid` mode.

  For instance, in the following code, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`. This assumption can lead to subnormal results for `arg1-arg2`. If you use the `forbid` mode and perform the operation `arg1-arg2` twice in succession, only the first operation is highlighted. The second operation is not highlighted because the subnormal result for the second operation arises from the same cause as the first operation.

warn-all	forbid

```void func (double arg1, double arg2)\n{\n   double difference1 = arg1 - arg2;\n   double difference2 = arg1 - arg2;\n   double val1 = difference1 * 2;\n   double val2 = difference2 * 2;\n}```
 	
In this example, all four operations can have subnormal results. The four checks for subnormal results are orange.
	```void func (double arg1, double arg2)\n{\n double difference1 = arg1 - arg2;\n double difference2 = arg1 - arg2;\n double val1 = difference1 * 2;\n double val2 = difference2 * 2;\n}```
	
	In this example, `difference1` can be subnormal if `arg1` and `arg2` are sufficiently close. The first check for subnormal results is orange. Following this check, the verification excludes from consideration:
	
	• The close values of `arg1` and `arg2` that led to the subnormal value of `difference1`.
	
	  In the subsequent operation `arg1 - arg2`, the check is green and `difference2` is not subnormal. The result of the check on `difference2 * 2` is green for the same reason.
	• The subnormal value of `difference1`.
	
	  In the subsequent operation `difference1 * 2`, the check is green.

Command-Line Information

Parameter: `-check-subnormal`
Value: `allow | warn-first | warn-all | forbid`
Default: `allow`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-check-subnormal forbid`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-check-subnormal forbid`

See Also

Polyspace Results
`Subnormal float`

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"

Introduced in R2016b

Detect uncalled functions (`-uncalled-function-checks`)

Detect functions that are not called directly or indirectly from `main` or another entry point function

Description

This option affects a Code Prover analysis only.

Detect functions that are not called directly or indirectly from `main` or another entry point function during run-time.

Set Option

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

Command line and options file: Use the option `-uncalled-function-checks`. See "Command-Line Information" on page 1-264.

Why Use This Option

Typically, after verification, the **Dashboard** pane shows functions that are not called during verification. However, you do not see them in your analysis results or reports. You cannot comment on them or justify them.

If you want to see these uncalled functions in your analysis results and reports, use this option.

Settings

Default: `none`

`none`
    The verification does not generate checks for uncalled functions.

`never-called`
    The verification generates checks for functions that are defined but not called.

`called-from-unreachable`
    The verification generates checks for functions that are defined and called from an unreachable part of the code.

`all`
    The verification generates checks for functions that are:

- Defined but not called
- Defined and called from an unreachable part of the code.

Command-Line Information
Parameter: `-uncalled-function-checks`

Value: `none` | `never-called` | `called-from-unreachable` | `all`
Default: `none`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-uncalled-function-checks all`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-uncalled-function-checks all`

See Also

`Function not called` | `Function not reachable`

Topics
"Specify Polyspace Analysis Options"
"Modify or Disable Code Prover Run-Time Checks"

Precision level (-0)

Specify a precision level for the verification

Description

This option affects a Code Prover analysis only.

Specify the precision level that the verification must use.

Set Option

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

Command line and options file: Use the option `-0#`, for instance, `-00` or `-01`. See "Command-Line Information" on page 1-267.

Why Use This Option

Higher precision leads to greater number of proven results but also requires more verification time. Each precision level corresponds to a different algorithm used for verification.

In most cases, you see the optimal balance between precision and verification time at level 2.

Settings

Default: 2

0

> This option corresponds to a static interval verification.

1

> This option corresponds to a more complex static interval verification.

2

> This option corresponds to a complex polyhedron model of domain values with additional precision for interprocedural analysis depending on the option `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

3

> This option is only suitable for code having less than 1000 lines. Using this option, the percentage of proven results can be very high.

Tips

- For best results in reasonable time, use the default level 2. If the verification takes a long time, reduce precision. However, the number of unproven checks can increase. Likewise, to reduce orange checks, you can improve your precision. But the verification can take significantly longer time.

- The precision levels 2 and below begin to take effect only from verification levels higher than `Software Safety Analysis level 0`. See also `Verification level (-to)`.

  For instance, to reduce analysis time, you might have reduced the verification level to `Software Safety Analysis level 0`. Do not try to reduce the precision level below 2 to lower the analysis time further.

  Note that algorithms used in precision level 3 can also apply to the verification level `Software Safety Analysis level 0`.

Command-Line Information

Parameter: -O0 | -O1 | -O2 | -O3
Default: -O2
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-O1`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-O1`

See Also

`Specific precision (-modules-precision)` | `Verification level (-to)`

Topics
"Specify Polyspace Analysis Options"
"Improve Verification Precision"

Verification level (`-to`)

Specify number of times the verification process runs on your code

Description

This option affects a Code Prover analysis only.

Specify the number of times the Polyspace verification process runs on your source code. Each run can lead to greater number of proven results but also requires more verification time.

Set Option

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

Command line and options file: Use the option `-to`. See "Command-Line Information" on page 1-270.

Why Use This Option

There are many reasons you might want to increase or decrease the verification level. For instance:

- Coding rules are checked early during the compilation phase, with some exception only. If you check for coding rules alone, you can lower the verification level. See "Check for Coding Standard Violations".

- If you see many orange checks after verification, try increasing the verification level. However, increasing the verification level also increases verification time.

  In most cases, you see the optimal balance between precision and verification time at level 2.

Settings

Default: Software Safety Analysis level 2

Source Compliance Checking

  Polyspace checks for compilation errors only. Most coding rule violations are also found in this phase.

Software Safety Analysis level 0

  The verification process performs some simple analysis. The analysis is designed to reach completion despite complexities in the code.

  If the verification gets stuck at a higher level, try running to this level and review the results.

Software Safety Analysis level 1

  The verification process analyzes each function once with algorithms whose complexity depends on the precision level. See `Precision level (-O)`. The analysis starts from the top of the function call hierarchy (an actual or generated `main` function) and propagates to the leaves of the call hierarchy.

Software Safety Analysis level 2

> The verification process analyzes each function twice. In the first pass, the analysis propagates from the top of the function call hierarchy to the leaves. In the second pass, the analysis propagates from the leaves back to the top. Each pass uses information gathered from the previous pass.
>
> Use this option for most accurate results in reasonable time.

Software Safety Analysis level 3

> The verification process runs three times on each function: from the top of the function call hierarchy to the leaves, from the leaves to the top, and from the top to the leaves again. Each pass uses information gathered from the previous pass.

Software Safety Analysis level 4

> The verification process runs four passes on each function: from the top of the function call hierarchy to the leaves twice. Each pass uses information gathered from the previous pass.

other

> If you use this option, Polyspace verification will make 20 passes unless you stop it manually.

Tips

- Use a higher verification level for fewer orange checks.

  In some cases, if the verification can detect that results of maximum precision are available after an earlier level, the verification stops and does not proceed to the level that you specify.

  Difference between Level 0 and 1

  The following example illustrates the difference between Software Safety Analysis level 0 and Software Safety Analysis level 1. In level 1, Code Prover can establish the success of the final assertion that involves a relation between two array elements even without knowing the actual elements of the array.

Software Safety Analysis Level 0	Software Safety Analysis Level 1
| ```
extern int tab[];

int main() {

    int i = tab[3];
    int j = tab[1];

    if (i > j) {
        int l = i-j;
        assert(l > 0);
    }
}
``` | ```
extern int tab[];

int main() {

    int i = tab[3];
    int j = tab[1];

    if (i > j) {
        int l = i-j;
        assert(l > 0);
    }
}
``` |

  In the table, verification produces an orange `Division by Zero` check during level 0 verification. The check turns green during level 1. The verification acquires more precise knowledge of x in the higher level.

  If a higher verification level fails because the verification runs out of memory, but results are available at a lower level, Polyspace displays the results from the lower level.

- For best results, use the option `Software Safety Analysis level 2`. If the verification takes too long, use a lower **Verification level**. Fix red errors and gray code before rerunning the verification with higher verification levels.

- Use the option `Other` sparingly since it can increase verification time by an unreasonable amount. Using `Software Safety Analysis level 2` provides optimal verification of your code in most cases.

- If the **Verification Level** is set to `Source Compliance Checking`, do not run verification on a remote server. The source compliance checking, or compilation, phase takes place on your local computer anyway. Therefore, if you are running verification only to the end of compilation, run verification on your local computer.

- If you want to see global variable sharing and usage only use `Show global variable sharing and usage only (-shared-variables-mode)` to run a less extensive analysis.

Command-Line Information

Parameter: `-to`
Value: `compile` | `pass0` | `pass1` | `pass2` | `pass3` | `pass4` | `other`
Default: `pass2`
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-to pass2`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-to pass2`

You can also use these additional values not available in the user interface:

- C projects: `c-to-il` (C to intermediate language conversion phase)
- C++ projects: `cpp-to-il` (C++ to intermediate language conversion phase), `cpp-normalize` (C++ compilation), `cpp-link` (C++ compilation)

Use these values only if you have specific reasons to do so. For instance, to generate a blank constraints (DRS) template for C++ projects, run an analysis up to the compilation by using `cpp-link` or `cpp-normalize`.

The values `cpp-link` and `cpp-normalize` will be removed in a future release. Use `compile` instead.

See Also

`Precision level (-O)` | `Show global variable sharing and usage only (-shared-variables-mode)`

Topics
"Specify Polyspace Analysis Options"
"Improve Verification Precision"

Verification time limit (`-timeout`)

Specify a time limit on your verification

Description

This option affects a Code Prover analysis only.

Specify a time limit for the verification in hours. If the verification does not complete within that limit, it stops.

Set Option

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

Command line and options file: Use the option `-timeout`. See "Command-Line Information" on page 1-271.

Why Use This Option

Use this option to impose a time limit on the verification.

By default, if an internal step in the verification lasts for more than 24 hours, the verification stops. You can use this option to reduce the time limit even further. Note that you can have verification results despite the verification timing out. For instance, if a step in Software Safety Analysis level 1 times out, you still get the results from level 0. See `Verification level (-to)`.

The option is useful only in very specific cases. Suppose your code has certain constructs that might slow down the verification. To check this, you can impose a time limit on the verification so that the verification stops if it takes too long.

Typically, Technical Support asks you to use this option as needed.

Settings

Enter the time in hours. For fractions of an hour, specify decimal form.

Command-Line Information
Parameter: `-timeout`
Value: *time*
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-timeout 5.75`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-timeout 5.75`

See Also

Topics
"Specify Polyspace Analysis Options"
"Improve Verification Precision"

Sensitivity context (`-context-sensitivity`)

Store call context information to identify function call that caused errors

Description

This option affects a Code Prover analysis only.

Specify the functions for which the verification must store call context information. If the function is called multiple times, using this option helps you to distinguish between the different calls.

Set Option

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

Command line and options file: Use the option `-context-sensitivity`. See "Command-Line Information" on page 1-273.

Why Use This Option

Suppose a function is called twice in your code. The check color on each operation in the function body is a combined result of both calls. If you want to distinguish between the colors in the two calls, use this option.

For instance, if a function contains a red or orange check and a green check on the same operation for two different calls, the software combines the contexts and displays an orange check on the operation. If you use this option, the check turns dark orange and the result details show the color of the check for each call.

For a tutorial on using this option, see "Identify Function Call with Run-Time Error".

Settings

Default: `none`

`none`

    The software does not store call context information for functions.

`auto`

    The software stores call context information for checks in:

    • Functions that form the leaves of the call tree. These functions are called by other functions, but do not call functions themselves.

- Small functions. The software uses an internal threshold to determine whether a function is small.

`custom`

The software stores call context information for functions that you specify. To enter the name of a function, click .

Tips

- If you select this option, you do not see tooltips in the body of the functions that benefit from this option (and keep the call contexts separate).

- If you select this option, the analysis can show some code operations in grey (unreachable code) even when you can identify execution paths leading to the operations. In this case, the grey code indicates operations that might be unreachable only in a particular call context.

  For instance, suppose this function is called with the arguments -1 and 1 :

  ```
  int isPositive (int num) {
      if(num < 0)
          return 0;
      return 1;
  }
  ```

  If you use the option with this function as argument, there are two unreachable code checks:

  - The check on `if` is grey because when the function is called with argument -1, the `if` condition is always true.
  - The check on the code inside the `if` branch is grey because when the function is called with argument 1, the `if` condition is always false.

  Each unreachable code check indicates code that is unreachable only in a particular call context. You see the call context in the result details.

Command-Line Information

Parameter: `-context-sensitivity`
Value: *function1*[,*function2*,...]
Default: none
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-context-sensitivity myFunc1,myFunc2`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-context-sensitivity myFunc1,myFunc2`

To allow the software to determine which functions receive call context storage, use the option `-context-sensitivity-auto`.

See Also

Topics
"Specify Polyspace Analysis Options"
"Identify Function Call with Run-Time Error"

Improve precision of interprocedural analysis (-path-sensitivity-delta)

Avoid certain verification approximations for code with fewer lines

Description

This option affects a Code Prover analysis only.

For smaller code, use this option to improve the precision of cross-functional analysis.

Set Option

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

Command line and options file: Use the option `-path-sensitivity-delta`. See "Command-Line Information" on page 1-274.

Why Use This Option

Use this option to avoid certain software approximations on execution paths. Avoiding these approximations results in fewer orange checks but a much longer verification time.

For instance, for deep function call hierarchies or nested conditional statements, to complete verification in a reasonable amount of time, the software combines many execution paths and stores less information at each stage of verification. If you use this option, the software stores more information about the execution paths, resulting in a more precise verification.

Settings

Default: Off

Enter a positive integer to turn on this option.

Entering a higher value leads to a greater number of proven results, but also increases verification time exponentially. For instance, a value of 10 can result in very long verification times.

Tips

Use this option only when you have less than 1000 lines of code.

Command-Line Information
Parameter: `-path-sensitivity-delta`
Value: Positive integer
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-path-sensitivity-delta 1`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-path-sensitivity-delta 1`

See Also

Topics
"Specify Polyspace Analysis Options"
"Improve Verification Precision"

Specific precision (`-modules-precision`)

Specify source files you want to verify at higher precision than the remaining verification

Description

This option affects a Code Prover analysis only.

Specify source files that you want to verify at a precision level higher than that for the entire verification.

Set Option

User interface (desktop products only): In your project configuration, the option is available on the **Precision** node. See "Dependency" on page 1-276 for other options you must also enable.

Command line and options file: Use the option `-modules-precision`. See "Command-Line Information" on page 1-276.

Why Use This Option

If a specific file is verified imprecisely leading to many orange checks in the file and elsewhere, you can improve the precision for that file.

Note that increasing precision also increases verification time.

Settings

Default: All files are verified with the precision you specified using **Precision > Precision level**.

Click ➕ to enter the name of a file without the extension `.c` and the corresponding precision level.

Dependency

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

Command-Line Information
Parameter: `-modules-precision`
Value: *file*:00 | *file*:01 | *file*:02 | *file*:03
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-O1 -modules-precision My_File:02`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-O1 -modules-precision My_File:02`

See Also
`Precision level (-O)`

Topics
"Specify Polyspace Analysis Options"

"Improve Verification Precision"

Inline (`-inline`)

Specify functions that must be cloned internally for each function call

Description

This option affects a Code Prover analysis only.

Specify the functions that the verification must clone internally for every function call.

Set Option

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

Command line and options file: Use the option `-inline`. See "Command-Line Information" on page 1-280.

Why Use This Option

Use this option sparingly. Sometimes, using the option helps to work around scaling issues during verification. If your verification takes too long, Technical Support can ask you to use this option for certain functions.

Do not use this option to understand results. For instance, suppose a function is called twice in your code. The check color on each operation in the function body is a combined result of both calls. If you want to distinguish between the colors in the two calls, use the option `Sensitivity context (-context-sensitivity)`.

Settings

No Default

Enter function names or choose from a list.

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

The verification internally clones the function for each call. For instance, if you specify the function `func` for inlining and `func` is called twice, the software internally creates two copies of `func` for verification.

However, for each run-time check in the function body, you see only one color in your verification results. The semantics of the check color is different from the normal specification.

Red checks:

- Normally, if a function is called twice and an operation causes a definite error only in one of the calls, the check color is orange.
- If you use this option, the color changes to dark orange (shown with an orange exclamation mark in the results list).

Gray checks:

- Normally, if a function is called twice and an `if` statement branch is unreachable in only one of the calls, the branch is shown as reachable.

- If you use this option, the worst color is shown for the check. Therefore, the `if` branch appears gray.

Below each check in an inlined function, you see information specific to each calling context. For instance, if a dark orange **Division by zero** occurs because a specific function call leads to a definite division by zero, you can identify the call along with values resulting from that call.

Do not use this option to understand results. Use this option only if a certain function causes scaling issues.

Tips

- Using this option can sometimes duplicate a lot of code and lead to scaling problems. Therefore choose functions to inline carefully.

- Choose functions to inline based on hints provided by the alias verification.

- Do not use this option for entry point functions, including `main`.

- Using this option can increase the number of gray **Unreachable code** checks.

  For example, in the following code, if you enter `max` for **Inline**, you obtain two **Unreachable code** checks, one for each call to `max`.

  ```
  int max(int a, int b) {
     return a > b ? a : b;
  }

  void main() {
    int i=3, j=1, k;
    k=max(i,j);
    i=0;
    k=max(i,j);
  }
  ```

- If you use the keyword `inline` before a function definition, place the definition in a header file and call the function from multiple source files, you have the same result as using the option **Inline**.
- For C++ code, this option applies to all overloaded methods of a class.

Command-Line Information

Parameter: `-inline`
Value: *function1*[,*function2*[,...]]
No Default
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-inline func1,func2`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-inline func1,func2`

See Also

Topics
"Specify Polyspace Analysis Options"

Depth of verification inside structures (-k-limiting)

Limit the depth of analysis for nested structures

Description

This option affects a Code Prover analysis only.

Specify a limit to the depth of analysis for nested structures.

Set Option

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

Command line and options file: Use the option `-k-limiting`. See "Command-Line Information" on page 1-281.

Why Use This Option

Use this option if the analysis is slow because your code has a structure that is many levels deep.

Typically, you see a warning message when a structure with a deep hierarchy is slowing down the verification.

Settings

Default: Full depth of nested structures is analyzed.

Enter a number to specify the depth of analysis for nested structures. For instance, if you specify 0, the analysis does not verify a structure inside a structure.

If you specify a number less than 2, the verification could be less precise.

Command-Line Information
Parameter: `-k-limiting`
Value: *positive integer*
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-k-limiting 3`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-k-limiting 3`

See Also

Topics
"Specify Polyspace Analysis Options"

Generate report

Specify whether to generate a report after the analysis

Description

Specify whether to generate a report along with analysis results.

Depending on the format you specify, you can view this report using an external software. For example, if you specify the format PDF, you can view the report in a `pdf` reader.

Set Option

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

Command line and options file: See "Command-Line Information" on page 1-283.

Why Use This Option

You can generate a report from your analysis results for archiving purposes. You can provide this report to your management or clients as proof of code quality.

Using other analysis options, you can tailor the report content and format for your specific needs. See `Bug Finder and Code Prover report (-report-template)` and `Output format (-report-output-format)`.

Settings

☑ On

Polyspace generates an analysis report using the template and format you specify.

The report is stored in the `Polyspace-Doc` subfolder of your results folder.

In Polyspace desktop products, to open your results folder from the user interface, on the **Project Browser** pane, right-click the results node and select **Open Folder with File Manager**.

To change the results folder location, see "Project and Results Folder Contents".

On the command-line, the results folder is the argument of the option `-results-dir`.

☐ Off (default)

Polyspace does not generate an analysis report. You can still view your results in the Polyspace interface.

Tips

This option allows you to specify report generation before starting an analysis.

To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting** > **Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.

Command-Line Information

There is no command-line option to solely turn on the report generator. However, using the options `-report-template` for template and `-report-output-format` for output format automatically turns on the report generator.

See Also

Bug Finder and Code Prover report (`-report-template`) | Output format (`-report-output-format`) | `polyspace-report-generator`

Topics
"Specify Polyspace Analysis Options"
"Generate Reports"

Bug Finder and Code Prover report (`-report-template`)

Specify template for generating analysis report

Description

Specify template for generating analysis report.

`.rpt` files for the report templates are available in *polyspaceroot*`\toolbox\polyspace\psrptgen\templates\`. Here, *polyspaceroot* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Reporting** node. You have separate options for Bug Finder and Code Prover analysis. See "Dependencies" on page 1-289 for other options you must also enable.

Command line and options file: Use the option `-report-template`. See "Command-Line Information" on page 1-290.

Why Use This Option

Depending on the template that you use, the report contains information about certain types of results from the **Results List** pane. The template also determines what information is presented in the report and how the information is organized. See the template descriptions below.

Settings – Bug Finder

Default: `BugFinderSummary`

`BugFinder`

The report lists:

- **Polyspace Bug Finder Summary**: Number of results in the project. The results are summarized by file. The files that are partially analyzed because of compilation errors are listed in a separate table.
- **Code Metrics**: Summary of the various code complexity metrics. For more information, see "Code Metrics".
- **Coding Rules**: Coding rule violations in the source code. For each rule violation, the report lists the:

  - Rule number and description.
  - Function containing the rule violation.
  - Review information, such as **Severity**, **Status** and comments.
- **Defects**: Defects found in the source code. For each defect, the report lists the:

- Function containing the defect.
- Defect information on the **Result Details** pane.
- Review information, such as **Severity**, **Status** and comments.

- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. If your project has source files with compilation errors, these files are also listed.

  If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

BugFinderSummary

The report lists:

- **Polyspace Bug Finder Summary**: Number of results in the project. The results are summarized by file. The files that are partially analyzed because of compilation errors are listed in a separate table.
- **Code Metrics**: Summary of the various code complexity metrics. For more information, see "Code Metrics".
- **Coding Rules Summary**: Coding rules along with number of violations.
- **Defect Summary**: Defects that Polyspace Bug Finder looks for. For each defect, the report lists the:

  - Defect group.
  - Defect name.
  - Number of instances of the defect found in the source code.

- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. For more information, see "Analysis Options in Polyspace Bug Finder". If your project has source files with compilation errors, these files are also listed.

  If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

CodeMetrics

The report lists the following:

- **Code Metrics Summary**: Various quantities related to the source code. For more information, see "Code Metrics".
- **Code Metrics Details**: Various quantities related to the source code with the information broken down by file and function.
- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. If you configured your project for multitasking, this section also lists the **Concurrency Modeling Summary**. If your project has source files with compilation errors, these files are also listed.

  If you check for coding rules, an additional **Coding Rules Configuration** section states the rules along with the information whether they were enabled or disabled.

`CodingStandards`

The report contains separate chapters for each coding standard enabled in the analysis (for instance, MISRA C: 2012, CERT C, custom rules, and so on). Each chapter contains the following information:

- **Summary - Violations by File**: Graph showing each file with number of rule violations.
- **Summary - Violations by Rule**: Graph showing each rule with number of violations. If a rule is not enabled or not violated, it does not appear in the graph.
- **Summary for all Files**: Table showing each file with number of rule violations.
- **Summary for Enabled Guidelines** or **Summary for Enabled Rules**: Table showing each guideline or rule with number of violations.
- **Violations**: Tables listing each rule violation, along with information such as ID, function name, severity, status, and so on. One table is created per file.

An appendix lists the options used in the Polyspace analysis.

`SecurityCWE`

The report contains the same information as the `BugFinder` report. However, in the **Defects** chapter, an additional column lists the CWE™ rules mapped to each defect. The **Configuration Settings** appendix also includes a **Security Standard to Polyspace Result Map**.

`Metrics`

Only available for results downloaded from the Polyspace Metrics interface.

The report lists information useful to quality engineers and available on the Polyspace Metrics interface, including:

- Information about whether the project satisfies quality objectives
- Time taken in each phase of analysis
- Metrics about the whole project. For each metric, the report lists the quality threshold and whether the metric satisfies this threshold.
- Coding rule violations in the project. For each rule, the report lists the number of violations justified and whether the justifications satisfy quality objectives.
- Definite as well as possible run-time errors in the project. For each type of run-time error, the report lists the number of errors justified and whether the justifications satisfy quality objectives.

The appendices contain further details of Polyspace configuration settings, code metrics, coding rule violations, and run-time errors.

Settings – Code Prover

Default: `Developer`

`CodeMetrics`

The report contains a summary of code metrics, followed by the complete metrics for an application.

`CodingStandards`

The report contains separate chapters for each coding standard enabled in the analysis (for instance, MISRA C: 2012, custom rules, and so on). Each chapter contains the following information:

- **Summary - Violations by File**: Graph showing each file with number of rule violations.
- **Summary - Violations by Rule**: Graph showing each rule with number of violations. If a rule is not enabled or not violated, it does not appear in the graph.
- **Summary for all Files**: Table showing each file with number of rule violations.
- **Summary for Enabled Guidelines** or **Summary for Enabled Rules**: Table showing each guideline or rule with number of violations.
- **Violations**: Tables listing each rule violation, along with information such as ID, function name, severity, status, and so on. One table is created per file.

An appendix lists the options used in the Polyspace analysis.

`Developer`

The report lists information useful to developers, including:

- Summary of results
- Coding rule violations
- List of proven run-time errors or red checks
- List of unproven run-time errors or orange checks
- List of unreachable procedures or gray checks
- Global variable usage in code. See "Global Variables".

The report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. If your project has source files with compilation errors, these files are also listed.

`DeveloperReview`

The report lists the same information as the `Developer` report. However, the reviewed results are sorted by severity and status, and unreviewed results are sorted by file location.

`Developer_withGreenChecks`

The report lists the same information as the `Developer` report. In addition, the report lists code proven to be error-free or green checks.

`Quality`

The report lists information useful to quality engineers, including:

- Summary of results
- Statistics about the code
- Graphs showing distributions of checks per file

The report also contains the Polyspace configuration settings and modifiable assumptions used in the analysis. If your project has source files with compilation errors, these files are also listed.

`VariableAccess`

The report displays the global variable access in your source code. The report first displays the number of global variables of each type. For information on the types, see "Global Variables". For each global variable, the report displays the following information:

- Variable name.

  The entry for each variable is denoted by |.
- Type of the variable.
- Number of read and write operations on the variable.
- Details of read and write operations. For each read or write operation, the table displays the following information:

  - File and function containing the operation in the form *file_name.function_name*.

    The entry for each read or write operation is denoted by ||. Write operations are denoted by < and read operations by >.
  - Line and column number of the operation.

This report captures the information available on the **Variable Access** pane in the Polyspace user interface.

CallHierarchy

The report displays the call hierarchy in your source code. For each function call in your source code, the report displays the following information:

- Level of call hierarchy, where the function is called.

  Each level is denoted by |. If a function call appears in the table as ||| -> *file_name.function_name*, the function call occurs at the third level of the hierarchy. Beginning from main or an entry point, there are three function calls leading to the current call.
- File containing the function call.

  In addition, the line and column is also displayed.
- File containing the function definition.

  In addition, the line and column where the function definition begins is also displayed.

In addition, the report also displays uncalled functions.

This report captures the information available on the **Call Hierarchy** pane in the Polyspace user interface.

SoftwareQualityObjectives

The report lists information useful to quality engineers and available on the Polyspace Metrics interface, including:

- Information about whether the project satisfies quality objectives
- Time taken in each phase of verification
- Metrics about the whole project. For each metric, the report lists the quality threshold and whether the metric satisfies this threshold.
- Coding rule violations in the project. For each rule, the report lists the number of violations justified and whether the justifications satisfy quality objectives.
- Definite as well as possible run-time errors in the project. For each type of run-time error, the report lists the number of errors justified and whether the justifications satisfy quality objectives.

The appendices contain further details of Polyspace configuration settings, code metrics, coding rule violations, and run-time errors.

This template is available only if you generate a report from results uploaded to the Polyspace Access web interface or from results uploaded to the Polyspace Metrics web interface (and then downloaded to the Polyspace user interface) . In each case, you have to set the objectives explicitly in the web interface and then generate the reports.

For more information on the predefined Software Quality Objectives, see "Software Quality Objectives".

SoftwareQualityObjectives_Summary

The report contains the same information as the `SoftwareQualityObjectives` report. However, it does not have the supporting appendices with details of code metrics, coding rule violations and run-time errors.

This template is available only if you generate a report from results uploaded to the Polyspace Access web interface or from results uploaded to the Polyspace Metrics web interface (and then downloaded to the Polyspace user interface). In each case, you have to set a quality objective level explicitly in the web interface and then generate the reports.

For more information on the predefined Software Quality Objectives, see "Software Quality Objectives".

Dependencies

In the user interface of the Polyspace desktop products, this option is enabled only if you select the `Generate report` option.

Tips

- This option allows you to specify report generation before starting an analysis.

  To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting > Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

  After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.

- In Bug Finder, the report does not contain the line or column number for a result. Use the report for archiving, gathering statistics and checking whether results have been reviewed and addressed (for certification purposes or otherwise). To review a result in your source code, use the Polyspace user interface or your IDE if you are using a Polyspace plugin.

- If you use the `SoftwareQualityObjectives_Summary` and `SoftwareQualityObjectives` templates to generate reports, the pass/fail status depends on whether you set the quality objectives level in Polyspace Metrics or Polyspace Access:

  - In Polyspace Access, the pass/fail status is determined based on all results. For instance, if you use the level SQO-4 which sets a threshold of 60% on orange overflow checks, your project has a **FAIL** status if the percentage of green and justified orange overflow checks is less than 60% of *all green and orange overflow checks*.

  - In Polyspace Metrics, the pass/fail status is determined based on a file-by-file basis. The overall status is **FAIL** if one of the files have a **FAIL** status. For instance, if you use the level SQO-4

which sets a threshold of 60% on orange overflow checks, your project has a **FAIL** status if the percentage of green and justified orange overflow checks *in any file* is less than 60% of green and orange overflow checks in that file.

- The first chapter of the reports contain a summary of the relevant results. You can enter a Pass/ Fail status in that chapter for your project based on the summary. If you use the template SoftwareQualityObjectives or SoftwareQualityObjectives_Summary, the status is automatically assigned based on your objectives and the verification results. For more information on enforcing objectives using Polyspace Metrics, see "Compare Metrics Against Software Quality Objectives".

Command-Line Information

Parameter: -report-template
Value: Full path to *template*.rpt
Example (Bug Finder): polyspace-bug-finder -sources *file_name* -report-template *polyspaceroot*\toolbox\polyspace\psrptgen\templates\bug_finder\BugFinder.rpt
Example (Code Prover): polyspace-code-prover -sources *file_name* -report-template *polyspaceroot*\toolbox\polyspace\psrptgen\templates\Developer.rpt
Example (Bug Finder Server): polyspace-bug-finder-server -sources *file_name* -report-template *polyspaceroot*\toolbox\polyspace\psrptgen\templates\bug_finder\BugFinder.rpt
Example (Code Prover Server): polyspace-code-prover-server -sources *file_name* -report-template *polyspaceroot*\toolbox\polyspace\psrptgen\templates\Developer.rpt

See Also

Generate report|Output format (-report-output-format)|polyspace-report-generator

Topics
"Specify Polyspace Analysis Options"
"Generate Reports"

Output format (`-report-output-format`)

Specify output format of generated report

Description

Specify output format of analysis report.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Reporting** node. See "Dependencies" on page 1-291 for other options you must also enable.

Command line and options file: Use the option `-report-output-format`. See "Command-Line Information" on page 1-292.

Why Use This Option

Use this option to specify whether you want a report in PDF, HTML or another format.

Settings

Default: `Word`

`HTML`
> Generate report in `.html` format

`PDF`
> Generate report in `.pdf` format

`Word`
> Generate report in `.docx` format.

Tips

- This option allows you to specify report generation before starting an analysis.

  To generate a report *after* an analysis is complete, in the user interface of the Polyspace desktop products, select **Reporting > Run Report**. Alternatively, at the command line, use the `polyspace-report-generator` command.

  After analysis, you can also export the result as a text file for further customization. Use the option `-generate-results-list-file` with the `polyspace-report-generator` command.

- If the table of contents or graphics in a `.docx` report appear outdated, select the content of the report and refresh the document. Use keyboard shortcuts **Ctrl+A** to select the content and **F9** to refresh it.

Dependencies

In the user interface of the Polyspace desktop products, this option is enabled only if you select the `Generate report` option.

Command-Line Information
Parameter: `-report-output-format`
Value: `html | pdf | word`
Default: `word`
Example (Bug Finder): `polyspace-bug-finder -sources` *`file_name`* `-report-output-format pdf`
Example (Code Prover): `polyspace-code-prover -sources` *`file_name`* `-report-output-format pdf`
Example (Bug Finder Server): `polyspace-bug-finder-server -sources` *`file_name`* `-report-output-format pdf`
Example (Code Prover Server): `polyspace-code-prover-server -sources` *`file_name`* `-report-output-format pdf`

See Also
`Bug Finder and Code Prover report (-report-template)|Generate report| polyspace-report-generator`

Topics
"Specify Polyspace Analysis Options"
"Generate Reports"

Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`)

Enable batch remote analysis

Description

Specify that the analysis must be offloaded to a remote server.

To offload a Polyspace analysis, you need these products:

- A Polyspace product on the client side to submit an analysis job. Typically, you use a desktop product such as Polyspace Bug Finder to submit jobs, but you can also use a server product such as Polyspace Bug Finder Server™ to offload an analysis from one server to another.
- A Polyspace server product (Polyspace Bug Finder Server or Polyspace Code Prover Server) on the server side to run the analysis.
- MATLAB Parallel Server™ to hold jobs from multiple clients in queue and allocate the jobs as Polyspace Server instances become available.

For details, see "Install Products for Submitting Polyspace Analysis from Desktops to Remote Server".

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Run Settings** node. You have separate options for a Bug Finder and a Code Prover analysis.

Command line and options file: Use the option `-batch`. See "Command-Line Information" on page 1-294.

Why Use This Option

Use this option if you want the analysis to run on a remote cluster instead of your local desktop.

For instance, you can run remote analysis when:

- You want to shut down your local machine but not interrupt the analysis.
- You want to free execution time on your local machine.
- You want to transfer the analysis to a more powerful computer.

Settings

☑ On

Run batch analysis on a remote computer. In this remote analysis mode, the analysis is queued on a cluster after the compilation phase. Therefore, on your local computer, after the analysis is queued:

- If you are running the analysis from the Polyspace user interface, you can close the user interface.

- If you are running the analysis from the command line, you can close the command-line window.

You can manage the queue from the Polyspace Job Monitor. To use the Polyspace Job Monitor:

- In the Polyspace user interface, select **Tools > Open Job Monitor**. See "Send Polyspace Analysis from Desktop to Remote Servers".
- On the DOS or UNIX® command line, use the `polyspace-jobs-manager` command. For more information, see "Send Polyspace Analysis from Desktop to Remote Servers Using Scripts".
- On the MATLAB command line, use the `polyspaceJobsManager` function.

After the analysis, you might have to manually download the results from the cluster.

If you use a Polyspace Server product to offload an analysis from one server to another, the results are automatically downloaded after analysis.

☐ Off (default)

Do not run batch analysis on a remote computer.

Dependencies

- If you use a third-party scheduler instead of the MATLAB Job Scheduler, add the option `-no-credentials-check`. The credentials check performed in the product is only compatible with the MATLAB Job Scheduler. In the Polyspace user interface, add this option to the `Other` field.
- Do not run a Code Prover analysis on a remote cluster if you run up to the **Verification Level** of `Source Compliance Checking`. For both local and remote analysis, the source compliance checking or compilation phase takes place on your local computer. Therefore, if you are running only up to this phase, run on your local computer.
- If you use a Polyspace Server product to offload an analysis from one server to another, the offloading uses the MATLAB Job Scheduler that comes by default with MATLAB Parallel Server. You cannot use a third-party scheduler.

Command-Line Information

To run a remote analysis from the command line, use with the `-scheduler` option.
Parameter: `-batch`
Value: `-scheduler` *host_name* if you have not set the **Job scheduler host name** in the Polyspace user interface
Default: Off
Example (Bug Finder): `polyspace-bug-finder -batch -scheduler NodeHost` or `polyspace-bug-finder -batch -scheduler MJSName@NodeHost`
Example (Code Prover): `polyspace-code-prover -batch -scheduler NodeHost` or `polyspace-code-prover -batch -scheduler MJSName@NodeHost`
Example (Bug Finder Server): `polyspace-bug-finder-server -batch -scheduler NodeHost`
Example (Code Prover Server): `polyspace-code-prover-server -batch -scheduler NodeHost`

See Also

`-scheduler`

Topics

"Install Products for Submitting Polyspace Analysis from Desktops to Remote Server"

"Specify Polyspace Analysis Options"

"Send Polyspace Analysis from Desktop to Remote Servers"

"Send Polyspace Analysis from Desktop to Remote Servers Using Scripts"

Upload results to Polyspace Metrics (`-add-to-results-repository`)

Upload analysis results for viewing on Polyspace Metrics web dashboard

Description

This option applies to the Polyspace desktop products only.

Specify upload of analysis results to the Polyspace Metrics results repository, allowing Web-based reporting of results and code metrics.

Set Option

User interface: In your project configuration, the option is on the **Run Settings** node. You have separate options for a Bug Finder and a Code Prover analysis. See "Dependencies" on page 1-296 for other options that you must also enable.

Command line and options file: Use the option `-add-to-results-repository`. See "Command-Line Information" on page 1-297.

Why Use This Option

Polyspace Metrics is a web dashboard that generates code quality metrics from your analysis results. Using this dashboard, you can:

- Provide your management a high-level overview of your code quality.
- Compare your code quality against predefined standards.
- Establish a process where you review in detail only those results that fail to meet standards.
- Track improvements or regression in code quality over time.

See "Generate Code Quality Metrics".

Settings

☑ On

> Analysis results are stored in the Polyspace Metrics results repository. This allows you to use a Web browser to view results and code metrics.
>
> The results are not downloaded automatically to your desktop.

☐ Off (default)

> Analysis results are stored locally.

Dependencies

The option to upload to Polyspace Metrics is available only if you select `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

If you perform a local analysis on your desktop, you can later upload your results to Polyspace Metrics. Right-click your results file and select **Upload to Metrics**.

Command-Line Information
Parameter: `-add-to-results-repository`
Default: Off
Example (Bug Finder): `polyspace-bug-finder -batch -scheduler NodeHost -add-to-results-repository -password` *passwordName*
Example (Code Prover): `polyspace-code-prover -batch -scheduler NodeHost -add-to-results-repository -password` *passwordName*

The password is optional.

The upload uses the Polyspace Metrics server that you set up in the Polyspace user interface. See "Set Up Polyspace Metrics". If you want to explicitly specify the Polyspace Metrics server during upload, use the option `-polyspace-metrics-server` *serverName:portNumber*. For instance:

`-add-to-results-repository -polyspace-metrics-server localhost:12427`

See Also
`Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`

Topics
"Set Up Polyspace Metrics"
"Generate Code Quality Metrics"

Command/script to apply after the end of the code verification (`-post-analysis-command`)

Specify command or script to be executed after analysis

Description

Specify a command or script to be executed after the analysis.

Set Option

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

Command line and options file: Use the option `-post-analysis-command`. See "Command-Line Information" on page 1-299.

Why Use This Option

Create scripts for tasks that you want performed after the Polyspace analysis.

For instance, you want to be notified by email that the Polyspace analysis is over. Create a script that sends an email and use this option to execute the script after the Polyspace analysis.

Settings

No Default

Enter full path to the command or script, or click to navigate to the location of the command or script. After the analysis, this script is executed.

The script is executed in the Polyspace results folder. In your script, consider the results folder as the current folder for relative paths to other files.

For a Perl script, in Windows, specify the full path to the Perl executable followed by the full path to the script. For example, to specify a Perl script `send_email.pl` that sends an email once the analysis is over, enter *polyspaceroot*\sys\perl\win32\bin\perl.exe *<absolute_path>*\send_email.pl. Here, *polyspaceroot* is the location of the current Polyspace installation, such as C:\Program Files\Polyspace\R2019a\, and *<absolute_path>* is the location of the Perl script.

Tips

Running post analysis commands on the server

If you perform verification on a remote server, after verification, the software executes your command on the server, not on the client desktop. If your command executes a script, the script must be present on the server.

For instance, if you specify the command, `/local/utils/send_mail.sh`, the Shell script `send_email.sh` must be present on the server in `/local/utils/`. The software does not copy the

script `send_email.sh` from your desktop to the server before executing the command. If the script is not present on the server, you encounter an error. Sometimes, there are multiple servers that the MATLAB Job Scheduler can run the verification on. Place the script on each of the servers because you do not control which server eventually runs your verification.

Running post analysis commands in the Polyspace user interface

To test the use of this option, run the following Perl script from a folder containing a Polyspace project (`.psprj` file). The script parses the latest Polyspace log file in the folder `Module_1\CP_Result` and writes the current project name and date to a file `report.txt`. The file is saved in `Module_1\CP_Result`.

```
foreach my $file (`ls Module_1\\CP_Result\\Polyspace_*.log`) {
  open (FH, $file);

while ($line = <FH>) {
    if ($line =~ m/Ending at: (.*)/) {
      $date=$1;
    }
    if ($line =~ m/-prog=(.*)/) {
      $project=$1;
    }
  }
}

my $filename = 'report.txt';
open(my $fh, '>', $filename) or die "Could not open file '$filename' $!";

print $fh "date=$date\n";
print $fh "project=$project\n";

close $fh;
```

In Linux, you can specify the Perl script for this option.

In Windows, instead of specifying the Perl script directly, specify a `.bat` file that invokes Perl and runs this script. For instance, the `.bat` file can contain the following line (assuming that the `.bat` file and `.pl` file are in the Polyspace project folder). Depending on your MATLAB installation, change the path to `perl.exe` appropriately.

```
"C:\Program Files\MATLAB\R2018b\sys\perl\win32\bin\perl.exe" command.pl
```

Run Code Prover. Check that the folder `Module_1\CP_Result` contains the file `report.txt` with the project name and date.

Command-Line Information

Parameter: `-post-analysis-command`
Value: Path to executable file or command in quotes
No Default
Example in Linux (Bug Finder): `polyspace-bug-finder -sources` *file_name* `-post-analysis-command` `` `pwd` ``/send_email.pl
Example in Linux (Code Prover): `polyspace-code-prover -sources` *file_name* `-post-analysis-command` `` `pwd` ``/send_email.pl

Example in Linux (Bug Finder Server): `polyspace-bug-finder-server -sources` *file_name* `-post-analysis-command ` `` `pwd` ``/send_email.pl`
Example in Linux (Code Prover Server): `polyspace-code-prover-server -sources` *file_name* `-post-analysis-command ` `` `pwd` ``/send_email.pl`
Example in Windows: `polyspace-bug-finder -sources` *file_name* `-post-analysis-command "C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"`

Note that in Windows, you use the full path to the Perl executable.

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

Topics
"Specify Polyspace Analysis Options"

Automatic Orange Tester (-automatic-orange-tester)

(To be removed) Specify that Automatic Orange Tester must be executed after verification

---

Note The Automatic Orange Tester will be removed in a future release. See "Compatibility Considerations".

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify that the Automatic Orange Tester must be executed at the end of the verification.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See "Dependency" on page 1-301 for other options you must also enable.

Command line and options file: Use the option `-automatic-orange-tester`. See "Command-Line Information" on page 1-302.

Why Use This Option

The Automatic Orange Tester runs dynamic tests on your code. The dynamic tests help you determine if an orange check represents a real run-time error or an imprecision of Polyspace analysis. For a tutorial, see "Test Orange Checks for Run-Time Errors".

To run the Automatic Orange Tester after verification, you must select this option *before verification*. During verification, Polyspace generates additional source code to test each orange check for errors. When you run the Automatic Orange Tester later, the software uses this instrumented code for testing.

Settings

☑ On

    After verification, when you run the Automatic Orange Tester, Polyspace creates tests for unproven code and runs them.

☐ Off (default)

    You cannot launch the Automatic Orange Tester after verification.

Dependency

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

Tips

- To launch the Automatic Orange Tester, after verification, open your results. Select **Tools > Automatic Orange Tester**.

- When using the automatic orange tester, you cannot:

  - Select **Division round down** under **Target & Compiler**.

  - Select the options `c18`, `tms320c3c`. `x86_64` or `sharc21x61` for **Target & Compiler > Target processor type**.

  - Specify the type `char` as 16-bit or `short` as 8-bit using the option `mcpu...(Advanced)` for **Target & Compiler > Target processor type**. For the same option, you must specify the type `pointer` as 32-bit.

  - Specify global asserts in the code, having the form `Pst_Global_Assert(A,B)`. In global assert mode, you cannot use **Constraint setup** under **Inputs & Stubbing**.

  - Select these options related to floating-point verification: **Subnormal detection mode** and **Consider non finite floats**.

Command-Line Information

Parameter: `-automatic-orange-tester`
Default: Off
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-lang c - automatic-orange-tester`

Compatibility Considerations

Automatic Orange Tester will be removed
Not recommended starting in R2020b

The Automatic Orange Tester will be removed in a future release. In the desktop interface, you set this option in the **Configuration** pane under the **Advanced Settings** node.

If you use these command-line options in your scripts, remove them:

- `-automatic-orange-tester`

- `-automatic-orange-tester-loop-max-iteration`

- `-automatic-orange-tester-tests-number`

- `-automatic-orange-tester-timeout`

See Also

`Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)|Maximum test time (-automatic-orange-tester-timeout)|Number of automatic tests (-automatic-orange-tester-tests-number)`

Topics
"Specify Polyspace Analysis Options"
"Test Orange Checks for Run-Time Errors"
"Limitations of Automatic Orange Tester"

Number of automatic tests (-automatic-orange-tester-tests-number)

(To be removed) Specify number of tests that Automatic Orange Tester must run

---

Note The Automatic Orange Tester will be removed in a future release. See "Compatibility Considerations".

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify number of tests that you want the Automatic Orange Tester to run. The more the number of tests, the greater the possibility of finding a run-time error, but longer it takes to complete.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See "Dependencies" on page 1-303 for other options you must also enable.

Command line and options file: Use the option `-automatic-orange-tester-tests-number`. See "Command-Line Information" on page 1-303.

Settings

Default: 500

Enter number of tests up to a maximum of 100,000.

Dependencies

This option is enabled only if you set the following options:

- Set `Source code language (-lang)` to `C` or `C-CPP`.
- Specify the option `Automatic Orange Tester (-automatic-orange-tester)`.

Command-Line Information
Parameter: `-automatic-orange-tester-tests-number`
Value: *positive integer*
Default: 500
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-lang c -automatic-orange-tester -automatic-orange-tester-tests-number 500`

Compatibility Considerations

Automatic Orange Tester will be removed
Not recommended starting in R2020b

The Automatic Orange Tester will be removed in a future release. In the desktop interface, you set this option in the **Configuration** pane under the **Advanced Settings** node.

If you use these command-line options in your scripts, remove them:

- `-automatic-orange-tester`
- `-automatic-orange-tester-loop-max-iteration`
- `-automatic-orange-tester-tests-number`
- `-automatic-orange-tester-timeout`

See Also

`Automatic Orange Tester (-automatic-orange-tester)`

Topics
"Specify Polyspace Analysis Options"
"Test Orange Checks for Run-Time Errors"

Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)

(To be removed) Specify number of loop iterations after which Automatic Orange Tester considers infinite loop

---

Note The Automatic Orange Tester will be removed in a future release. See "Compatibility Considerations".

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify number of loop iterations after which the Automatic Orange Tester considers the loop to be infinite. Specifying a large number decreases the possibility of identifying an infinite loop incorrectly, but takes more time to complete.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See "Dependencies" on page 1-305 for other options you must also enable.

Command line and options file: Use the option `-automatic-orange-tester-loop-max-iteration`. See "Command-Line Information" on page 1-305.

Settings

Default: 1000

Enter number of loop iterations. The maximum value that the software supports is 1000.

Dependencies

This option is enabled only if you set the following options:

- Set `Source code language (-lang)` to C or C-CPP.
- Specify the option `Automatic Orange Tester (-automatic-orange-tester)`.

Command-Line Information
Parameter: `-automatic-orange-tester-loop-max-iteration`
Value: *positive integer*
Default: 1000
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-lang c -automatic-orange-tester -automatic-orange-tester-loop-max-iteration 500`

Compatibility Considerations

Automatic Orange Tester will be removed
Not recommended starting in R2020b

The Automatic Orange Tester will be removed in a future release. In the desktop interface, you set this option in the **Configuration** pane under the **Advanced Settings** node.

If you use these command-line options in your scripts, remove them:

- `-automatic-orange-tester`
- `-automatic-orange-tester-loop-max-iteration`
- `-automatic-orange-tester-tests-number`
- `-automatic-orange-tester-timeout`

See Also

Automatic Orange Tester (`-automatic-orange-tester`)

Topics
"Specify Polyspace Analysis Options"
"Test Orange Checks for Run-Time Errors"

Maximum test time (-automatic-orange-tester-timeout)

(To be removed) Specify time in seconds allowed for a single test in Automatic Orange Tester

---

Note The Automatic Orange Tester will be removed in a future release. See "Compatibility Considerations".

Description

This option affects a Code Prover analysis only. Use this option only if you review the Code Prover results in the Polyspace desktop products.

Specify time in seconds allowed for a single test. After this time is over, the Automatic Orange Tester proceeds to the next test. Increasing this time reduces number of tests that do not complete, but increases total verification time.

Set Option

User interface (desktop products only): In your project configuration, the option is on the **Advanced Settings** node. See "Dependencies" on page 1-307 for other options you must also enable.

Command line and options file: Use the option `-automatic-orange-tester-timeout`. See "Command-Line Information" on page 1-307.

Settings

Default: 5

Enter time in seconds. The maximum value that the software supports is 60.

Dependencies

This option is enabled only if you set the following options:

- Set `Source code language (-lang)` to C or C-CPP.
- Specify the option `Automatic Orange Tester (-automatic-orange-tester)`.

Command-Line Information
Parameter: `-automatic-orange-tester-timeout`
Value: *time*
Default: 5
Example (Code Prover): `polyspace-code-prover -sources` *file_name* `-lang c -automatic-orange-tester -automatic-orange-tester-test-timeout 10`

1-307

Compatibility Considerations

Automatic Orange Tester will be removed
Not recommended starting in R2020b

The Automatic Orange Tester will be removed in a future release. In the desktop interface, you set this option in the **Configuration** pane under the **Advanced Settings** node.

If you use these command-line options in your scripts, remove them:

- `-automatic-orange-tester`
- `-automatic-orange-tester-loop-max-iteration`
- `-automatic-orange-tester-tests-number`
- `-automatic-orange-tester-timeout`

See Also
Automatic Orange Tester (`-automatic-orange-tester`)

Topics
"Specify Polyspace Analysis Options"
"Test Orange Checks for Run-Time Errors"

Other

Specify additional flags for analysis

Description

This option is useful only if you run an analysis in the user interface of the Polyspace desktop products.

Enter command-line-style flags such as `-max-processes`.

Set Option

In your project configuration, the option is on the **Advanced Settings** node. You can enter multiple options in this field. If you enter the same option multiple times with different arguments, the analysis uses your last argument.

Why Use This Option

Use this option to add nonofficial or command-line only options to the analyzer.

If you have to add several command line options, you can save them in a text file and specify the file using the option `-options-file`. You can reuse the options file across projects.

Tip

Nonofficial options: In rare circumstances, to work around very specific issues, MathWorks Technical Support might provide you some undocumented options. If you are running verification from the user interface, you use the **Other** field in the **Configuration** pane to enter the options. Sometimes, the options and their arguments have to be preceded by extra flags. When providing you the option, Technical Support will let you know if the extra flags are required.
Possible Flags: `-extra-flags` | `-c-extra-flags` | `-cpp-extra-flags` | `-cfe-extra-flags` | `-il-extra-flags`
Example (Bug Finder): `polyspace-bug-finder -extra-flags` *-option-name* `-extra-flags` *option_param*
Example (Code Prover): `polyspace-code-prover -extra-flags` *-option-name* `-extra-flags` *option_param*
Example (Bug Finder Server): `polyspace-bug-finder-server -extra-flags` *-option-name* `-extra-flags` *option_param*
Example (Code Prover Server): `polyspace-code-prover-server -extra-flags` *-option-name* `-extra-flags` *option_param*

Analysis Options, Command-Line Only

-asm-begin -asm-end

Exclude compiler-specific `asm` functions from analysis

Syntax

```
-asm-begin "mark1[,mark2,...]" -asm-end "mark1[,mark2,...]"
```

Description

`-asm-begin "mark1[,mark2,...]" -asm-end "mark1[,mark2,...]"` excludes compiler-specific assembly language source code functions from the analysis. You must use these two options together.

Polyspace recognizes most inline assemblers by default. Use the option only if compilation errors occur due to introduction of assembly code. For more information, see "Assumptions About Assembly Code" on page 4-33.

Mark the offending code block by two `#pragma` directives, one at the beginning of the assembly code and one at the end. In the command usage, give these marks in the same order for `-asm-begin` as they are for `-asm-end`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

A block of code is delimited by `#pragma start1` and `#pragma end1`. These names must be in the same order for their respective options. Either:

```
-asm-begin "start1" -asm-end "end1"
```

or

```
-asm-begin "mark1,...markN,start1" -asm-end "mark1,...markN,end1"
```

The following example marks two functions for exclusion, `foo_1` and `foo_2`.

Code:

```
#pragma asm_begin_foo
int foo(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_foo

#pragma asm_begin_bar
void bar(void) { /* asm code to be ignored by Polyspace */ }
#pragma asm_end_bar
```

Polyspace Command:

- Bug Finder:

```
polyspace-bug-finder -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
        -asm-end "asm_end_foo,asm_end_bar"
```

- Code Prover:

```
polyspace-code-prover -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
        -asm-end "asm_end_foo,asm_end_bar"
```

- Bug Finder Server:

```
polyspace-bug-finder-server -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
        -asm-end "asm_end_foo,asm_end_bar"
```

- Code Prover Server:

```
polyspace-code-prover-server -lang c -asm-begin "asm_begin_foo,asm_begin_bar"
        -asm-end "asm_end_foo,asm_end_bar"
```

`asm_begin_foo` and `asm_begin_bar` mark the beginning of the assembly source code sections to be ignored. `asm_end_foo` and `asm_end_bar` mark the end of those respective sections.

Tips

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

See Also

Topics
"Specify Polyspace Analysis Options"

-author

Specify project author

Syntax

```
-author "value"
```

Description

`-author "value"` assigns an author to the Polyspace project. The name appears as the project owner in Polyspace Metrics and on generated reports.

The default value is the user name of the current user, given by the DOS or UNIX command `whoami`.

In the user interface of the Polyspace desktop products, select to specify the Project name, Version, and Author parameters in the **Polyspace Project – Properties** dialog box.

Examples

Assign a project author to your Polyspace Project.

*   Bug Finder:

    ```
    polyspace-bug-finder -author "John Smith"
    ```
*   Code Prover:

    ```
    polyspace-code-prover -author "John Smith"
    ```
*   Bug Finder Server:

    ```
    polyspace-bug-finder-server -author "John Smith"
    ```
*   Code Prover Server:

    ```
    polyspace-code-prover-server -author "John Smith"
    ```

Tips

This option is not required for a Polyspace as You Code analysis.

See Also

`-date` | `-prog`

Topics
"Specify Polyspace Analysis Options"

-code-behavior-specifications

Map imprecisely analyzed function to standard function for precise analysis

Syntax

`-code-behavior-specifications` *file*

Description

`-code-behavior-specifications` *file* allows you to associate certain behaviors with elements of your code and modify the results of checks on those elements. Here, *file* is an XML file that assigns specific behaviors to code elements such as functions.

For instance, you can:

- Map your library functions to corresponding standard functions that Polyspace recognizes. Mapping to standard library functions can help with precision improvement or automatic detection of new threads.

- Specify that a function has a special behavior or must be subjected to special checks.

  For instance, you can specify that a function must only take addresses of initialized variables as arguments, or that a function must not be used altogether.

If you run verification from the command line, specify the absolute path to the XML files or path relative to the folder from which you run the command. If you run verification from the user interface (desktop products only), specify the option along with an absolute path to the XML file in the **Other** field. See `Other`. Note that a report generated from the analysis results only show the use of this option and not the details of which behaviors were associated with code elements.

A sample template file `code-behavior-specifications-template.xml` shows the XML syntax. The file is in *polyspaceroot*\polyspace\verifier\cxx\ where *polyspaceroot* is the Polyspace installation folder.

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

Using Option for Precision Improvement

XML Syntax: `<function name="`*custom_function*`" std="`*std_function*`"> </function>`

Use this entry in the XML file to reduce the number of orange checks from imprecise Code Prover analysis of your function (or false negatives from an imprecise Bug Finder analysis). Sometimes, the verification does not analyze certain kinds of functions precisely because of inherent limitations in static verification. In those cases, if you find a standard function that is a close analog of your function, use this mapping. Though your function itself is not analyzed, the analysis is more precise at the locations where you call the function. For instance, if the verification cannot analyze your function `cos32` precisely and considers full range for its return value, map it to the `cos` function for a return value in [-1,1].

The verification ignores the body of your function. However, the verification emulates your function behavior in the following ways:

- The verification assumes the same return values for your function as the standard function.

  For instance, if you map your function `cos32` to the standard function `cos`, the verification assumes that `cos32` returns values in [-1,1].

- The verification checks for the same issues as it checks with the standard function.

  For instance, if you map your function `acos32` to the standard function `acos`, the `Invalid use of standard library routine` check determines if the argument of `acos32` is in [-1,1].

The functions that you can map to include:

- Standard library functions from `math.h`.
- Memory management functions from `string.h`.
- `__ps_meminit`: A function specific to Polyspace that initializes a memory area.

  Sometimes, the verification does not recognize your memory initialization function and produces an orange `Non-initialized local variable` check on a variable that you initialized through this function. If you know that your memory initialization function initializes the variable through its address, map your function to `__ps_meminit`. The check turns green.

- `__ps_lookup_table_clip`: A function specific to Polyspace that returns a value within the range of the input array.

  Sometimes, the verification considers full range for the return values of functions that look up values in large arrays (look-up table functions). If you know that the return value of a look-up table function must be within the range of values in its input array, map the function to `__ps_lookup_table_clip`.

  In code generated from models, the verification by default makes this assumption for look-up table functions. To identify if the look-up table uses linear interpolation and no extrapolation, the verification uses the function names. Use the mapping only for handwritten functions, for instance, functions in a C/C++ S-Function block. The names of those functions do not follow specific conventions. You must explicitly specify them.

See also "Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries".

Using Option for Concurrency Detection

XML Syntax: `<function name="`*custom_function*`" std="`*std_function*`"> </function>`

Use this entry in the XML file for automatic detection of thread-creation functions and functions that begin and end critical sections. Polyspace supports automatic detection for certain families of multitasking primitives only. Extend the support using this XML entry.

If your thread-creation function, for instance, does not belong to one of the supported families, map your function to a supported concurrency primitive.

See "Extend Concurrency Defect Checkers to Unsupported Multithreading Environments".

Using Option for Blacklisting Functions

This section applies only to a Bug Finder analysis.

XML Syntax:

```
<function name="function_name">
    <behavior name="FORBIDDEN_FUNC">
</function>
```

Use this entry in the XML file to specify if the function must not be used in your source code.

See "Flag Deprecated or Unsafe Functions Using Bug Finder Checkers".

Using Option for Extending Initialization Checks

XML Syntax:

```
<function name="function_name">
    <check name="ARGUMENT_POINTS_TO_INITIALIZED_VALUE" arg="n"/>
</function>
```

The number *n* specifies which argument must be checked for buffer initialization.

Use this entry in the XML file to specify if the pointer argument to a function must point to an initialized buffer.

See .

Using Option for Modifying Global Behavior

XML Syntax:

```
  <global_scope>
    <parameter name="MAX_NUMBER_NESTED_LEVEL_CONTROL_FLOW" value="n1"/>
    <parameter name="MAX_NUMBER_NESTED_LEVEL_INCLUDES" value="n2"/>
    <parameter name="MAX_NUMBER_CONSTANT_IN_ENUMERATION" value="n3"/>
    <parameter name="MAX_NUMBER_MACROS_TRANSLATION_UNIT" value="n4"/>
    <parameter name="MAX_NUMBER_MEMBERS_IN_STRUCT" value="n5"/>
    <parameter name="MAX_NUMBER_NESTED_MEMBERS_IN_STRUCT" value="n6"/>
    <parameter name="NUMBER_SIGNIFICANT_CHARACTER_EXTERNAL_IDENTIFIER" value="n7"/>
    <parameter name="NUMBER_SIGNIFICANT_CHARACTER_INTERNAL_IDENTIFIER" value="n8"/>
  </global_scope>
```

Here, *n1,..,n8* specifies numerical values.

Use the entries *n1,..,n6* for the following parameters to specify limits on certain aspects of your program. The modifications affect the checking of `MISRA C:2012 Rule 1.1`.

- • `MAX_NUMBER_NESTED_LEVEL_CONTROL_FLOW`: Maximum depth of nesting allowed in control flow statements.
  - `MAX_NUMBER_NESTED_LEVEL_INCLUDES`: Maximum levels of inclusion allowed using include files.
  - `MAX_NUMBER_CONSTANT_IN_ENUMERATION`: Maximum number of constants allowed in an enumeration.
  - `MAX_NUMBER_MACROS_TRANSLATION_UNIT`Maximum number of macros allowed in a translation unit.
  - `MAX_NUMBER_MEMBERS_IN_STRUCT`: Maximum number of members allowed in a structure.
  - `MAX_NUMBER_NESTED_MEMBERS_IN_STRUCT`: Maximum levels of nesting allowed in a structure.

Use the entries *n7* and *n8* to specify how many characters must be compared to determine if two identifiers as identical. The modifications affect the checking of Rules 5.x.

- `NUMBER_SIGNIFICANT_CHARACTER_EXTERNAL_IDENTIFIER`: Number of characters to compare for external identifiers. External identifiers are ones declared with global scope or storage class `extern`.
- `NUMBER_SIGNIFICANT_CHARACTER_INTERNAL_IDENTIFIER`: Number of characters to compare for internal identifiers.

Examples

The examples in the next sections refer to a Code Prover analysis. For Bug Finder examples, see:

- "Extend Bug Finder Checkers for Standard Library Functions to Custom Libraries"
- "Flag Deprecated or Unsafe Functions Using Bug Finder Checkers"
- "Extend Concurrency Defect Checkers to Unsupported Multithreading Environments"

Specify Mapping to Standard Function

You can adapt the sample mapping XML file provided with your Polyspace installation and map your function to a standard function.

Suppose the default verification produces an orange `User assertion` check on this code:

```
double x = acos32(1.0) ;
assert(x <= 2.0);
```

Suppose you know that the function `acos32` behaves like the function `acos` and the return value is 0. You expect the check on the `assert` statement to be green. However, the verification considers that `acos32` returns any value in the range of type `double` because `acos32` is not precisely analyzed. The check is orange. To map your function `acos32` to `acos`:

1 Copy the file `code-behavior-specifications-template.xml` from *polyspaceroot* `\polyspace\verifier\cxx\` to another location, for instance, `"C:\Polyspace_projects \Common\Config_files"`. Change the write permissions on the file.

2 To map your function to a standard function, modify the contents of the XML file. To map your function `acos32` to the standard library function `acos`, change the following code:

   ```
   <function name="my_lib_cos" std="acos"> </function>
   ```

   To:

   ```
   <function name="acos32" std="acos"> </function>
   ```

3 Specify the location of the file for verification:

   - Code Prover:

     ```
     polyspace-code-prover -code-behavior-specifications
       "C:\Polyspace_projects\Common\Config_files
         \code-behavior-specifications-template.xml"
     ```

   - Code Prover Server:

```
polyspace-code-prover-server -code-behavior-specifications
  "C:\Polyspace_projects\Common\Config_files
   \code-behavior-specifications-template.xml"
```

Specify Mapping to Standard Function with Argument Remapping

Sometimes, the arguments of your function do not map one-to-one with arguments of the standard function. In those cases, remap your function argument to the standard function argument. For instance:

- `__ps_lookup_table_clip`:

  This function specific to Polyspace takes only a look-up table array as argument and returns values within the range of the look-up table. Your look-up table function might have additional arguments besides the look-up table array itself. In this case, use argument remapping to specify which argument of your function is the look-up table array.

  For instance, suppose a function `my_lookup_table` has the following declaration:

  ```
  double my_lookup_table(double u0, const real_T *table,
                                         const double *bp0);
  ```

  The second argument of your function `my_lookup_table` is the look-up table array. In the file `code-behavior-specifications-template.xml`, add this code:

  ```
  <function name="my_lookup_table" std="__ps_lookup_table_clip">
      <mapping std_arg="1" arg="2"></mapping>
  </function>
  ```

  When you call the function:

  ```
  res = my_lookup_table(u, table10, bp);
  ```

  The verification interprets the call as:

  ```
  res =__ps_lookup_table_clip(table10);
  ```

  The verification assumes that the value of `res` lies within the range of values in `table10`.

- `__ps_meminit`:

  This function specific to Polyspace takes a memory address as the first argument and a number of bytes as the second argument. The function assumes that the bytes in memory starting from the memory address are initialized with a valid value. Your memory initialization function might have additional arguments. In this case, use argument remapping to specify which argument of your function is the starting address and which argument is the number of bytes.

  For instance, suppose a function `my_meminit` has the following declaration:

  ```
  void my_meminit(enum InitKind k, void* dest, int is_aligned,
                              unsigned int size);
  ```

  The second argument of your function is the starting address and the fourth argument is the number of bytes. In the file `code-behavior-specifications-template.xml`, add this code:

  ```
  <function name="my_meminit" std="__ps_meminit">
      <mapping std_arg="1" arg="2"></mapping>
      <mapping std_arg="2" arg="4"></mapping>
  </function>
  ```

When you call the function:

```
my_meminit(INIT_START_BY_END, &buffer, 0, sizeof(buffer));
```

The verification interprets the call as:

```
__ps_meminit(&buffer, sizeof(buffer));
```

The verification assumes that `sizeof(buffer)` number of bytes starting from `&buffer` are initialized.

- `memset`: Variable number of arguments.

  If your function has variable number of arguments, you cannot map it directly to a standard function without explicit argument remapping. For instance, say your function is declared as:

  ```
  void* my_memset(void*, int, size_t, ...)
  ```

  To map the function to the `memset` function, use the following mapping:

  ```
  <function name="my_memset" std="memset">
      <mapping std_arg="1" arg="1"></mapping>
      <mapping std_arg="2" arg="2"></mapping>
      <mapping std_arg="3" arg="3"></mapping>
  </function>
  ```

Effect of Mapping on Precision

These examples show the result of mapping certain functions to standard functions:

- `my_acos → acos`:

  If you use the mapping, the `User assertion` check turns green. The verification assumes that the return value of `my_acos` is 0.

  - *Before mapping*:

    ```
    double x = my_acos(1.0);
    assert(x <= 2.0);
    ```

  - *Mapping specification*:

    ```
    <function name="my_acos" std="acos">
    </function>
    ```

  - *After mapping*:

    ```
    double x = my_acos(1.0);
    assert(x <= 2.0);
    ```

- `my_sqrt → sqrt`:

  If you use the mapping, the `Invalid use of standard library routine` check turns red. Otherwise, the verification does not check whether the argument of `my_sqrt` is nonnegative.

  - *Before mapping*:

    ```
    res = my_sqrt(-1.0);
    ```

  - *Mapping specification*:

```
<function name="my_sqrt" std="sqrt">
</function>
```

- *After mapping*:

```
res = my_sqrt(-1.0);
```

- `my_lookup_table` (argument 2) → `__ps_lookup_table_clip` (argument 1):

  If you use the mapping, the `User assertion` check turns green. The verification assumes that the return value of `my_lookup_table` is within the range of the look-up table array `table`.

  - *Before mapping*:

```
double table[3] = {1.1, 2.2, 3.3}
.
.
double res = my_lookup_table(u, table, bp);
assert(res >= 1.1 && res <= 3.3);
```

  - *Mapping specification*:

```
<function name="my_lookup_table" std="__ps_lookup_table_clip">
    <mapping std_arg="1" arg="2"></mapping>
</function>
```

  - *After mapping*:

```
double table[3] = {1.1, 2.2, 3.3}
.
.
res_real = my_lookup_table(u, table9, bp);
assert(res_real >= 1.1 && res_real <= 3.3);
```

- `my_meminit` → `__ps_meminit`:

  If you use the mapping, the `Non-initialized local variable` check turns green. The verification assumes that all fields of the structure x are initialized with valid values.

  - *Before mapping*:

```
struct X {
  int field1 ;
  int field2 ;
};
.
.
struct X x;
my_meminit(&x, sizeof(struct X));
return x.field1;
```

  - *Mapping specification*:

```
<function name="my_meminit" std="__ps_meminit">
    <mapping std_arg="1" arg="1"></mapping>
    <mapping std_arg="2" arg="2"></mapping>
</function>
```

  - *After mapping*:

```
struct X {
  int field1 ;
  int field2 ;
```

```
    };
    .
    .
    struct X x;
    my_meminit(&x, sizeof(struct X));
    return x.field1;
```

- my_meminit → __ps_meminit:

  If you use the mapping, the `Non-initialized local variable` check turns red. The verification assumes that only the field `field1` of the structure `x` is initialized with valid values.

  - *Before mapping*:

    ```
    struct X {
      int field1 ;
      int field2 ;
    };
    .
    .
    struct X x;
    my_meminit(&x, sizeof(int));
    return x.field2;
    ```

  - *Mapping specification*:

    ```
    <function name="my_meminit" std="__ps_meminit">
    </function>
    ```

  - *After mapping*:

    ```
    struct X {
      int field1 ;
      int field2 ;
    };
    .
    .
    struct X x;
    my_meminit(&x, sizeof(int));
    return x.field2;
    ```

See Also

Topics
"Specify Polyspace Analysis Options"

Introduced in R2016b

-consider-external-array-access-unsafe

Remove the default assumption that external arrays of unspecified size can be safely accessed at any index

Syntax

```
-consider-external-array-access-unsafe
```

Description

`-consider-external-array-access-unsafe` removes the default Code Prover assumption that external arrays of unspecified size can be safely accessed at any index. By default, because of this assumption, Code Prover shows green **Out of bounds array index** checks on external array accesses code despite their size being unknown. If you use this option, the same check is orange indicating that the access is not proven safe and requires manual inspection.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Run Code Prover on this example with and without the option.

```
extern int arr[];

int getFifthElement(void) {
    return arr[5];
}
```

The array access shows a green **Out of bounds array index** check without the option but an orange check with the option.

See Also
```
Generic target options|Out of bounds array index|Target processor type (-
target)
```

Topics
"Specify Polyspace Analysis Options"

-custom-target

Create a custom target processor with specific data type sizes

Syntax

```
-custom-target target_sizes
```

Description

`-custom-target target_sizes` defines a custom target processor for the Polyspace analysis. The target processor definition includes sizes in bytes of fundamental data types, signedness of plain `char`, alignment of structures and underlying types of standard `typedef`-s such as `size_t`, `ptrdiff_t` and `wchar_t`.

`target_sizes` is a comma-separated list specifying these values. From left to right, the values are the following. If a data type is not supported, -1 is used for its size.

Specification	Possible Values
Whether plain `char` is signed	`true` or `false`
Size of `char` in bits Other sizes are in bytes.	Number
Size of `short`	Number
Size of `int`	Number
Size of `short long`	Number
Size of `long`	Number
Size of `long long`	Number
Size of `float`	Number
Size of `double`	Number
Size of `long double`	Number
Size of pointer	Number
Maximum alignment of all integer types	Number
Maximum alignment of variables of type `struct` or `union`	Number
Endianness	`little` or `big`
Underlying type of `size_t`	unknown, `unsigned_char`, `unsigned_short`, `unsigned_int`, `unsigned_long`, or `unsigned_long_long`
Underlying type of `ptrdiff_t`	unknown, `signed_char`, `short`, `int`, `long`, or `long_long`
Underlying type of `wchar_t`	unknown, `short`, `unsigned_short`, `int`, `unsigned_int`, `long`, or `unsigned_long`

Typically, this option is used when the `polyspace-configure` command creates an options file for the subsequent Polyspace analysis. However, you can directly enter this option when manually writing options files. This option is useful in situations where your target specifications are not covered by one of the predefined target processors. See `Target processor type (-target)`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

An usage of the option looks like this:

`-custom-target false,8,2,4,-1,4,8,4,8,8,4,8,1,little,unsigned_int,int,unsigned_int`

The option argument translates to the following target specification.

Specification	Possible Values
Whether plain `char` is signed	`false`
Size of `char`	8 bits
Size of `short`	2 bytes
Size of `int`	4 bytes
Size of `short long`	`short long` is not supported.
Size of `long`	4 bytes
Size of `long long`	8 bytes
Size of `float`	4 bytes
Size of `double`	8 bytes
Size of `long double`	8 bytes
Size of pointer	4 bytes
Maximum alignment of all integer types	8 bytes
Maximum alignment of variables of type `struct` or `union`	1 byte
Endianness	`little`
Underlying type of `size_t`	`unsigned_int`
Underlying type of `ptrdiff_t`	`int`
Underlying type of `wchar_t`	`unsigned_int`

Tips

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

See Also
`Generic target options` | `Target processor type (-target)`

Topics
"Specify Polyspace Analysis Options"

-date

Specify date of analysis

Syntax

`-date "`*date*`"`

Description

`-date "`*date*`"` specifies the date stamp for the analysis in the format `dd/mm/yyyy`. By default the value is the date the analysis starts.

Examples

Assign a date to your Polyspace Project:

* Bug Finder:

  `polyspace-bug-finder -date "15/03/2012"`
* Code Prover:

  `polyspace-code-prover -date "15/03/2012"`
* Bug Finder Server:

  `polyspace-bug-finder-server -date "15/03/2012"`
* Code Prover Server:

  `polyspace-code-prover-server -date "15/03/2012"`

Tips

This option is not required for a Polyspace as You Code analysis.

See Also
`-author | -date`

Topics
"Specify Polyspace Analysis Options"

-doc | -documentation

Display Polyspace documentation in help browser

Syntax

```
-doc
-documentation
```

Description

`-doc` and `-documentation` opens Polyspace documentation in a help browser. You can see information such as getting started, workflows and reference pages for commands and analysis options. You can also search through the documentation in the help browser.

Examples

Display Polyspace documentation in a help browser:

- Bug Finder:

  ```
  polyspace-bug-finder -doc
  polyspace-bug-finder -documentation
  ```

- Code Prover:

  ```
  polyspace-code-prover -doc
  polyspace-code-prover -documentation
  ```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -doc
  polyspace-bug-finder-server -documentation
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -doc
  polyspace-code-prover-server -documentation
  ```

See Also
```
-h[elp]
```

-dump-preprocessing-info

Show all macros implicitly defined during a particular analysis

Syntax

`-dump-preprocessing-info`

Description

`-dump-preprocessing-info` prints all the macros implicitly defined (or undefined) during a particular Polyspace analysis. The macro definitions come from:

- Your specification for the option `Compiler (-compiler)`

  Polyspace emulates a compiler by defining the compiler-specific macros.
- Macros defined (or undefined) in the Polyspace implementation of Standard Library headers
- Macros that you explicitly define (or undefine) using the options `Preprocessor definitions (-D)` and `Disabled preprocessor definitions (-U)`

Use this option only if you want to know how Polyspace defines a specific macro. In case you want to use a different definition for the macro, you can then override the current definition.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`. On the **Output Summary** pane, you can see each macro definition on a separate line. You can search for the macro name in the user interface and click the line with the macro name to see further details in the **Detail** pane.

Examples

Suppose that you use the ARM v6 compiler for building your source code. For the Polyspace analysis, you use the value `armclang` for the option `Compiler (-compiler)`. Suppose that you want to know what Polyspace uses as definition for the macro __ARM_ARCH.

1 Enter the following command and pipe the console output to a file that you can search later:

```
polyspace-bug-finder -sources aFile.c -compiler armclang -dump-preprocessing-info
```

aFile.c can be a simple C file. You can also replace `polyspace-bug-finder` with `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server`.

2 Search for __ARM_ARCH in the file containing the console output. You can see the line with the macro definition:

```
Remark: Definition of macro __ARM_ARCH (pre-processing __polyspace__stdstubs.c)
|#define __ARM_ARCH 8
|defined by syntax extension xml file
|predefined macro
```

In this example, the macro is set to the value 8.

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

See Also
`Compiler (-compiler)`

Topics
"Specify Polyspace Analysis Options"

-generate-launching-script-for

Extract information from project file

Syntax

`-generate-launching-script-for` *PRJFILE*

Description

`-generate-launching-script-for` *PRJFILE* extracts information from a project file *PRJFILE* (created in the user interface of the Polyspace desktop products) so that you can run an analysis from the command line. For each project module and each configuration in each module, a folder is created containing the following files::

- `source_command.txt` — List of source files for the `-sources-list-file` option.
- `options_command.txt` — List of the analysis options for the `-options-file` option.
- `temporal_exclusions.txt` — List of temporal exclusions, generated only if you specify the Temporally exclusive tasks (`-temporal-exclusions-file`) option.
- `.polyspace_conf.psprj` — A copy of the project file Polyspace used to generate the scripting files.
- `launchingCommand.sh` (UNIX) or `launchingCommand.bat` (DOS) — shell script that calls the correct commands. The script also calls any options that cannot be given to the `-options-file` command, such as `-batch` or `-add-to-results-repository`. You can give this file additional analysis options as parameters.

After you set up a project in the Polyspace user interface, you can create this script from the resulting project file (with extension `.psprj`). The script that Polyspace generates runs the same analysis as a run in the user interface. If your project runs without errors in the Polyspace user interface, the script runs without errors at the command line.

To generate the script, you must run the command from the same folder as the project file.

Examples

Extract information to run `myproject` from the command line. Use this option with the desktop binary `polyspace`:

- Bug Finder:

  `polyspace -generate-launching-script-for myproject.psprj -bug-finder`
- Code Prover:

  `polyspace -generate-launching-script-for myproject.psprj`

See Also

Topics
"Configure Polyspace Analysis Options in User Interface and Generate Scripts"

-h | -help

Display list of possible options

Syntax

```
-h
-help
```

Description

`-h` and `-help` display the list of possible options in the command window along with option argument syntax.

Examples

Display the command-line help:

- Bug Finder:

  ```
  polyspace-bug-finder -h
  polyspace-bug-finder -help
  ```

- Code Prover:

  ```
  polyspace-code-prover -h
  polyspace-code-prover -help
  ```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -h
  polyspace-bug-finder-server -help
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -h
  polyspace-code-prover-server -help
  ```

See Also

`-doc | -documentation`

-I

Specify include folder for compilation

Syntax

```
-I folder
```

Description

-I *folder* specifies a folder that contains include files required for compiling your sources. You can specify only one folder for each instance of -I. However, you can specify this option multiple times.

The analysis looks for include files relative to the folder paths that you specify. For instance, if your code contains the preprocessor directive #include<../mylib.h> and you include the folder:

```
C:\My_Project\MySourceFiles\Includes
```

the folder C:\My_Project\MySourceFiles must contain a file mylib.h.

The analysis automatically includes the ./sources folder (if it exists) after the include folders that you specify.

Examples

Include two folders with the analysis:

- Bug Finder:

  ```
  polyspace-bug-finder -I /com1/inc -I /com1/sys/inc
  ```
- Code Prover:

  ```
  polyspace-code-prover -I /com1/inc -I /com1/sys/inc
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -I /com1/inc -I /com1/sys/inc
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -I /com1/inc -I /com1/sys/inc
  ```

The source folder is implicitly included. Include files in the source folder can be found automatically without explicit inclusion of the source folder with the -I option.

Tips

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

See Also

Topics
"Specify Polyspace Analysis Options"

-import-comments

Import review information from previous analysis

Syntax

`-import-comments` *resultsFolder*

Description

`-import-comments` *resultsFolder* imports the review information (status, severity and additional notes) from a previous analysis, as specified by the results folder.

You can import review information from the same type of results only. For instance:

- You cannot import review information from a results of a Bug Finder checker to a Code Prover run-time check. Even when the checker names sound similar, the underlying semantics of Bug Finder and Code Prover can be different. The only exception is checkers for coding rules. You can import comments between Bug Finder and Code Prover for coding rule violations.

- You cannot import review information from results of a file-by-file verification in Code Prover to results of a regular Code Prover verification.

You can also use this option to create a baseline for the analysis results. In the Polyspace user interface, if you click the **New** button, only the analysis results that are new compared to the baseline remain in the results list.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Import review information from the previous results:

- Bug Finder:

```
polyspace-bug-finder -sources filename
        -import-comments C:\Results\myProj\1.2
```

- Code Prover:

```
polyspace-code-prover -sources filename
        -import-comments C:\Results\myProj\1.2
```

- Bug Finder Server:

```
polyspace-bug-finder-server -sources filename
        -import-comments C:\Results\myProj\1.2
```

- Code Prover Server:

```
polyspace-code-prover-server -sources filename
        -import-comments C:\Results\myProj\1.2
```

Tips

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

See Also

`-v[ersion]|polyspace-comments-import`

Topics
"Import Review Information from Previous Polyspace Analysis"

-list-all-values

Display valid option arguments for a given command-line option

Syntax

`-list-all-values` *option*

Description

`-list-all-values` *option* displays all the valid option arguments for the command-line option *option*.

Examples

Display the valid option arguments for option `-misra3`:

- Polyspace Bug Finder:

  `polyspace-bug-finder -list-all-values -misra3`
- Polyspace Code Prover:

  `polyspace-code-prover -list-all-values -misra3`
- Polyspace Bug Finder Server:

  `polyspace-bug-finder-server -list-all-values -misra3`
- Polyspace Code Prover Server:

  `polyspace-code-prover-server -list-all-values -misra3`

See Also

Topics
"Specify Polyspace Analysis Options"

Introduced in R2020a

-max-processes

Specify maximum number of processors for analysis

Syntax

`-max-processes` *num*

Description

`-max-processes` *num* specifies the maximum number of processes that you want the analysis to use. On a multicore system, the software parallelizes the analysis and creates the specified number of processes to speed up the analysis. The valid range of *num* is 1 to 128.

Unless you specify this option, a Code Prover verification uses up to four processes. If you have fewer than four processes, the verification uses the maximum available number. To increase or restrict the number of processes, use this option.

Unless you specify this option, a Bug Finder analysis uses the maximum number of available processes. Use this option to restrict the number of processes used.

To use this option effectively, determine the number of processors available for use. If the number of processes you create is greater than the number of processors available, the analysis does not benefit from the parallelization. Check the system information in your operating system.

Note that when you start a verification, a message states the number of logical processors detected on your system. However, the analysis is parallelized to the physical processor cores on a machine. Multithreading implementations such as hyper-threading is not taken into account.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Disable parallel processing during the analysis:

*   Bug Finder:

    `polyspace-bug-finder -max-processes 1`
*   Code Prover:

    `polyspace-code-prover -max-processes 1`
*   Bug Finder Server:

    `polyspace-bug-finder-server -max-processes 1`
*   Code Prover Server:

    `polyspace-code-prover-server -max-processes 1`

Tips

You must have at least 4 GB of RAM per processor for analysis. For instance, if your machine has 16 GB of RAM, do not use this option to specify more than four processes.

This option is not useful in a Polyspace as You Code analysis.

See Also

Topics
"Specify Polyspace Analysis Options"

-no-assumption-on-absolute-addresses

Remove assumption that absolute address usage is valid

Syntax

`-no-assumption-on-absolute-addresses`

Description

This option affects a Code Prover analysis only.

`-no-assumption-on-absolute-addresses` removes the default assumption that absolute addresses used in your code are valid. If you use this option, the verification produces an orange `Absolute address usage` check when you assign an absolute address to a pointer. Otherwise, the check is green by default.

The type of the pointer to which you assign the address determines the initial value stored in the address. For instance, if you assign the address to an `int*` pointer, following this check, the verification assumes that the memory zone that the address points to is initialized with an `int` value. The value can be anything allowed for the data type `int`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

The use of option `-no-assumption-on-absolute-addresses` can increase the number of orange checks in your code. For instance, the following table shows an additional orange check with the option enabled.

Absolute Address Usage Green	Absolute Address Usage Orange
```void main() { int *p = (int *)0x32; int x; x=*p; }``` In this example, the software produces: • A green **Absolute address usage** check when the address `0x32` is assigned to a pointer `p`. • A green **Illegally dereferenced pointer** check when the pointer `p` is read. x potentially has all values allowed for an `int` variable.	```void main() { int *p = (int *)0x32; int x; x=*p; }``` In this example, the software produces: • An orange **Absolute address usage** check when the address `0x32` is assigned to a pointer `p`. • A green **Illegally dereferenced pointer** check when the pointer `p` is read. x potentially has all values allowed for an `int` variable.

For best use of the **Absolute address usage** check, leave this check green by default during initial stages of development. During integration stage, use the option `-no-assumption-on-absolute-`

`addresses` and detect all uses of absolute memory addresses. Browse through them and make sure that the addresses are valid.

See Also

Topics
"Specify Polyspace Analysis Options"

Introduced in R2016a

-non-preemptable-tasks

Specify functions that represent nonpreemptable tasks

Syntax

`-non-preemptable-tasks` *function1*`[,`*function2*`[,...]]`

Description

`-non-preemptable-tasks` *function1*`[,`*function2*`[,...]]` specifies functions that represent nonpreemptable tasks.

The functions cannot be interrupted by other noncyclic tasks and cyclic tasks but can be interrupted by interrupts, preemptable or nonpreemptable. Noncyclic tasks are specified with the option `Tasks (-entry-points)`, cyclic tasks with the option `Cyclic tasks (-cyclic-tasks)` and interrupts with the option `Interrupts (-interrupts)`. For examples, see "Define Preemptable Interrupts and Nonpreemptable Tasks".

To specify a function as a nonpreemptable cyclic task, you must first specify the function as a cyclic or noncyclic task. The functions that you specify must have the prototype:

`void` *function_name*`(void);`

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Tips

This option is not useful in a Polyspace as You Code analysis.

See Also

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Critical section details (-critical-section-begin -critical-section-end)` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)`

Topics
"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"
"Define Preemptable Interrupts and Nonpreemptable Tasks"
"Concurrency Defects"

Introduced in R2016b

-options-for-sources

Specify analysis options specific to a source file

Syntax

```
-options-for-sources filename options
```

Description

`-options-for-sources` *filename* *options* associates a semicolon-separated list of Polyspace analysis options with the source file specified by *filename.*.

This option is primarily used when the `polyspace-configure` command creates an options file for the subsequent Polyspace analysis. The option `-options-for-sources` associates a group of analysis options such as include folders and macro definitions with specific source files.

However, you can directly enter this option when manually writing options files. This option is useful in situations where you want to associate a group of options with a specific source file without applying it to other files.

In the user interface of the Polyspace desktop products, you can create a Polyspace project from your build command. The project uses the option `-options-for-sources` to associate specific Polyspace analysis options with specific files. However, when you open the project in the user interface, you cannot see the use of this option. Open the project in a text editor to see this option.

Examples

In this sample options file, the include folder `/usr/lib/gcc/x86_64-linux-gnu/6/include` and the macros `__STDC_VERSION__` and `__GNUC__` are associated only with the source file `file.c` and not `fileAnother.c`.

```
-options-for-sources file.c;-I /usr/lib/gcc/x86_64-linux-gnu/6/include;
-options-for-sources file.c;-D __STDC_VERSION__=201112L;-D __GNUC__=6;
-sources file.c
-sources fileAnother.c
```

For the options used in this example, see:

- `-sources`
- `-I`
- Preprocessor definitions (-D)

Tips

When associating multiple options with a source file, if you use an option separator other than semicolon, use a second option `-options-for-sources-delimiter` to explicitly specify this separator. For instance, if you use the separator @, specify the additional option:

```
-options-for-sources-delimiter @
```

Otherwise, the analysis assumes a semicolon separator.

See Also

`-options-file` | `polyspace-configure`

Topics
"Specify Polyspace Analysis Options"

-preemptable-interrupts

Specify functions that represent preemptable interrupts

Syntax

```
-preemptable-interrupts function1[,function2[,...]]
```

Description

`-preemptable-interrupts` *function1*[,*function2*[,...]] specifies functions that represent preemptable interrupts.

The function acts as an interrupt in every way except that it can be interrupted by other interrupts, preemptable or nonpreemptable. Interrupts are specified with the option `Interrupts (-interrupts)`. For examples, see "Define Preemptable Interrupts and Nonpreemptable Tasks".

To specify a function as a preemptable interrupt, you must first specify the function as an interrupt. The functions that you specify must have the prototype:

```
void function_name(void);
```

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Tips

This option is not useful in a Polyspace as You Code analysis.

See Also
`-non-preemptable-tasks`|`-preemptable-interrupts`|`Critical section details (-critical-section-begin -critical-section-end)`|`Cyclic tasks (-cyclic-tasks)`|`Interrupts (-interrupts)`|`Tasks (-entry-points)`|`Temporally exclusive tasks (-temporal-exclusions-file)`

Topics
"Specify Polyspace Analysis Options"
"Analyze Multitasking Programs in Polyspace"
"Configuring Polyspace Multitasking Analysis Manually"
"Protections for Shared Variables in Multitasking Code"
"Define Preemptable Interrupts and Nonpreemptable Tasks"
"Concurrency Defects"

Introduced in R2016b

-options-file

Run Polyspace using list of options

Syntax

```
-options-file file
```

Description

`-options-file file` specifies a file which lists your analysis options. The file must be a text file with each option on a separate line. Use # to add comments to this file.

Examples

1   Create an options file called `listofoptions.txt` with your options. For example:

   • Bug Finder or Bug Finder Server:

   ```
   #These are the options for MyBugFinderProject
   -lang c
   -prog MyBugFinderProject
   -author jsmith
   -sources "mymain.c,funAlgebra.c,funGeometry.c"
   -target x86_64
   -compiler generic
   -dos
   -misra2 required-rules
   -do-not-generate-results-for all-headers
   -checkers default
   -disable-checkers concurrency
   -results-dir C:\Polyspace\MyBugFinderProject
   ```

   • Code Prover or Code Prover Server:

   ```
   #These are the options for MyCodeProverProject
   -lang c
   -prog MyCodeProverProject
   -author jsmith
   -sources "mymain.c,funAlgebra.c,funGeometry.c"
   -target x86_64
   -compiler generic
   -dos
   -misra2 required-rules
   -do-not-generate-results-for all-headers
   -main-generator
   -results-dir C:\Polyspace\MyCodeProverProject
   ```

2   Run Polyspace using options in the file `listofoptions.txt`:

   • Bug Finder:

   ```
   polyspace-bug-finder -options-file listofoptions.txt
   ```

   • Code Prover:

```
polyspace-code-prover -options-file listofoptions.txt
```
- Bug Finder Server:

```
polyspace-bug-finder-server -options-file listofoptions.txt
```
- Code Prover Server:

```
polyspace-code-prover-server -options-file listofoptions.txt
```

See Also

Topics
"Specify Polyspace Analysis Options"

-prog

Specify name of project

Syntax

`-prog projectName`

Description

`-prog projectName` specifies a name for your Polyspace project. This name must use only letters, numbers, underscores (_), dashes (-), or periods (.).

The name appears in the analysis log and a few other places.

Examples

Assign a name to your Polyspace project:

- Bug Finder:

  `polyspace-bug-finder -prog MyApp`
- Code Prover:

  `polyspace-code-prover -prog MyApp`
- Bug Finder Server:

  `polyspace-bug-finder-server -prog MyApp`
- Code Prover Server:

  `polyspace-code-prover-server -prog MyApp`

Tips

This option is not required for a Polyspace as You Code analysis.

See Also
`-author` | `-date`

Topics
"Specify Polyspace Analysis Options"

-regex-replace-rgx -regex-replace-fmt

Make replacements in preprocessor directives

Syntax

`-regex-replace-rgx` *matchFileName* `-regex-replace-fmt` *replacementFileName*

Description

`-regex-replace-rgx` *matchFileName* `-regex-replace-fmt` *replacementFileName* replaces tokens in preprocessor directives for the purposes of Polyspace analysis. The original source code is unchanged. You match a token using a regular expression in the file *matchFileName* and replace the token using a replacement in the file *replacementFileName*.

Use this option only to replace or remove tokens in the preprocessor directives *before preprocessing*. If a token in your source code causes a compilation error, you can typically replace or remove the token from the preprocessed code. Use the more convenient option `Command/script to apply to preprocessed files (-post-preprocessing-command)`. You cannot make the replacements in preprocessed code only for tokens in preprocessor directives.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

In the user interface, specify absolute paths to the text files with the search and replace patterns.

Examples

Suppose you want to replace `&_rom_beg` in this `#define` directive:

`#define ROM_BEG_ADDR (uint16_t)(&_rom_beg)`

and modify the directive to:

`#define ROM_BEG_ADDR (0x4000u)`

Specify this regular expression in a file `match.txt`:

`^#define\s+ROM_BEG_ADDR\s+\(uint16_t\)\(\&_rom_beg\)`

These elements are used in the regular expression:

- `^` asserts position at the start of a line.
- `\s+` represents one or more whitespace characters.

The characters `*`, `&`, `(` and `)` in the original expression are escaped with `\`. For a complete list of regular expressions, see Perl documentation.

Specify the replacement in a file `replace.txt`.

`#define ROM_BEG_ADDR \(0x4000u\)`

Specify the two text files during analysis with the options `-regex-replace-rgx` and `-regex-replace-fmt`:

- Bug Finder:

  ```
  polyspace-bug-finder -sources filename
                                -regex-replace-rgx match.txt
                                -regex-replace-fmt replace.txt
  ```

- Code Prover:

  ```
  polyspace-code-prover -sources filename
                                -regex-replace-rgx match.txt
                                -regex-replace-fmt replace.txt
  ```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -sources filename
                                -regex-replace-rgx match.txt
                                -regex-replace-fmt replace.txt
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -sources filename
                                -regex-replace-rgx match.txt
                                -regex-replace-fmt replace.txt
  ```

Tips

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

See Also

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

Topics
"Specify Polyspace Analysis Options"

-report-output-name

Specify name of report

Syntax

`-report-output-name` *reportName*

Description

`-report-output-name` *reportName* specifies the name of an analysis report.

The default name for a report is *Prog_Template.Format*:

- *Prog* is the name of the project specified by `-prog`.
- *TemplateName* is the type of report template specified by `-report-template`.
- *Format* is the file extension for the report specified by `-report-output-format`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Specify the name of the analysis report:

- Bug Finder:

  ```
  polyspace-bug-finder -report-template Developer
        -report-output-name Airbag_v3.doc
  ```
- Code Prover:

  ```
  polyspace-code-prover -report-template Developer
        -report-output-name Airbag_v3.doc
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -report-template Developer
        -report-output-name Airbag_v3.doc
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -report-template Developer
        -report-output-name Airbag_v3.doc
  ```

Tips

You cannot generate reports with Polyspace as You Code.

See Also
`Bug Finder and Code Prover report (-report-template)`|`Output format (-report-output-format)`

Topics
"Specify Polyspace Analysis Options"
"Generate Reports"

-results-dir

Specify the results folder

Syntax

```
-results-dir resultsFolder
```

Description

`-results-dir resultsFolder` specifies where to save the analysis results. The default location at the command line is the current folder.

Note that the results folder is cleaned up and repopulated at each run. To avoid accidental removal of files during the cleanup, instead of using an existing folder that contains other files, specify a dedicated folder for the Polyspace results.

If you are running analysis in the user interface of the Polyspace desktop products, see "Run Polyspace Analysis on Desktop".

Examples

Specify to store your results in the RESULTS folder:

- Bug Finder:

  ```
  polyspace-bug-finder -results-dir RESULTS
  ```
- Code Prover:

  ```
  polyspace-code-prover -results-dir RESULTS
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -results-dir RESULTS
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -results-dir RESULTS
  ```

You can create the name of the results folder based on the verification date and time. For instance, in a Bash shell, enter these commands to create a variable RESULTS that begins with `results_` and contains the current date and time:

```
export DATETIME=$(date +%d%B_%HH%M_%A)
export RESULTS=results_$DATE
```

You can then use the variable RESULTS as argument of the option `-results-dir`:

```
-results-dir $RESULTS
```

Tips

If you use Polyspace as You Code extensions in IDEs, this option is implemented through the IDE extension setting. You do not have to use this option explicitly. If you want to explicitly use this option, enter the option in an analysis options file. See options file.

See Also

Topics
"Specify Polyspace Analysis Options"

-scheduler

Specify cluster or job scheduler

Syntax

`-scheduler` *schedulingOption*

Description

`-scheduler` *schedulingOption* specifies the head node of the MATLAB Parallel Server cluster that manages Polyspace analysis submissions from multiple clients and allocates the analysis to worker nodes. You use this option along with the option `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)` to offload an analysis from a desktop to a remote cluster. Note that you use this option with the commands in the desktop products (`polyspace-bug-finder` and `polyspace-code-prover`) and not the commands in the server products (`polyspace-bug-finder-server` and `polyspace-code-prover-server`).

For more information, see "Install Products for Submitting Polyspace Analysis from Desktops to Remote Server".

Examples

Run a batch analysis on a remote server using one of these syntaxes for the job scheduler:

- Bug Finder:

  ```
  polyspace-bug-finder -batch -scheduler NodeHost
  polyspace-bug-finder -batch -scheduler 192.168.1.124:12400
  polyspace-bug-finder -batch -scheduler MJSName@NodeHost
  ```

- Code Prover:

  ```
  polyspace-code-prover -batch -scheduler NodeHost
  polyspace-code-prover -batch -scheduler 192.168.1.124:12400
  polyspace-code-prover -batch -scheduler MJSName@NodeHost
  ```

For details, see "Send Polyspace Analysis from Desktop to Remote Servers Using Scripts".

You can track the status of the job using the `polyspace-jobs-manager` command:

`polyspace-jobs-manager listjobs -scheduler NodeHost`

Tips

You cannot submit analysis jobs to a remote cluster with Polyspace as You Code.

See Also
`Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`

Topics
"Send Polyspace Analysis from Desktop to Remote Servers Using Scripts"

"Install Products for Submitting Polyspace Analysis from Desktops to Remote Server"

-show-similar-overflows

Show all overflows in wrap-around mode

Syntax

```
-show-similar-overflows
```

Description

`-show-similar-overflows` causes all overflows to be shown in wrap-around mode even if they come from the same root cause.

If you select `warn-with-wrap-around` for the option `Overflow mode for signed integer (-signed-integer-overflows)` or `Overflow mode for unsigned integer (-unsigned-integer-overflows)`, values that overflow are wrapped. For instance, the value `INT_MAX + 1` wraps around to `INT_MIN`. A path with an overflowing value continues beyond the overflow with the wrapped value and can lead to a similar overflow several lines later. By default, Code Prover detects overflows from the same root cause and shows only the first of similar overflows. If you fix this overflow, the subsequent overflows are also fixed. If you want to see all overflows in wrap-around mode, use the option `-show-similar-overflows`.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

In this example, the value of `var` is unknown and edge cases can lead to overflows in the operation `*copy1 = var * 2`. The same edge cases also lead to overflows in the next operation. Code Prover shows an orange **Overflow** check on the first operation only.

```
int input();

void getEven(int* copy1, int* copy2) {
    int var;
    var = input();
    *copy1 = var * 2;
    *copy2 = var * 2;
}
```

If you use the option `-show-similar-overflows`, Code Prover shows orange **Overflow** checks on both operations.

```
int input();

void getEven(int* copy1, int* copy2) {
    int var;
    var = input();
    *copy1 = var * 2;
    *copy2 = var * 2;
}
```

See Also

`-options-file`|`Overflow`|`Overflow mode for signed integer (-signed-integer-overflows)`|`Overflow mode for unsigned integer (-unsigned-integer-overflows)`

Topics
"Specify Polyspace Analysis Options"

Introduced in R2020b

-sources

Specify source files

Syntax

```
-sources file1[,file2,...]
-sources file1 -sources file2
```

Description

`-sources file1[,file2,...]` or `-sources file1 -sources file2` specifies the list of source files that you want to analyze. You can use standard UNIX wildcards with this option to specify your sources.

The source files are compiled in the order in which they are specified.

Examples

Analyze the files `mymain.c`, `funAlgebra.c`, and `funGeometry.c`.

- Bug Finder:

  ```
  polyspace-bug-finder -sources mymain.c
      -sources funAlgebra.c -sources funGeometry.c
  ```

- Code Prover:

  ```
  polyspace-code-prover -sources mymain.c
      -sources funAlgebra.c -sources funGeometry.c
  ```

- Bug Finder Server:

  ```
  polyspace-bug-finder-server -sources mymain.c
      -sources funAlgebra.c -sources funGeometry.c
  ```

- Code Prover Server:

  ```
  polyspace-code-prover-server -sources mymain.c
      -sources funAlgebra.c -sources funGeometry.c
  ```

- Polyspace as You Code:

  ```
  polyspace-bug-finder-access -sources myfile.c
  ```

  Note that you can only analyze one file at a time with Polyspace as You Code. If you use Polyspace as You Code extensions in IDEs, you do not have to specify this option. The analysis runs on the file that is currently active in your IDE.

See Also

`-sources-list-file` | `polyspace-configure`

Topics
"Specify Polyspace Analysis Options"

-sources-list-file

Specify file containing list of sources

Syntax

```
-sources-list-file file_path
```

Description

`-sources-list-file` *file_path* specifies the absolute path to a text file that lists each file name that you want to analyze.

To specify your sources in the text file, on each line, specify the path to a source file. You can specify an absolute path or a path relative to the folder from which you are running the analysis. For example:

```
C:\Sources\myfile.c
C:\Sources2\myfile2.c
```

Examples

Run analysis on files listed in `files.txt`:

- Bug Finder:

  ```
  polyspace-bug-finder -sources-list-file "C:\Analysis\files.txt"
  polyspace-bug-finder -sources-list-file "/home/polyspace/files.txt"
  ```
- Code Prover:

  ```
  polyspace-code-prover -sources-list-file "C:\Analysis\files.txt
  polyspace-code-prover -sources-list-file "/home/polyspace/files.txt"
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -sources-list-file "C:\Analysis\files.txt"
  polyspace-bug-finder-server -sources-list-file "/home/polyspace/files.txt"
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -sources-list-file "C:\Analysis\files.txt
  polyspace-code-prover-server -sources-list-file "/home/polyspace/files.txt"
  ```

Tips

You cannot use this option with Polyspace as You Code.

See Also

Topics
"Specify Polyspace Analysis Options"

-submit-job-from-previous-compilation-results

Specify that the analysis job must be resubmitted without recompilation

Syntax

`-submit-job-from-previous-compilation-results`

Description

`-submit-job-from-previous-compilation-results` specifies that the Polyspace analysis must start after the compilation phase with compilation results from a previous analysis. The option is primarily useful when offloading a Polyspace analysis from desktops to remote servers. If a remote analysis stops after compilation, for instance because of communication problems between the server and client computers, use this option. Note that you use this option with the commands in the desktop products (`polyspace-bug-finder` and `polyspace-code-prover`) and not the commands in the server products (`polyspace-bug-finder-server` and `polyspace-code-prover-server`).

When you perform a remote analysis:

1   On the local host computer, the Polyspace software performs code compilation and coding rule checking.
2   The analysis job is then submitted to the MATLAB job scheduler on the head node of the MATLAB Parallel Server cluster.
3   The head node of the MATLAB Parallel Server cluster assigns the verification job to a worker node, where the remaining phases of the Polyspace analysis occur.

If an analysis stops after completing the first step and you restart the analysis, use this option to reuse compilation results from the previous analysis. You thereby avoid restarting the analysis from the compilation phase.

If previous compilation results do not exist in the current folder, an error occurs. Remove the option and restart analysis from the compilation phase.

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Specify remote analysis with compilation results from a previous analysis:

*   Bug Finder:

    ```
    polyspace-bug-finder -batch -scheduler localhost
            -submit-job-from-previous-compilation-results
    ```
*   Code Prover:

    ```
    polyspace-code-prover -batch -scheduler localhost
            -submit-job-from-previous-compilation-results
    ```

Tips

You cannot submit analysis jobs to a remote cluster with Polyspace as You Code.

See Also

Topics
"Send Polyspace Analysis from Desktop to Remote Servers Using Scripts"
"Install Products for Submitting Polyspace Analysis from Desktops to Remote Server"

-tmp-dir-in-results-dir

Keep temporary files in results folder

Syntax

`-tmp-dir-in-results-dir`

Description

`-tmp-dir-in-results-dir` specifies that temporary files must be stored in a subfolder of the results folder. Use this option only when the standard temporary folder does not have enough disk space. If the results folder is mounted on a network drive, this option can slow down your processor.

To learn how Polyspace determines the temporary folder location, see "Storage of Temporary Files".

If you are running an analysis from the user interface (Polyspace desktop products only), on the **Configuration** pane, you can enter this option in the **Other** field. See `Other`.

Examples

Store temporary files in the results folder:

- Bug Finder:

  `polyspace-bug-finder -tmp-dir-in-results-dir`
- Code Prover:

  `polyspace-code-prover -tmp-dir-in-results-dir`
- Bug Finder Server:

  `polyspace-bug-finder-server -tmp-dir-in-results-dir`
- Code Prover Server:

  `polyspace-code-prover-server -tmp-dir-in-results-dir`

Tips

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

See Also

Topics
"Specify Polyspace Analysis Options"

-v | -version

Display Polyspace version number

Syntax

```
-v
-version
```

Description

`-v` or `-version` displays the version number of your Polyspace product.

Examples

Display the version number and release of your Polyspace product:

- Bug Finder:

  ```
  polyspace-bug-finder -v
  ```
- Code Prover:

  ```
  polyspace-code-prover -v
  ```
- Bug Finder Server:

  ```
  polyspace-bug-finder-server -v
  ```
- Code Prover Server:

  ```
  polyspace-code-prover-server -v
  ```

-verif-version

Assign a version identifier

Syntax

`-verif-version` *id*

Description

`-verif-version` *id* assigns an identifier, *id*, to identify the analysis. You can use this identifier to refer to different analyses at the command line. For example, you can import comments from a previous analysis using the identifier.

Examples

Assign a verification identifier:

- Bug Finder:

  `polyspace-bug-finder -verif-version 1.3`
- Code Prover:

  `polyspace-code-prover -verif-version 1.3`
- Bug Finder Server:

  `polyspace-bug-finder-server -verif-version 1.3`
- Code Prover Server:

  `polyspace-code-prover-server -verif-version 1.3`

Tips

This option is not useful for Polyspace as You Code.

See Also

Topics
"Specify Polyspace Analysis Options"

-xml-annotations-description

Apply custom code annotations to Polyspace analysis results

Syntax

`-xml-annotations-description` *file_path*

Description

`-xml-annotations-description` *file_path* uses the annotation syntax defined in the XML file located in *file_path* to interpret code annotations in your source files. You can use the XML file to specify an annotation syntax and map it to the Polyspace annotation syntax. When you run an analysis by using this option, you can justify and hide results with annotations that use your syntax. If you run Polyspace at the command line, *file_path* is the absolute path or path relative to the folder from which you run the command. If you run Polyspace through the user interface, *file_path* is the absolute path.

If you are running an analysis through the user interface, you can enter this option in the **Other** field, under the **Advanced Settings** node on the **Configuration** pane. See `Other`.

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

Why Use This Option

If you have existing annotations from previous code reviews, you can import these annotations to Polyspace. You do not have to review and justify results that you have already annotated. Similarly, if your code comments must adhere to a specific format, you can map and import that format to Polyspace.

Examples

Import Existing Annotations for Coding Rule Violations

Suppose that you have previously reviewed source file `zero_div.c` containing the following code, and justified certain MISRA C: 2012 violations by using custom annotations.

```
#include <stdio.h>

/* Violation of Misra C:2012
rules 8.4 and 8.7 on the next
line of code. */

int func(int p) //My_rule 50, 51
{
    int i;
    int j = 1;

    i = 1024 / (j - p);
    return i;
}

/* Violation of Misra C:2012
rule 8.4 on the next line of
code */

int func2(void){ //My_rule 50
    int x=func(2);
    return x;
}
```

The code comments **My_rule 50, 51** and **My_rule 50** do not use the Polyspace annotation syntax. Instead, you use a convention where you place all MISRA rules in a single numbered list. In this list, rules 8.4 and 8.7 correspond to the numbers 50 and 51.You can check this code for MISRA C: 2012 violations by typing the command:

- Bug Finder:

  `polyspace-bug-finder -sources source_path -misra3 all`
- Code Prover:

  `polyspace-code-prover -sources source_path -misra3 all -main-generator`
- Bug Finder Server:

  `polyspace-bug-finder-server -sources source_path -misra3 all`
- Code Prover Server:

  `polyspace-code-prover-server -sources source_path -misra3 all -main-generator`

source_path is the path to `zero_div.c`.

The annotated violations appear in the **Results List** pane. You must review and justify them again.

This XML example defines the annotation format used in `zero_div.c` and maps it to the Polyspace annotation syntax:

- The format of the annotation is the keyword `My_rule`, followed by a space and one or more comma-separated alphanumeric rule identifiers.

- Rule identifiers 50 and 51 are mapped to MISRA C: 2012 rules 8.4 and 8.7 respectively. The mapping uses the Polyspace annotation syntax.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
             Group="exampleCustomAnnotation">

  <Expressions Search_For_Keywords="My_rule"
               Separator_Result_Name="," >


    <!-- This section defines the annotation syntax format -->
    <Expression Mode="SAME_LINE"
                Regex="My_rule\s(\w+(\s*,\s*\w+)*)"
                Rule_Identifier_Position="1"
                />

  </Expressions>
  <!-- This section maps the user annotation to the Polyspace
   annotation syntax -->
  <Mapping>
  <Result_Name_Mapping  Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
  <Result_Name_Mapping  Rule_Identifier="51" Family="MISRA-C3" Result_Name="8.7"/>
  </Mapping>
</Annotations>
```
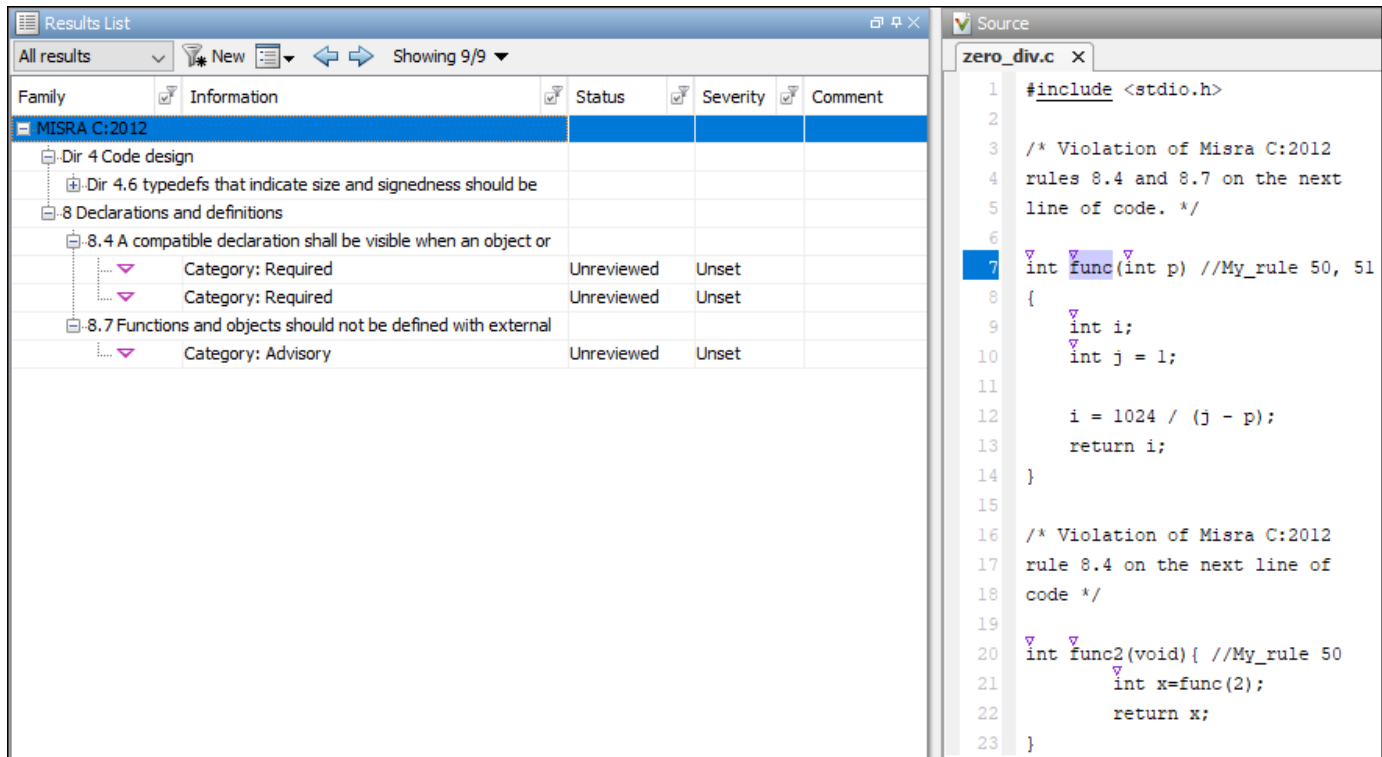
To import the existing annotations and apply them to the corresponding Polyspace results:

1 Copy the preceding code example to a text editor and save it on your machine as `annotations_description.xml`, for instance in `C:\Polyspace_workspace\annotations\`.

2 Rerun the analysis on `zero_div.c` by using the command:

- Bug Finder:

```
polyspace-bug-finder -sources source_path -misra3 all ^
-xml-annotations-desription ^
C:\Polyspace_workspace\annotations\annotations_description.xml
```

- Code Prover:

```
polyspace-code-prover -sources source_path -misra3 all ^
-main-generator -xml-annotations-desription ^
C:\Polyspace_workspace\annotations\annotations_description.xml
```

- Bug Finder Server:

```
polyspace-bug-finder-server -sources source_path -misra3 all ^
-xml-annotations-desription ^
C:\Polyspace_workspace\annotations\annotations_description.xml
```

- Code Prover Server:

```
polyspace-code-prover-server -sources source_path -misra3 all ^
-main-generator -xml-annotations-desription ^
C:\Polyspace_workspace\annotations\annotations_description.xml
```

Polyspace considers the annotated results justified and hides them in the **Results List** pane.
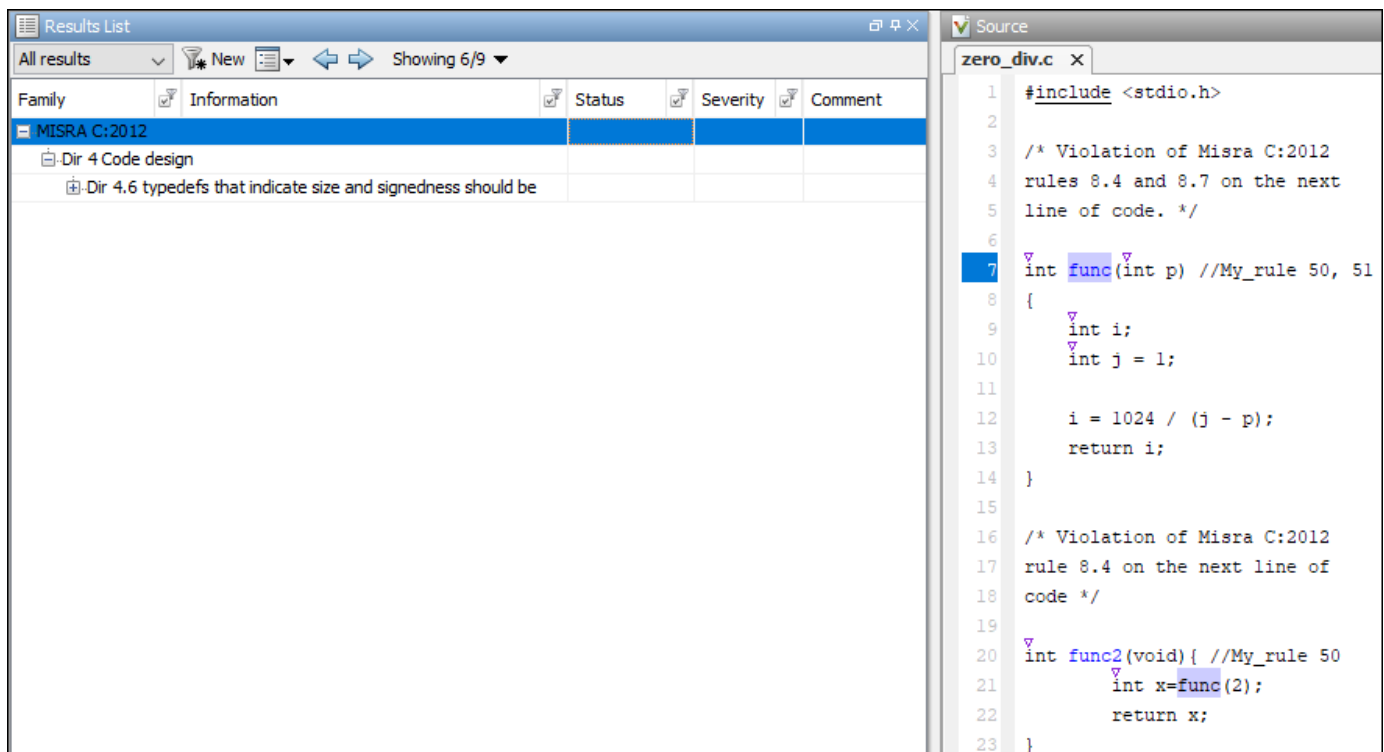
See Also

Topics
"Specify Polyspace Analysis Options"
"Define Custom Annotation Format"
"Annotation Description Full XML Template"
"Resolve -xml-annotations-description Errors"

Introduced in R2017b

Run-Time Checks

Absolute address usage

Absolute address is assigned to pointer

Description

This check appears when an absolute address is assigned to a pointer.

By default, this check is green. The software assumes the following about the absolute address:

- The address is valid.
- The type of the pointer to which you assign the address determines the initial value stored in the address.

  If you assign the address to an int* pointer, the memory zone that the address points to is initialized with an int value. The value can be anything allowed for the data type int.

To turn this check orange by default for each absolute address usage, use the command-line option -no-assumption-on-absolute-addresses.

Diagnosing This Check

"Review and Fix Absolute Address Usage Checks"

Examples

Reading content of absolute address

```
enum typeList {CHAR,INT,LONG};
enum typeList showType(void);
long int returnLong(void);

void main() {
    int *p = (int *)0x32; //Green absolute address usage
    enum typeList myType = showType();

    char x_char;
    int x_int;
    long int x_long;

    if(myType == CHAR)
        x_char = *p;
    else if(myType == INT)
        x_int = *p;
    else {
        x_long = *p;
        long int x2_long = returnLong();
    }
}
```

In this example, the option -no-assumption-on-absolute-addresses is not used. Therefore, the **Absolute address usage** check is green when the pointer p is assigned an absolute address.

Following this check, the verification assumes that the address is initialized with an `int` value. If you use `x86_64` for `Target processor type (-target)` (sizeof(char) < sizeof(int) < sizeof(long int)), the assumption results in the following:

- In the `if(myType == CHAR)` branch, an orange **Overflow** occurs because `x_char` cannot accommodate all values allowed for an `int` variable.

- In the `else if(myType == INT)` branch, if you place your cursor on `x_int` in your verification results, the tooltip shows that `x_int` potentially has all values allowed for an `int` variable.

- In the `else` branch, if you place your cursor on `x_long`, the tooltip shows that `x_long` potentially has all values allowed for an `int` variable. If you place your cursor on `x2_long`, the tooltip shows that `x2_long` potentially has all values allowed for a `long int` variable. The range of values that `x2_long` can take is wider than the values allowed for an `int` variable in the same target.

Arithmetic on pointers with absolute address

```
void main() {
    int *p = (int *)0x32;
    int x = *p;
    p++;
    x = *p;
}
```

In this example, the option `-no-assumption-on-absolute-addresses` is used. The **Absolute address usage** check is orange when the pointer `p` is assigned an absolute address.

Following this check:

- Polyspace considers that `p` points to a valid memory location. Therefore the **Illegally dereferenced pointer** check on the following line is green.

- In the next two lines, the pointer `p` is incremented and then dereferenced. In this case, an **Illegally dereferenced pointer** check appears on the dereference and not an **Absolute address usage** check even though `p` still points to an absolute address.

Check Information
Group: Static memory
Language: C | C++
Acronym: ABS_ADDR

Non-compliance with AUTOSAR specification

RTE API function is used with arguments that violate the AUTOSAR standard specification

Description

This check determines if the arguments to an RTE API function violate the AUTOSAR standard specifications.

For instance, checks on `Rte_Write_*` or `Rte_Byps_Write_*` function calls determine if the pointer-to-data argument in the call:

- Is NULL valued.
- Points to a memory buffer.
- Points to an initialized memory buffer.
- For buffers with enum values, values are within the enum range.

For more information on the RTE API specifications, see the AUTOSAR documentation.

To enable this check, use the value `autosar` for the option `Libraries used (-library)`.

This check finds a subset of issues found with the check `Invalid use of AUTOSAR runtime environment function`. Setting up for this check does not require providing the design constraints in ARXML format, therefore this check cannot find the constraint violations found with the other check. See also "Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace".

Diagnosing This Check

To diagnose this check, read the message on the **Result Details** pane. The message shows all checks performed on the RTE API function, along with information about whether the check passed. For instance, this message:

Shows the results of three checks. Only one of the checks indicates a possible issue. The first argument of the function might not point to initialized memory.

Investigate the root cause of the issue further.

Examples

Rte_Byps_Write_* Argument Pointing to Statically Allocated Noninitialized Buffer

```
#include <stdlib.h>
```

```
// Type declarations that are typically in AUTOSAR header Rte_type.h
typedef unsigned char uint8_T;
typedef unsigned int uint32_T;
typedef uint8_T Std_ReturnType;

typedef struct {
    uint8_T color;
    uint32_T number;
}
colorNumber;

extern Std_ReturnType Rte_Byps_Write_out_colorNumber_1(colorNumber*);

void SendData() {
    colorNumber aColor;
    uint8_T copyColor;
    uint32_T copyNumber;

    colorNumber* aPtrColor = &aColor;
    Rte_Byps_Write_out_colorNumber_1(aPtrColor);

    copyColor = aColor.color;
    copyNumber = aColor.number;
}
```

In this example, the function Rte_Byps_Write_out_colorNumber_1 takes a pointer to a non-initialized variable. The check on this function is red indicating a definite issue.

The check message states that:

*   The pointer cannot have a NULL value.
*   The pointer is allocated a buffer.
*   The poined buffer is not initialized.

Since one of the constituent checks, the third one, indicates a definite error, the check is red.

Rte_Byps_Write_* Argument Pointing to Dynamically Allocated Noninitialized Buffer

```
#include <stdlib.h>

// Type declarations that are typically in AUTOSAR header Rte_type.h
typedef unsigned char uint8_T;
typedef unsigned int uint32_T;
typedef uint8_T Std_ReturnType;

typedef struct {
    uint8_T color;
    uint32_T number;
}
colorNumber;
extern Std_ReturnType Rte_Byps_Write_out_colorNumber_2(colorNumber*);

void SendData() {
    colorNumber* arrayColorNumber = (colorNumber*) malloc(2*sizeof(colorNumber));
    uint8_T copyColor;
    uint32_T copyNumber;
```

```
        Rte_Byps_Write_out_colorNumber_2(arrayColorNumber);

        copyColor = arrayColorNumber[0].color;
        copyNumber = arrayColorNumber[0].number;
}
```

In this example, the function `Rte_Byps_Write_out_colorNumber_2` takes a pointer returned from a memory allocation with `malloc`. The check on this function is red indicating a definite issue.

The check message indicates that:

- The pointer might have a NULL value.
- If the pointer value is not NULL, the pointer is allocated a buffer.
- If the pointer value is not NULL and the pointer points to a buffer, the buffer is not initialized.

Since one of the constituent checks, the third one, indicates a definite error, the check is red.

Rte_Byps_Write_* Argument Pointing to Possibly Noninitialized Buffer

```
#include <stdlib.h>

// Type declarations that are typically in AUTOSAR header Rte_type.h
typedef unsigned char uint8_T;
typedef unsigned int uint32_T;
typedef uint8_T Std_ReturnType;

typedef struct {
    uint8_T color;
    uint32_T number;
}
colorNumber;
extern Std_ReturnType Rte_Byps_Write_out_colorNumber_2(colorNumber*);

void SendData(uint8_T hasInitialData, colorNumber* initialColorData) {
    colorNumber arrayColorNumber[2];
    uint8_T copyColor;
    uint32_T copyNumber;

    if(hasInitialData == 1) {
        for(uint8_T i = 0; i < 2; i++) {
            arrayColorNumber[i].color = initialColorData[i].color;
            arrayColorNumber[i].number = initialColorData[i].number;
        }
    }
    else if (hasInitialData == 0){
        for(uint8_T i = 0; i < 2; i++) {
            arrayColorNumber[i].color = 0;
            arrayColorNumber[i].number = 0;
        }
    }

    Rte_Byps_Write_out_colorNumber_2(arrayColorNumber);

}
```

In this example, the function `Rte_Byps_Write_out_colorNumber_2` takes a pointer to a possibly noninitialized buffer. The check on this function is orange, indicating a possible error, for instance, an

error that occurs only on certain execution paths. You can see that because of a missing catch-all `else` clause in the `if-else if-else` statement, the buffer `arrayColorNumber` is not initialized for values of `hasInitialData` other than 0 and 1.

The check message indicates that:

• The pointer cannot have a NULL value.

• The pointer is allocated a buffer.

• The buffer might not be initialized.

Since one of the constituent checks, the third one, indicates a possible error, the check is orange.

Check Information

Group: Other
Language: C | C++
Default: On if you use the value `autosar` for the option `Libraries used (-library)`, otherwise off
Command-Line Syntax: `autosar_compliance`

See Also

`Libraries used (-library)`

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"
"Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace"

Introduced in R2021a

Invalid result of AUTOSAR runnable implementation

Return value or output arguments violate AUTOSAR specifications

Description

This check evaluates functions implementing AUTOSAR runnables. The check determines if the output arguments and return value from the runnable can violate AUTOSAR specifications at run-time.

Using the information on the **Result Details** pane, determine whether the return value or an argument violates data constraints in the AUTOSAR XML specifications or can be NULL-valued. Look for the **!** icon that indicates a definite error or the **?** icon that indicates a possible error.

For each output argument and the return value, the check looks for these violations:

- *Data constraint violations*:

  Suppose, in this implementation of the runnable `foo`, the return value, which represents an application error, has an enumeration data type with a finite set of values. The analysis checks if the return value can acquire a value outside that set at run time.

  ```
  iOperations_ApplicationError foo(
      Rte_Instance const self,
      app_Array_2_n320to320ConstRef aInput,
      app_Array_2_n320to320Ref aOutput,
      app_Enum001Ref aOut2) {
  ...
  }
  ```

  The check can result in a message such as this. The message indicates that the argument has a value that falls outside the constrained range (in this case, the value 43).

  > **?** aReturn may not meet its specification.
  >
  > Specification: {24U,42U,0U,1U,64U,64U,128U,128U,129U,130U,131U,132U,133U,134U,135U,136U,137U,138U,139U,140U,141U,0U,1U}
  >
  > Actual value (const unsigned int 8): [0 .. 1] or 24 or 43

  In general, the analysis verifies if each output argument of the runnable and the return value stays within the constrained range allowed by their AUTOSAR data types. You limit values of AUTOSAR data types by referring to data constraints in your ARXML files.

- *NULL or unallocated pointers*:

  Suppose, in this implementation of the runnable `foo`, the first output argument `aOutput` is a pointer. The analysis checks if the pointer is non-NULL and allocated for all possible execution paths upon return from the runnable.

  ```
  iOperations_ApplicationError foo(
      Rte_Instance const self,
      app_Array_2_n320to320ConstRef aInput,
      app_Array_2_n320to320Ref aOutput,
      app_Enum001Ref aOut2) {
  ```

```
...
}
```

The check can result in a message such as this.

```
✓ aOutput meets its specification.
   Specification: non-NULL

✓ aOutput meets its specification.
   Specification: allocated
```
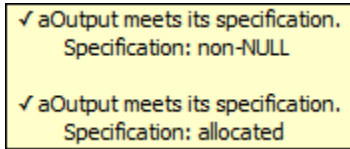
In general, the analysis verifies if a pointer output arguments from the runnable are non-NULL and allocated upon return from the runnable.

By default, the analysis assumes that pointer arguments to runnables and pointers returned from `Rte_` functions are not NULL. To change this assumption, undefine the macro `RTE_PTR2USERCODE_SAFE` using the option `-U` of the `polyspace-autosar` command.

See "Run Polyspace on AUTOSAR Code with Conservative Assumptions".

The check first considers the return from the runnable and then the output arguments. If the return from the runnable indicates an error, the check does not look at output arguments on execution paths with the error.

For instance, in this example, the return value is `RTE_E_OK` only if the output argument `aOut2` is not NULL. The check does not consider other execution paths (where the return value is not `RTE_E_OK`). Therefore, it determines that `aOut2` cannot be NULL.

```
// Runnable implementation
iOperations_ApplicationError foo(
    Rte_Instance const self,
    app_Array_2_n320to320ConstRef aInput,
    app_Array_2_n320to320Ref aOutput,
        app_Enum001Ref aOut2)
{
    iOperations_ApplicationError rc = E_NOT_OK;


    if (aOut2!=NULL_PTR)
    {
        // set invalid value will trigger STD_LIB RED in prove-runnable wrapper
        *aOut2 = 4;
        rc = RTE_E_OK;
    }
    return rc;
}
```

The reason for this behavior is the following: If the return from the runnable indicates an error status on a certain execution path, you can evaluate the error status and take corrective action. Run-time checks are not required for those paths. In certain situations, you might be using one or more output arguments to provide further information on an error status. You might want to check if those output argument can be NULL when the runnable completes execution. If you have this requirement, contact Technical Support.

The check does not flag these situations:

- Output arguments are not written at all within the body of the runnable (or not written along certain execution paths).
- The return value is not initialized within the body of the runnable (or not initialized along certain execution paths).

The analysis checks for conformance with data constraints only when the return value is initialized or output arguments written.

Result Information
Group: Other
Language: C
Acronym: AUTOSAR_IMPL

See Also
`Invalid use of AUTOSAR runtime environment function`

Topics
"Review Polyspace Results on AUTOSAR Code"
"Interpret Code Prover Results in Polyspace Desktop User Interface"

Introduced in R2018a

AUTOSAR runnable not implemented

Function implementing AUTOSAR runnable is not found

Description

This check determines if an AUTOSAR runnable specified in the ARXML specifications is implemented through a function in the source code. The check shows a result only if a function is not found.

You can navigate from the result to the runnable specification through the spec link.

Result Information
Group: Other
Language: C
Acronym: AUTOSAR_NOIMPL

See Also
`Invalid result of AUTOSAR runnable implementation`

Topics
"Review Polyspace Results on AUTOSAR Code"

Introduced in R2018a

Invalid use of AUTOSAR runtime environment function

RTE function argument violates AUTOSAR specifications

Description

This check evaluates calls to functions provided by the AUTOSAR Run-Time Environment (`Rte_` functions). The check determines if the function arguments can violate AUTOSAR XML specifications at run-time.

Using the information on the **Result Details** pane, determine whether an argument violates data constraints in the AUTOSAR XML specifications or can be NULL-valued. Look for the **!** icon that indicates a definite error or the **?** icon that indicates a possible error.

For each function argument, the check looks for these violations:

- *Data constraint violations*:

  Suppose, in this call to `Rte_IWrite_step_out_e4`, the second argument points to a data type that must obey a data constraint. The analysis checks if the constraint can be violated at run time.

  `Rte_IWrite_step_out_e4(self, arg);`

  The check can result in a message such as this. The message indicates that the argument has a value that falls outside the constrained range (in this case, the value 321).

  In general, the analysis verifies if each `Rte_` function argument stays within the constrained range allowed by its AUTOSAR data type. You limit values of AUTOSAR data types by referring to data constraints in your ARXML files. For instance, a constraint specification can look like this (AUTOSAR XML schema version 4.0).

```
<DATA-CONSTR>
    <SHORT-NAME>n320to320</SHORT-NAME>
      <DATA-CONSTR-RULES>
        <DATA-CONSTR-RULE>
          <PHYS-CONSTRS>
            <LOWER-LIMIT INTERVAL-TYPE="CLOSED">-320</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE="CLOSED">320</UPPER-LIMIT>
            <UNIT-REF DEST="UNIT">/jyb/types/units/NoUnit</UNIT-REF>
          </PHYS-CONSTRS>
        </DATA-CONSTR-RULE>
      </DATA-CONSTR-RULES>
</DATA-CONSTR>
...
<APPLICATION-PRIMITIVE-DATA-TYPE>
        <SHORT-NAME>Int_n320to320</SHORT-NAME>
        <CATEGORY>VALUE</CATEGORY>
        <SW-DATA-DEF-PROPS>
```

```
        <SW-DATA-DEF-PROPS-VARIANTS>
            <SW-DATA-DEF-PROPS-CONDITIONAL>
                ...
                <DATA-CONSTR-REF DEST="DATA-CONSTR">types/app/constraints/n320to320
                </DATA-CONSTR-REF>
                ...
            </SW-DATA-DEF-PROPS-CONDITIONAL>
        </SW-DATA-DEF-PROPS-VARIANTS>
    </SW-DATA-DEF-PROPS>
</APPLICATION-PRIMITIVE-DATA-TYPE>
```

- *Violations of AUTOSAR standard specifications such as passing unallocated pointers as input arguments to some RTE functions*:

  Suppose, in this call to `Rte_IWrite_step_out_e4`, the second argument is a pointer. The analysis checks if the pointer is non-NULL and allocated for all possible execution paths.

  `Rte_IWrite_step_out_e4(self,arg);`

  The check can result in a message such as this.
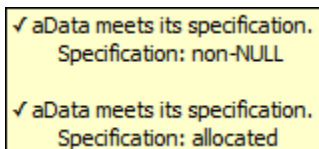
  In general, the analysis verifies if a pointer argument to an `Rte_` function is non-NULL and allocated.

  These violations are also found with the checker `Non-compliance with AUTOSAR specification`.

Result Information
Group: Other
Language: C
Acronym: AUTOSAR_USE

See Also
`Invalid result of AUTOSAR runnable implementation` | `Non-compliance with AUTOSAR specification`

Topics
"Review Polyspace Results on AUTOSAR Code"
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"
"Choose Between Component-Based and Integration Analysis of AUTOSAR Code with Polyspace"

Introduced in R2018a

Correctness condition

Mismatch occurs during pointer cast or function pointer use

Description

This check determines whether:

- An array is mapped to a larger array through a pointer cast
- A function pointer points to a function with a valid prototype
- A global variable falls outside the range specified through the **Global Assert** mode. See also "Constrain Global Variable Range".

Diagnosing This Check

"Review and Fix Correctness Condition Checks"

Examples

Array is mapped to larger array

```
typedef int smallArray[10];
typedef int largeArray[100];


void main(void) {
    largeArray myLargeArray;
    smallArray *smallArrayPtr = (smallArray*) &myLargeArray;
    largeArray *largeArrayPtr = (largeArray*) smallArrayPtr;
}
```

In this example:

- In the first pointer cast, a pointer of type `largeArray` is cast to a pointer of type `smallArray`. Because the data type `smallArray` represents a smaller array, the **Correctness condition** check is green.

- In the second pointer cast, a pointer of type `smallArray` is cast to a pointer of type `largeArray`. Because the data type `largeArray` represents a larger array, the **Correctness condition** check is red.

Function pointer does not point to function

```
typedef void (*callBack) (float data);
typedef struct {
    char funcName[20];
    callBack func;
} funcStruct;

funcStruct myFuncStruct;
```

```
void main(void) {
    float val = 0.f;
    myFuncStruct.func(val);
}
```

In this example, the global variable `myFuncStruct` is not initialized, so the function pointer `myFuncStruct.func` contains NULL. When the pointer `myFuncStruct.func` is dereferenced, the **Correctness condition** check produces a red error.

Function pointer points to function through absolute address usage

```
#define MAX_MEMSEG 32764
typedef void (*ptrFunc)(int memseg);
ptrFunc operation = (ptrFunc)(0x003c);

void main(void) {
    for (int i=1; i <= MAX_MEMSEG; i++)
        operation(i);
}
```

In this example, the function pointer `operation` is cast to the contents of a memory location. Polyspace cannot determine whether the location contains a variable or a function code and whether the function is well-typed. Therefore, when the pointer `operation` is dereferenced and used in a function call, the **Correctness condition** check is orange.

After an orange **Correctness condition** check due to absolute address usage, the software assumes that the following variables have the full range of values allowed by their type:

- Variable storing the return value from the function call.

  In the following example, the software assumes that the return value of `operation` is full-range.

  ```
  typedef int (*ptrFunc)(int);
  ptrFunc operation = (ptrFunc)(0x003c);

  int main(void) {
    return operation(0);
  }
  ```

- Variables that can be modified through the function arguments.

  In the following example, the function pointer `operation` takes a pointer argument `ptr` that points to a variable `var`. After the call to `operation`, the software assumes that `var` has full-range value.

  ```
  typedef void (*ptrFunc)(int*);
  ptrFunc operation = (ptrFunc)(0x003c);

  void main(void) {
    int var;
    int *ptr=&var;
    operation(ptr);
  }
  ```

Function pointer points to function with wrong argument type

```
typedef struct {
  double real;
```

```
  double imag;
} complex;

typedef int (*typeFuncPtr) (complex*);

int func(int* x);

void main() {
  typeFuncPtr funcPtr = (typeFuncPtr)&func;
  int arg = 0, result = funcPtr((complex*)&arg);
}
```

In this example, the function pointer `funcPtr` points to a function with argument type `complex*`. However, the pointer is assigned the address of function `func` whose argument type is `int*`. Because of this type mismatch, the **Correctness condition** check is orange.

Function pointer points to function with wrong number of arguments

```
typedef int (*typeFuncPtr) (int, int);

int func(int);

void main() {
    typeFuncPtr funcPtr = (typeFuncPtr)&func;
    int arg1 = 0, arg2 = 0, result = funcPtr(arg1,arg2);
}
```

In this example, the function pointer `funcPtr` points to a function with two `int` arguments. However, it is assigned the function `func` which has one `int` argument only. Because of this mismatch in number of arguments, the **Correctness condition** check is orange.

Function pointer points to function with wrong return type

```
typedef double (*typeFuncPtr) (int);

int func(int);

void main() {
    typeFuncPtr funcPtr = (typeFuncPtr)&func;
    int arg = 0;
    double result = funcPtr(arg);
}
```

In this example, the function pointer `funcPtr` points to a function with return type `double`. However, it is assigned the function `func` whose return type is `int`. Because of this mismatch in return types, the **Correctness condition** check is orange.

Variable falls outside Global Assert range

```
int glob = 0;
int func();

void main() {
    glob = 5;
    glob = func();
    glob+= 20;
}
```

In this example, a range of `0..10` was specified for the global variable `glob`.

- In the statement `glob=5;`, a green **Correctness condition** check appears on `glob`.
- In the statement `glob=func();`, an orange **Correctness condition** check appears on `glob` because the return value of stubbed function `func` can be outside `0..10`.

  After this statement, Polyspace considers that `glob` has values in `0..10`.

- In the statement `glob+=20;`, a red **Correctness condition** check appears on `glob` because after the addition, `glob` has values in `20..30`.

See also "Constrain Global Variable Range".

Check Information

Group: Other
Language: C | C++
Acronym: COR

See Also

`Constraint setup (-data-range-specifications)|Permissive function pointer calls (-permissive-function-pointer)`

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"
"Constrain Global Variable Range"

Division by zero

Division by zero occurs

Description

This check determines whether the right operand of a division or modulus operation is zero.

Diagnosing This Check

"Review and Fix Division by Zero Checks"

Examples

Red integer division by zero

```
#include <stdio.h>

void main() {
    int x=2;
    printf("Quotient=%d",100/(x-2));
}
```

In this example, the denominator x-2 is zero.

Correction — Check for zero denominator

One possible correction is to check for a zero denominator before division.

In a complex code, it is difficult to keep track of values and avoid zero denominators. Therefore, it is good practice to check for zero denominator before every division.

```
#include <stdio.h>
int input();
void main() {
    int x=input();
    if(x>0) { //Avoid overflow
        if(x!=2 && x>0)
            printf("Quotient=%d",100/(x-2));
        else
            printf("Zero denominator.");
    }
}
```

Red integer division by zero after for loop

```
#include <stdio.h>
void main() {
    int x=-10;
    for (int i=0; i<10; i++)
        x+=3;
    printf("Quotient=%d",100/(x-20));
}
```

In this example, the denominator `x-20` is zero.

Correction — Check for zero denominator

One possible correction is to check for a zero denominator before division.

After several iterations of a `for` loop, it is difficult to keep track of values and avoid zero denominators. Therefore, it is good practice to check for zero denominator before every division.

```c
#include <stdio.h>
#define MAX 10000
int input();

void main() {
    int x=input();
    for (int i=0; i<10; i++) {
        if(x < MAX) //Avoid overflow
            x+=3;
    }

    if(x>0) { //Avoid overflow
        if(x!=20)
            printf("Quotient=%d",100/(x-20));
        else
            printf("Zero denominator.");
    }
}
```

Orange integer division by zero inside `for` loop

```c
#include<stdio.h>

void main() {
    printf("Sequence of ratios: \n");
    for(int count=-100; count<=100; count++)
        printf(" %.2f ", 1/count);
}
```

In this example, `count` runs from -100 to 100 through zero. When `count` is zero, the **Division by zero** check returns a red error. Because the check returns green in the other `for` loop runs, the `/` symbol is orange.

There is also a red **Non-terminating loop** error on the `for` loop. This red error indicates a definite error in one of the loop runs.

Correction — Check for zero denominator

One possible correction is to check for a zero denominator before division.

```c
#include<stdio.h>

void main() {
    printf("Sequence of ratios: \n");
    for(int count=-100; count<=100; count++) {
        if(count != 0)
            printf(" %.2f ", 1/count);
        else
            printf(" Infinite ");
```

```
        }
    }
```

Orange float division by zero inside `for` loop

```
#include <stdio.h>
#include <math.h>

#define stepSize 0.1

void main() {
    float divisor = -1.0;
    int numberOfSteps = (int)((2.0*1.0)/stepSize);

    printf("Divisor running from -1.0 to 1.0\n");
    for(int count = 1; count <= numberOfSteps; count++) {
        divisor+= stepSize;
    divisor = ceil(divisor * 10.) / 10.; // one digit of imprecision
        printf(" .2f ", 1.0/divisor);
    }
}
```

In this example, `divisor` runs from –1.0 to 1.0 through 0.0. When `divisor` is 0.0, the **Division by zero** check returns a red error. Because the check returns green in the other `for` loop runs, the `/` symbol is orange.

There is no red **Non-terminating loop** error on the `for` loop. The red error does not appear because Polyspace approximates the values of `divisor` by a broader range. Therefore, Polyspace cannot determine if there is a definite error in one of the loop runs.

Correction — Check for zero denominator

One possible correction is to check for a zero denominator before division. For `float` variables, do not check if the denominator is exactly zero. Instead, check whether the denominator is in a narrow range around zero.

```
#include <stdio.h>
#include <math.h>

#define stepSize 0.1

void main() {
    float divisor = -1.0;
    int numberOfSteps = (int)((2*1.0)/stepSize);

    printf("Divisor running from -1.0 to 1.0\n");;
    for(int count = 1; count <= numberOfSteps; count++) {
        divisor += stepSize;
    divisor = ceil(divisor * 10.) / 10.; // one digit of imprecision
        if(divisor < -0.00001 || divisor > 0.00001)
            printf(" .2f ", 1.0/divisor);
        else
            printf(" Infinite ");
    }
}
```

Check Information

Group: Numerical
Language: C | C++
Acronym: ZDV

See Also

`Consider non finite floats (-allow-non-finite-floats)`

Topics

"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Function not called

Function is defined but not called

Description

This check on a function definition determines if the function is called anywhere in the code. This check is disabled if your code does not contain a `main` function.

Use this check to satisfy ISO 26262 requirements about function coverage. For example, see table 15 of ISO 26262, part 6.

Note This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Diagnosing This Check

"Review and Fix Function Not Called Checks"

Examples

Function not called

```
#define max 100
int var;
int getValue(void);
int getSaturation(void);

void reset() {
    var=0;
}

void main() {
    int saturation = getSaturation(),val;
    for(int index=1; index<=max; index++) {
        val = getValue();
        if(val>0 && val<10)
            var += val;
        if(var > saturation)
            var=0;
    }
}
```

In this example, the function `reset` is defined but not called. Therefore, a gray **Function not called** check appears on the definition of `reset`.

Correction: Call Function

One possible correction is to call the function `reset`. In this example, the function call `reset` serves the same purpose as instruction `var=0;`. Therefore, replace the instruction with the function call.

```
#define max 100

int var;
int getValue(void);
int getSaturation(void);

void reset() {
    var=0;
}

void main() {
    int saturation = getSaturation(),val;
    for(int index=1; index<=max; index++) {
        val = getValue();
        if(val>0 && val<10)
            var += val;
        if(var > saturation)
            reset();
    }
}
```

Function Called from Another Uncalled Function

```
#define max 100

int var;
int numberOfResets;
int getValue();
int getSaturation();

void updateCounter() {
  numberOfResets++;
}

void reset() {
  updateCounter();
  var=0;
}

void main() {
  int saturation = getSaturation(),val;
  for(int index=1; index<=max; index++) {
    val = getValue();
    if(val>0 && val<10)
      var += val;
    if(var > saturation) {
      numberOfResets++;
      var=0;
    }
  }
}
```

In this example, the function reset is defined but not called. Since the function updateCounter is called only from reset, a gray **Function not called** error appears on the definition of updateCounter.

Correction: Call Function

One possible correction is to call the function `reset`. In this example, the function call `reset` serves the same purpose as the instructions in the branch of `if(var > saturation)`. Therefore, replace the instructions with the function call.

```
#define max 100

int var;
int numberOfResets;
int getValue(void);
int getSaturation(void);

void updateCounter() {
  numberOfResets++;
}

void reset() {
  updateCounter();
  var=0;
}

void main() {
  int saturation = getSaturation(),val;
  for(int index=1; index<=max; index++) {
    val = getValue();
    if(val>0 && val<10)
      var += val;
    if(var > saturation)
      reset();
  }
}
```

Check Information

Group: Data flow
Language: C | C++
Acronym: FNC

See Also

`Detect uncalled functions (-uncalled-function-checks)`|`Function not reachable`

Topics
"Reasons for Unchecked Code"

Function not reachable

Function is called from unreachable part of code

Description

This check appears on a function definition. The check appears gray if the function is called only from an unreachable part of the code. The unreachable code can occur in one of the following ways:

- The code is reached through a condition that is always false.
- The code follows a `break` or `return` statement.
- The code follows a red check.

If your code does not contain a `main` function, this check is disabled

___

Note This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

___

Diagnosing This Check

"Review and Fix Function Not Reachable Checks"

Examples

Function Call from Unreachable Branch of Condition

```c
#include<stdio.h>
#define SIZE 100

void increase(int* arr, int index);

void printError()
{
  printf("Array index exceeds array size.");
}

void main() {
  int arr[SIZE],i;
  for(i=0; i<SIZE; i++)
    arr[i]=0;

  for(i=0; i<SIZE; i++) {
    if(i<SIZE)
      increase(arr,i);
    else
      printError();
  }
}
```

In this example, in the second `for` loop in `main`, `i` is always less than `SIZE`. Therefore, the `else` branch of the condition `if(i<SIZE)` is never reached. Because the function `printError` is called from the `else` branch alone, there is a gray **Function not reachable** check on the definition of `printError`.

Function Call Following Red Error

```
#include<stdio.h>

int getNum(void);

void printSuccess()
{
  printf("The operation is complete.");
}

void main() {
  int num=getNum(), den=0;
  printf("The ratio is %.2f", num/den);
  printSuccess();
}
```

In this example, the function `printSucess` is called following a red **Division by Zero** error. Therefore, there is a gray **Function not reachable** check on the definition of `printSuccess`.

Function Call from Another Unreachable Function

```
#include<stdio.h>
#define MAX 1000
#define MIN 0

int getNum(void);

void checkUpperBound(double ratio)
{
    if(ratio < MAX)
        printf("The ratio is within bounds.");
}

void checkLowerBound(double ratio)
{
    if(ratio > MIN)
        printf("The ratio is within bounds.");
}

void checkRatio(double ratio)
{
    checkUpperBound(ratio);
    checkLowerBound(ratio);
}

void main() {
    int num=getNum(), den=0;
    double ratio;
    ratio=num/den;
```

```
        checkRatio(ratio);
}
```

In this example, the function `checkRatio` follows a red **Division by Zero** error. Therefore, there is a gray **Function not reachable** error on the definition of `checkRatio`. Because `checkUpperBound` and `checkLowerBound` are called only from `checkRatio`, there is also a gray **Function not reachable** check on their definitions.

Function Call from Unreachable Code Using Function Pointer

```
#include<stdio.h>

int getNum(void);
int getChoice(void);

int num, den, choice;
double ratio;

void display(void)
{
    printf("Numerator = %d, Denominator = %d", num, den);
}

void display2(void)
{
    printf("Ratio = %.2f",ratio);
}


void main() {
    void (*fptr)(void);

    choice = getChoice();
    if(choice == 0)
        fptr = &display;
    else
        fptr = &display2;

    num = getNum();
    den = 0;
    ratio = num/den;

    (*fptr)();
}
```

In this example, depending on the value of `choice`, the function pointer `fptr` can point to either `display` or to `display2`. The call through `fptr` follows a red **Division by Zero** error. Because `display` and `display2` are called only through `fptr`, a gray **Function not reachable** check appears on their definitions.

Check Information
Group: Data flow
Language: C | C++
Acronym: FNR

See Also

`Detect uncalled functions (-uncalled-function-checks)` | `Function not called` | `Unreachable code`

Topics
"Reasons for Unchecked Code"

Function not returning value

C++ function does not return value when expected

Description

This check determines whether a function with a return type other than `void` returns a value. This check appears on the function definition.

Diagnosing This Check

"Review and Fix Function Not Returning Value Checks"

Examples

Function does not return value for any input

```
#include <stdio.h>
int input();
int inputRep();

int reply(int msg) {
  int rep = inputRep();
  if (msg > 0) return rep;
}

void main(void) {
  int ch = input(), ans;
  if (ch<=0)
    ans = reply(ch);
  printf("The answer is %d.",ans);
}
```

In this example, for all values of ch, `reply(ch)` has no return value. Therefore the **Function not returning value** check returns a red error on the definition of `reply()`.

Correction — Return value for all inputs

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>
int input();
int inputRep();

int reply(int msg) {
  int rep = inputRep();
  if (msg > 0) return rep;
  return 0;
}

void main(void) {
  int ch = input(), ans;
  if (ch<=0)
```

```
    ans = reply(ch);
  printf("The answer is %d.",ans);
}
```

Function does not return value for some inputs

```
#include <stdio.h>
int input();
int inputRep(int);

int reply(int msg) {
  int rep = inputRep(msg);
  if (msg > 0) return rep;
}

void main(void) {
  int ch = input(), ans;
  if (ch<10)
    ans = reply(ch);
  else
    ans = reply(10);
  printf("The answer is %d.",ans);
}
```

In this example, in the first branch of the `if` statement, the value of `ch` can be divided into two ranges:

- `ch < = 0`: For the function call `reply(ch)`, there is no return value.
- `ch > 0` and `ch < 10`: For the function call `reply(ch)`, there is a return value.

Therefore the **Function not returning value** check returns an orange error on the definition of `reply()`.

Correction — Return value for all inputs

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>
int input();
int inputRep(int);

int reply(int msg) {
  int rep = inputRep(msg);
  if (msg > 0) return rep;
  return 0;
}

void main(void) {
  int ch = input(), ans;
  if (ch<10)
    ans = reply(ch);
  else
    ans = reply(10);
  printf("The answer is %d.",ans);
}
```

Check Information

Group: C++
Language: C++
Acronym: FRV

See Also

`Return value not initialized`

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"

Global variable not assigned a value in initialization code

Global variable is not assigned a value in the initialization section of program

Description

This check determines if all non-const global variables (and local static variables) are explicitly assigned a value at declaration or in the section of code designated as initialization code.

To indicate the end of initialization code, you enter the line

```
#pragma polyspace_end_of_init
```

in the `main` function. The initialization code starts from the beginning of `main` and continues up to this pragma. To enable this check, use the option `Check that global variables are initialized after warm reboot (-check-globals-init)`.

The check on a global variable is:

- Red, if the variable is not initialized at all, either explicitly at declaration or in the initialization code (or is initialized in dead code within the initialization code).
- Orange, if the variable is not initialized on certain execution paths through the initialization code. For instance, the variable is initialized in an `if` branch of a conditional statement but not the `else` branch.
- Green, if the variable is always initialized once the initialization code completes execution.

In a warm reboot, to save time, the data segment of a program, which might hold variable values from a previous state, is not loaded. Instead, the program is supposed to explicitly initialize all non-const variables before execution. This check verifies that all non-const global variables are indeed initialized in a warm reboot.

Diagnosing This Check

Browse through all instances of the uninitialized or possibly uninitialized variable on the **Variable Access** pane (or the **Global Variables** pane in the Polyspace Access web interface). See if any of the references occur before the pragma `polyspace_end_of_init` is encountered.

See also "Variable Access".

Examples

Global Variable Not Initialized in Initialization Code

```
int aVar;
const int aConst = -1;
int anotherVar;

int main() {
```

```
        aVar = aConst;
#pragma polyspace_end_of_init
        return 0;
}
```

In this example, the global variable `aVar` is initialized in the initialization code section but the variable `anotherVar` is not.

Global Variable Not Initialized on Specific Paths Through Initialization Code

```
int var;

int checkSomething(void);
int checkSomethingElse(void);

int main() {
    int local_var;
    if(checkSomething())
    {
        var=0;
    }
    else if(checkSomethingElse()) {
        var=1;
    }
    #pragma polyspace_end_of_init
    var=2;
    local_var = var;
    return 0;
}
```

The check on `var` is orange because `var` might remain uninitialized when the `if` and `else if` statements are skipped.

Global Variable Appears Initialized Because of Read Accesses in Initialization Code

```
int aVar;
int anotherVar;

int checkSomething();

init0() {
  if (checkSomething())
      aVar = 0;
}
init1() {
  anotherVar = aVar; //Orange check: Non-initialized variable
}
main() {
  init0();
  init1();
#pragma polyspace_end_of_init
}
```

In this example, both variables `aVar` and `anotherVar` appear initialized (green check). However, the following path leads to both variables being non-initialized:

- The `if` statement in `init0` is skipped, leading to `aVar` being non-initialized.
- If `aVar` is non-initialized, `anotherVar` is also non-initialized (initialized with unpredictable values).

The issue is highlighted by a different check, `Non-initialized variable`. The check is orange on this line:

```
anotherVar = aVar;
```

Following the orange check, the execution path where `aVar` is non-initialized is removed from consideration. This removal leads to `anotherVar` appearing as initialized (green) according to all checks and `aVar` appearing as initialized (green) according to the check **Global variable not assigned a value in initialization code**.

To avoid misleading interpretation of green results for initialization:

- Verify the initialization code only using the options `Check that global variables are initialized after warm reboot (-check-globals-init)` and `Verify initialization section of code only (-init-only-mode)`.
- Make sure that there are no orange results for *both these checks*:

    - **Global variable not assigned a value in initialization code**
    - **Non-initialized variable**

Check Information
Group: Data flow
Language: C
Acronym: GLOBAL_SET_AT_INITIALIZATION

See Also
`Check that global variables are initialized after warm reboot (-check-globals-init)|Verify initialization section of code only (-init-only-mode)`

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Assumptions About Global Variable Initialization" on page 4-15

Illegally dereferenced pointer

Pointer is dereferenced outside bounds

Description

This check on a pointer dereference determines whether the pointer is NULL or points outside its bounds. The check occurs only when you dereference a pointer and not when you reassign to another pointer or pass the pointer to a function.

The check message shows you the pointer offset and buffer size in bytes. A pointer points outside its bounds when the sum of the offset and pointer size exceeds the buffer size.

- *Buffer*: When you assign an address to a pointer, a block of memory is allocated to the pointer. You cannot access memory beyond that block using the pointer. The size of this block is the buffer size.

  Sometimes, instead of a definite value, the size can be a range. For instance, if you create a buffer dynamically using `malloc` with an unknown input for the size, Polyspace assumes that the array size can take the full range of values allowed by the input data type.

- *Offset*: You can move a pointer within the allowed memory block by using pointer arithmetic. The difference between the initial location of the pointer and its current location is the offset.

  Sometimes, instead of a definite value, the offset can be a range. For instance, if you access an array in a loop, the offset changes value in each loop iteration and takes a range of values throughout the loop.

For instance, if the pointer points to an array:

- The buffer size is the array size.
- The offset is the difference between the beginning of the array and the current location of the pointer.

Diagnosing This Check

"Review and Fix Illegally Dereferenced Pointer Checks"

Examples

Pointer points outside array bounds

```
#define Size 1024

int input(void);

void main() {
    int arr[Size];
    int *p = arr;

    for (int index = 0; index < Size ; index++, p++){
        *p = input();
    }
```

```
    *p = input();
}
```

In this example:

- Before the `for` loop, `p` points to the beginning of the array `arr`.
- After the `for` loop, `p` points outside the array.

The **Illegally dereferenced pointer** check on dereference of `p` after the `for` loop produces a red error.

Correction — Remove illegal dereference

One possible correction is to remove the illegal dereference of `p` after the `for` loop.

```
#define Size 1024

int input(void);

void main() {
    int arr[Size];
    int *p = arr;

    for (int index = 0; index < Size ; index++, p++) {
        *p = input();
    }
}
```

Pointer points outside structure field

```
typedef struct S {
    int f1;
    int f2;
    int f3;
} S;

void Initialize(int *ptr) {
    *ptr = 0;
    *(ptr+1) = 0;
    *(ptr+2) = 0;
}

void main(void) {
    S myStruct;
    Initialize(&myStruct.f1);
}
```

In this example, in the body of `Initialize`, `ptr` is an `int` pointer that points to the first field of the structure. When you attempt to access the second field through `ptr`, the **Illegally dereferenced pointer** check produces a red error.

Correction — Avoid memory access outside structure field

One possible correction is to pass a pointer to the entire structure to `Initialize`.

```
typedef struct S {
    int f1;
    int f2;
```

```
    int f3;
} S;

void Initialize(S* ptr) {
    ptr->f1 = 0;
    ptr->f2 = 0;
    ptr->f3 = 0;
}

void main(void) {
    S myStruct;
    Initialize(&myStruct);
}
```

NULL pointer is dereferenced

```
#include<stdlib.h>

void main() {
    int *ptr=NULL;
    *ptr=0;
}
```

In this example, ptr is assigned the value NULL. Therefore when you dereference ptr, the **Illegally dereferenced pointer** check produces a red error.

Correction — Avoid NULL pointer dereference

One possible correction is to initialize ptr with the address of a variable instead of NULL.

```
void main() {
    int var;
    int *ptr=&var;
    *ptr=0;
}
```

Offset on NULL pointer

```
int getOffset(void);

void main() {
    int *ptr = (int*) 0 + getOffset();
    if(ptr != (int*)0)
        *ptr = 0;
}
```

In this example, although an offset is added to (int*) 0, Polyspace does not treat the result as a valid address. Therefore when you dereference ptr, the **Illegally dereferenced pointer** check produces a red error.

Bit field type is incorrect

```
struct flagCollection {
    unsigned int flag1: 1;
    unsigned int flag2: 1;
    unsigned int flag3: 1;
    unsigned int flag4: 1;
```

```
    unsigned int flag5: 1;
    unsigned int flag6: 1;
    unsigned int flag7: 1;
};

char getFlag(void);

int main()
{
    unsigned char myFlag = getFlag();
    struct flagCollection* myFlagCollection;
    myFlagCollection = (struct flagCollection *) &myFlag;
    if (myFlagCollection->flag1 == 1)
        return 1;
    return 0;
}
```

In this example:

- The fields of `flagCollection` have type `unsigned int`. Therefore, a `flagCollection` structure requires 32 bits of memory in a 32-bit architecture even though the fields themselves occupy 7 bits.

- When you cast a `char` address &myFlag to a `flagCollection` pointer `myFlagCollection`, you assign only 8 bits of memory to the pointer. Therefore, the **Illegally dereferenced pointer** check on dereference of `myFlagCollection` produces a red error.

Correction — Use correct type for bit fields

One possible correction is to use `unsigned char` as field type of `flagCollection` instead of `unsigned int`. In this case:

- The structure `flagCollection` requires 8 bits of memory.

- When you cast the `char` address &myFlag to the `flagCollection` pointer `myFlagCollection`, you also assign 8 bits of memory to the pointer. Therefore, the **Illegally dereferenced pointer** check on dereference of `myFlagCollection` is green.

```
struct flagCollection {
    unsigned char flag1: 1;
    unsigned char flag2: 1;
    unsigned char flag3: 1;
    unsigned char flag4: 1;
    unsigned char flag5: 1;
    unsigned char flag6: 1;
    unsigned char flag7: 1;
};

char getFlag(void);

int main()
{
    unsigned char myFlag = getFlag();
    struct flagCollection* myFlagCollection;
    myFlagCollection = (struct flagCollection *) &myFlag;
    if (myFlagCollection->flag1 == 1)
        return 1;
```

```
    return 0;
}
```

Return value of `malloc` is not checked for NULL

```
#include <stdlib.h>

void main(void)
{
    char *p = (char*)malloc(1);
    char *q = p;
    *q = 'a';
}
```

In this example, `malloc` can return NULL to `p`. Therefore, when you assign `p` to `q` and dereference `q`, the **Illegally dereferenced pointer** check produces a red error.

Correction — Check return value of `malloc` for NULL

One possible correction is to check `p` for NULL before dereferencing `q`.

```
#include <stdlib.h>
void main(void)
{
    char *p = (char*)malloc(1);
    char *q = p;
    if(p!=NULL) *q = 'a';
}
```

Pointer to union gets insufficient memory from `malloc`

```
#include <stdlib.h>

enum typeName {CHAR,INT};

typedef struct {
    enum typeName myTypeName;
    union {
        char myChar;
        int myInt;
    } myVar;
} myType;

void main() {
    myType* myTypePtr;
    myTypePtr = (myType*)malloc(sizeof(int) + sizeof(char));
    if(myTypePtr != NULL) {
        myTypePtr->myTypeName = INT;
    }
}
```

In this example:

- Because the union `myVar` has an `int` variable as a field, it must be assigned 4 bytes in a 32-bit architecture. Therefore, the structure `myType` must be assigned 4+4 = 8 bytes.

- `malloc` returns `sizeof(int) + sizeof(char)`=4+1=5 bytes of memory to `myTypePtr`, a pointer to a `myType` structure. Therefore, when you dereference `myTypePtr`, the **Illegally dereferenced pointer** check returns a red error.

Correction — Assign sufficient memory to pointer

One possible correction is to assign 8 bytes of memory to `myTypePtr` before dereference.

```
#include <stdlib.h>

enum typeName {CHAR,INT};

typedef struct {
    enum typeName myTypeName;
    union {
        char myChar;
        int myInt;
    } myVar;
} myType;

void main() {
    myType* myTypePtr;
    myTypePtr = (myType*)malloc(sizeof(int) + sizeof(int));
    if(myTypePtr != NULL) {
        myTypePtr->myTypeName = INT;
    }
}
```

Structure is allocated memory partially

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} rectangle;

typedef struct {
    int length;
    int breadth;
    int height;
} cuboid;

void main() {
    cuboid *cuboidPtr = (cuboid*)malloc(sizeof(rectangle));
    if(cuboidPtr!=NULL) {
        cuboidPtr->length = 10;
        cuboidPtr->breadth = 10;
    }
}
```

In this example, `cuboidPtr` obtains sufficient memory to accommodate two of its fields. Because the ANSI C standards do not allow such partial memory allocations, the **Illegally dereferenced pointer** check on the dereference of `cuboidPtr` produces a red error.

Correction — Allocate full memory

To observe ANSI C standards, `cuboidPtr` must be allocated full memory.

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} rectangle;

typedef struct {
    int length;
    int breadth;
    int height;
} cuboid;

void main() {
    cuboid *cuboidPtr = (cuboid*)malloc(sizeof(cuboid));
    if(cuboidPtr!=NULL) {
        cuboidPtr->length = 10;
        cuboidPtr->breadth = 10;
    }
}
```

Correction — Use Polyspace analysis option

You can allow partial memory allocation for structures, yet not have a red **Illegally dereferenced pointer** error. To allow partial memory allocation, on the **Configuration** pane, under **Check Behavior**, select **Allow incomplete or partial allocation of structures**.

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} rectangle;

typedef struct {
    int length;
    int breadth;
    int height;
} cuboid;

void main() {
    cuboid *cuboidPtr = (cuboid*)malloc(sizeof(rectangle));
    if(cuboidPtr!=NULL) {
        cuboidPtr->length = 10;
        cuboidPtr->breadth = 10;
    }
}
```

Pointer to one field of structure points to another field

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} square;


void main() {
```

```
    square mySquare;
    char* squarePtr = (char*)&mySquare.length;
//Assign zero to mySquare.length byte by byte
    for(int byteIndex=1; byteIndex<=4; byteIndex++) {
        *squarePtr=0;
        squarePtr++;
    }
//Assign zero to first byte of mySquare.breadth
    *squarePtr=0;
}
```

In this example, although `squarePtr` is a `char` pointer, it is assigned the address of the integer `mySquare.length`. Because:

- `char` occupies 1 byte,

- `int` occupies 4 bytes in a 32–bit architecture,

`squarePtr` can access the four bytes of `mySquare.length` through pointer arithmetic. But when it accesses the first byte of another field `mySquare.breadth`, the **Illegally dereferenced pointer** check produces a red error.

Correction — Assign address of structure instead of field

One possible correction is to assign `squarePtr` the address of the full structure `mySquare` instead of `mySquare.length`. `squarePtr` can then access all the bytes of `mySquare` through pointer arithmetic.

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} square;


void main() {
    square mySquare;
    char* squarePtr = (char*)&mySquare;
//Assign zero to mySquare.length byte by byte
    for(int byteIndex=1; byteIndex<=4; byteIndex++) {
        *squarePtr=0;
        squarePtr++;
    }
//Assign zero to first byte of mySquare.breadth
    *squarePtr=0;
}
```

Correction — Use Polyspace analysis option (not available in C++)

You can use a pointer to navigate across the fields of a structure and not produce a red **Illegally dereferenced pointer** error. To allow such navigation, on the **Configuration** pane, under **Check Behavior**, select **Enable pointer arithmetic across fields**.

This option is not available for C++ projects. In C++, pointer arithmetic becomes nontrivial when dealing with concepts such as polymorphic types.

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} square;


void main() {
    square mySquare;
    char* squarePtr = (char*)&mySquare.length;
//Assign zero to mySquare.length byte by byte
    for(int byteIndex=1; byteIndex<=4; byteIndex++) {
        *squarePtr=0;
        squarePtr++;
    }
//Assign zero to first byte of mySquare.breadth
    *squarePtr=0;
}
```

Function returns pointer to local variable

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In the following code, `ptr` points to `ret`. Because the scope of `ret` is limited to `func1`, when `ptr` is accessed in `func2`, the access is illegal. The verification produces a red **Illegally dereferenced pointer** check on `*ptr`.

By default, Polyspace Code Prover does not detect functions returning pointers to local variables. To detect such cases, use the option Detect stack pointer dereference outside scope (-detect-pointer-escape).

Check Information
Group: Static memory
Language: C | C++
Acronym: IDP

See Also
Allow incomplete or partial allocation of structures (-size-in-bytes)|Detect stack pointer dereference outside scope (-detect-pointer-escape)|Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)|Non-initialized pointer

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"

"Code Prover Analysis Following Red and Orange Checks"

Incorrect object oriented programming

Dynamic type of `this` pointer is incorrect

Description

This check on a class member function call determines if the call is valid.

A member function call can be invalid for the following reasons:

- You call the member function through a function pointer that points to the function. However, the data types of the arguments or return values of the function and the function pointer do not match.

- You call a pure `virtual` member function from the class constructor or destructor.

- You call a `virtual` member function through an incorrect `this` pointer. The `this` pointer stores the address of the object used to call the function. The `this` pointer can be incorrect because:

  - You obtain an object through a cast from another object. The objects are instances of two unrelated classes.

  - You perform pointer arithmetic on a pointer pointing to an array of objects. However, the pointer arithmetic causes the pointer to go outside the array bounds. When you dereference the pointer, it is not pointing to a valid object.

Diagnosing This Check

"Review and Fix Incorrect Object Oriented Programming Checks"

Examples

Pointer to method has incorrect type

```
#include <iostream>
class myClass {
public:
  void method() {}
};

void main() {
  myClass Obj;
  int (myClass::*methodPtr) (void) = (int (myClass::*) (void))
&myClass::method;
  int res = (Obj.*methodPtr)();
  std::cout << "Result = " << res;
}
```

In this example, the pointer `methodPtr` has return type `int` but points to `myClass:method` that has return type `void`. Therefore, when `methodPtr` is dereferenced, the **Incorrect object oriented programming** check produces a red error.

Pointer to method contains NULL when dereferenced

```
#include <iostream>
class myClass {
public:
  void method() {}
};

void main() {
  myClass Obj;
  void (myClass::*methodPtr) (void) =  &myClass::method;
  methodPtr = 0;
  (Obj.*methodPtr)();
}
```

In this example, `methodPtr` has value `NULL` when it is dereferenced.

Pure virtual function is called in base class constructor

```
class Shape {
public:
  Shape(Shape *myShape) {
    myShape->setShapeDimensions(0.0);
  }
  virtual void setShapeDimensions(double) = 0;
};

class Square: public Shape {
  double side;
public:
  Square():Shape(this) {
  }
  void setShapeDimensions(double);
};

void Square::setShapeDimensions(double val) {
  side=val;
}

void main() {
  Square sq;
  sq.setShapeDimensions(1.0);
}
```

In this example, the derived class constructor `Square::Square` calls the base class constructor `Shape::Shape()` with its `this` pointer. The base class constructor then calls the pure virtual function `Shape::setShapeDimensions` through the `this` pointer. Since the call to a pure virtual function from a constructor is undefined, the **Incorrect object oriented programming** check produces a red error.

Incorrect this Pointer: Cast Between Pointers to Unrelated Objects

```
#include <new>

class Foo {
public:
  void funcFoo() {}
};
```

```
class Bar {
public:
  virtual void funcBar() {}
};

void main() {
  Foo *FooPtr = new Foo;
  Bar *BarPtr = (Bar*)(void*)FooPtr;
  BarPtr->funcBar();
}
```

In this example, the classes Foo and Bar are not related. When a Foo* pointer is cast to a Bar* pointer and the Bar* pointer is used to call a virtual member function of class Bar, the **Incorrect object oriented programming** check produces a red error.

Incorrect this Pointer: Pointer Out of Bounds

```
#include <new>
class Foo {
public:
    virtual void func() {}
};

void main() {
    Foo *FooPtr = new Foo[4];
    for(int i=0; i<=4; i++)
        FooPtr++;
    FooPtr->func();
    delete [] FooPtr;
}
```

In this example, the pointer FooPtr points outside the allocated bounds when it is used to call the virtual member function func(). It does not point to a valid object. Therefore, the **Incorrect object oriented programming** check produces a red error.

Incorrect this Pointer: Non-initialized Object

```
class Foo {
public:
  virtual int func() {
    return 1;
  }
};

class Ref {
public:
  Ref(Foo* foo) {
    foo->func();
  }
};

class Bar {
private:
  Ref m_ref;
  Foo m_Foo;
public:
```

```
    Bar() : m_ref(&m_Foo) {}
};
```

In this example, the constructor `Bar::Bar()` calls the constructor `Ref::Ref()` with the address of `m_Foo` before `m_Foo` is initialized. When the `virtual` member function `func` is called through a pointer pointing to &m_Foo, the **Incorrect object oriented programming** check produces a red error.

To reproduce the results, analyze only the class `Bar` using the option `Class (-class-analyzer)`.

Incorrect this Pointer: Cast from Base to Derived Class Pointer

```
#include <new>

class Foo {
public:
  virtual void funcFoo() {}
};


class Bar: public Foo {
public:
  void funcFoo() {}
};

void main() {
  Foo *FooPtr = new Foo;
  Bar *BarPtr = (Bar*)(void*)FooPtr;
  BarPtr->funcFoo();
}
```

In this example, you might intend to call the derived class version of `funcFoo` but depending on your compiler, you call the base class version or encounter a segmentation fault.

The pointer `FooPtr` points to a `Foo` object. The cast incorrectly attempts to convert the `Foo*` pointer `FooPtr` to a `Bar*` pointer `BarPtr`. `BarPtr` still points to the base `Foo` object and cannot access `Bar::funcFoo`.

Correction – Make Base Class Pointer Point Directly to Derived Class Object

C++ polymorphism allows defining a pointer that can traverse the class hierarchy to point to the most derived member function. To implement polymorphism correctly, start from the base class pointer and make it point to a derived class object.

```
#include <new>

class Foo {
public:
  virtual void funcFoo() {}
};


class Bar: public Foo {
public:
  void funcFoo() {}
};

void main() {
```

```
    Foo *FooPtr = new Bar;
    FooPtr->funcFoo();
}
```

Check Information

Group: C++
Language: C++
Acronym: OOP

See Also

Base class destructor not virtual | Incompatible types prevent overriding |
Missing virtual inheritance | Partial override of overloaded virtual functions

Topics

"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Invalid C++ specific operations

C++ specific invalid operations occur

Description

These checks on C++ code operations determine whether the operations are valid. The checks look for a range of invalid behaviors:

- Array size is not strictly positive.
- `typeid` operator dereferences a `NULL` pointer.
- `dynamic_cast` operator performs an invalid cast.
- (C++11 and beyond) The number of array initializer clauses exceeds the number of array elements to initialize.
- (C++11 and beyond) The pointer argument to a placement new operator does not point to enough memory.

Diagnosing This Check

"Review and Fix Invalid C++ Specific Operations Checks"

Examples

Array size Not Strictly Positive

```
class License {
protected:
  int numberOfUsers;
  char (*userList)[20];
  int *licenseList;
public:
  License(int numberOfLicenses);
  void initializeList();
  char* getUser(int);
  int getLicense(int);
};

License::License(int numberOfLicenses) : numberOfUsers(numberOfLicenses) {
  userList = new char [numberOfUsers][20];
  licenseList = new int [numberOfUsers];
  initializeList();
}

int getNumberOfLicenses();
int getIndexForSearch();

void main() {
  int n = getNumberOfLicenses();
  if(n >= 0 && n <= 100) {
    License myFirm(n);
    int index = getIndexForSearch();
```

```
    myFirm.getUser(index);
    myFirm.getLicense(index);
  }
}
```

In this example, the argument n to the constructor `License::License` falls into two categories:

- `n = 0`: When the `new` operator uses this argument, the **Invalid C++ specific operations** produce an error.

- `n > 0`: When the `new` operator uses this argument, the **Invalid C++ specific operations** is green.

Combining the two categories of arguments, the **Invalid C++ specific operations** produce an orange error on the `new` operator.

typeid Operator Dereferencing NULL Pointer

To see this issue, enable the option `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.

```
#include <iostream>
#include <typeinfo>
#define PI 3.142

class Shape {
public:
  Shape();
  virtual void setVal(double) = 0;
  virtual double area() = 0;
};

class Circle: public Shape {
  double radius;
public:
  Circle(double radiusVal):Shape() {
    setVal(radiusVal);
  }

  void setVal(double radiusVal) {
     radius = radiusVal;
  }

  double area() {
    return (PI * radius * radius);
  }
};


Shape* getShapePtr();

void main() {
  Shape* shapePtr = getShapePtr();
  double val;

  if(typeid(*shapePtr)==typeid(Circle)) {
    std::cout<<"Enter radius:";
```

```
      std::cin>>val;
      shapePtr->setVal(val);
      std::cout<<"Area of circle = "<<shapePtr->area();
  }
  else {
      std::cout<<"Shape is not a circle.";
  }

}
```

In this example, the Shape* pointer shapePtr returned by getShapePtr() function can be NULL. Because a possibly NULL-valued shapePtr is used with the typeid operator, the **Invalid C++ specific operations** check is orange.

Incorrect dynamic_cast on Pointers

```
class Base {
public :
  virtual void func() ;
};

class Derived : public Base  {
};

Base* returnObj(int flag) {
  if(flag==0)
    return new Derived;
  else
    return new Base;
}


int main() {

    Base * ptrBase;
    Derived * ptrDerived;

    ptrBase = returnObj(0) ;
    ptrDerived = dynamic_cast<Derived*>(ptrBase); //Correct dynamic cast
    assert(ptrDerived != 0); //Returned pointer is not null

    ptrBase = returnObj(1);
    ptrDerived = dynamic_cast<Derived*>(ptrBase); //Incorrect dynamic cast
    // Verification continues despite red
    assert(ptrDerived == 0); //Returned pointer is null
}
```

In this example, the **Invalid C++ specific operations** on the dynamic_cast operator are:

- Green, when the pointer ptrBase that the operator casts to Derived is already pointing to a Derived object.
- Red, when the pointer ptrBase that the operator casts to Derived is pointing to a Base object.

  Red checks typically stop the verification in the same scope as the check. However, after red **Invalid C++ specific operations** on dynamic_cast operation involving pointers, the verification continues. The software assumes that the dynamic_cast operator returns a NULL pointer.

Incorrect dynamic_cast on References

```
class Base {
public :
  virtual void func() ;
};

class Derived : public Base  {
};

Base& returnObj(int flag) {
  if(flag==0)
    return *(new Derived);
  else
    return *(new Base);
}


int main() {
  Base & refBase1 = returnObj(0);
  Derived & refDerived1 = dynamic_cast<Derived&>(refBase1); //Correct dynamic cast;

  Base & refBase2 = returnObj(1);
  Derived & refDerived2 = dynamic_cast<Derived&>(refBase2); //Incorrect dynamic cast
  // Analysis stops
  assert(1);
}
```

In this example, the **Invalid C++ specific operations** on the dynamic_cast operator are:

- Green, when the reference refBase1 that the operator casts to Derived& is already referring to a Derived object.

- Red, when the reference refBase2 that the operator casts to Derived& is referring to a Base object.

  After red **Invalid C++ specific operations** on dynamic_cast operation involving pointers, the software does not verify the code in the same scope as the check. For instance, the software does not perform the **User assertion** check on the assert statement.

(C++11 and Beyond) Excess Initializer Clauses in Array Initialization

```
#include <stdio.h>

int* arr_const;

void allocate_consts(int size) {
    if(size>1)
      arr_const = new int[size]{0,1,2};
    else if(size==1)
      arr_const = new int[size]{0,1};
    else
      printf("Nonpositive array size!");
}

int main() {
    allocate_consts(3);
```

```
        allocate_consts(1);
        return 0;
}
```

In this example, the **Invalid C++ specific operations** check determines if the number of initializer clauses match the number of elements to initialize.

In the first call to `allocate_consts`, the initialization list has three elements to initialize an array of size three. The **Invalid C++ specific operations** check on the `new` operator is green. In the second call, the initialization list has two elements but initializes an array of size one. The check on the `new` operator is red.

(C++11 and Beyond) Pointer Argument to Placement `new` Operator Does Not Point to Enough Memory

```
#include <new>

class aClass {
  virtual void func();
};

void allocateNObjects(unsigned int n) {
    char* location = new char[sizeof(aClass)];
    aClass* objectLocation = new(location) aClass[n];
}
```

In this example, memory equal to the size of one `aClass` object is associated with the pointer `location`. However, depending on the function argument `n` more than one object can be allocated when using the placement `new` operator. The pointer `location` might not have enough memory for the objects allocated.

Check Information
Group: C++
Language: C++
Acronym: CPP

See Also

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

External Websites
C++ Reference: `dynamic_cast` conversion

Invalid operation on floats

Result of floating-point operation is NaN for non-NaN operands

Description

This check determines if the result of a floating-point operation is NaN. The check is performed only if you enable a verification mode that incorporates NaNs and specify that the verification must highlight operations that result in NaN.

If you specify that the verification must produce a warning for NaN, the check is:

- Red, if the operation produces NaN on all execution paths that the software considers, and the operands are not NaN.
- Orange, if the operation produces NaN on some of the execution paths when the operands are not NaN.
- Green, if the operation does not produce NaN unless the operands are NaN.

If you specify that the verification must forbid NaN, the check color depends on the result of the operation only. The color does not depend on the operands.

The check also highlights conversions from floating-point variables to integers where the floating-point variable can be NaN. In this case, the check is always performed when you incorporate NaNs in the verification and does not allow NaNs as input to the conversion.

To enable this verification mode, use these options:

- `Consider non finite floats (-allow-non-finite-floats)`
- `NaNs (-check-nan)`: Use argument `warn-first` or `forbid`.

Examples

NaN Detected with Red Check

Results in `forbid` mode:

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

In this example, both the operands of the - operation are not NaN but the result is NaN. The **Invalid operation on floats** check on the - operation is red. In the `forbid` mode, the verification stops after the red check. For instance, a **Non-initialized local variable** check does not appear on y in the `return` statement.

Results in `warn-first` mode:

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
```

```
        return y;
    }
```

In this example, both the operands of the - operation are not NaN but the result is NaN. The **Invalid operation on floats** check on the - operation is red. The red checks in `warn-first` mode are different from red checks for other check types. The verification does not stop after the red check. For instance, a green **Non-initialized local variable** check appears on `y` in the `return` statement. If you place your cursor on `y` in the verification result, you see that it has the value NaN.

NaN Detected with Orange Check

Results in `forbid` mode:

```
double func(double arg1, double arg2) {
    double ret=arg1-arg2;
    return ret;
}
```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can be both infinite, for instance, and the result of `arg1-arg2` can be NaN. In the `forbid` mode, following the check, the verification terminates the execution path that results in NaN. If you place your cursor on `ret` in the `return` statement, it does not have the value NaN.

Results in `warn-first` mode:

```
double func(double arg1, double arg2) {
    double ret=arg1-arg2;
    return ret;
}
```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can be both infinite, for instance, and the result of `arg1-arg2` can be NaN. The orange checks in `warn-first` mode are different from orange checks for other check types. Following the check, the verification does not terminate the execution path that results in NaN. If you place your cursor on `ret` in the `return` statement, it continues to have the value NaN along with other possible values.

Orange Check Despite NaN Being the Only Result

```
double func(double arg1, double arg2) {
    double z=arg1-arg2;
    return z;
}

void caller() {
    double x=1.0/0.0;
    double y=x-x;
    func(x,x);
    func(y,y);
}
```

In this example, in `func`, the result of the - operation is always NaN but the **Invalid operation on floats** check is orange instead of red.

- In the first call to `func`, both the operands `arg1` and `arg2` are not NaN, but the result is NaN. So, the check is red.

- In the second call to `func`, both the operands `arg1` and `arg2` are NaN, and therefore the result is NaN. So, the check is green, indicating that the result is not NaN unless the operands are NaN.

Combining the colors for the two calls to `func`, the check is orange.

In the example, the option `-check-nan warn-first` was used.

NaN in Conversion from Floating Point to Integers

```
void func() {
    double x= 1.0/0.0;
    double y= x-x;
    int z = y;
}
```

In this example, the **Invalid operation on floats** check detects the assignment of NaN to an integer variable `z`.

The check is enabled if you specify that non-finite floats must be considered in the verification. The check blocks further verification on the same execution path irrespective of whether you allow, forbid or ask for warnings on non-finite floats.

Result Information
Group: Numerical
Language: C | C++
Acronym: INVALID_FLOAT_OP

See Also
NaNs (-check-nan)|Overflow|Subnormal float

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"
"Order of Code Prover Run-Time Checks"

Introduced in R2016a

Invalid shift operations

Shift operations are invalid

Description

This check on shift operations on a variable `var` determines:

- Whether the shift amount is larger than the range allowed by the type of `var`.
- If the shift is a left shift, whether `var` is negative.

Diagnosing This Check

"Review and Fix Invalid Shift Operations Checks"

Examples

Shift amount outside bounds

```
#include <stdlib.h>
#define shiftAmount 32
enum shiftType {
 SIGNED_LEFT,
 SIGNED_RIGHT,
 UNSIGNED_LEFT,
 UNSIGNED_RIGHT
};

enum shiftType getShiftType();

void main() {
  enum shiftType myShiftType = getShiftType();
  int signedInteger = 1;
  unsigned int unsignedInteger = 1;
  switch(myShiftType) {
  case SIGNED_LEFT:
    signedInteger = signedInteger << shiftAmount;
    break;
  case SIGNED_RIGHT:
    signedInteger = signedInteger >> shiftAmount;
    break;
  case UNSIGNED_LEFT:
    unsignedInteger = unsignedInteger << shiftAmount;
    break;
  case UNSIGNED_RIGHT:
    unsignedInteger = unsignedInteger >> shiftAmount;
    break;
  }
}
```

In this example, the shift amount `shiftAmount` is outside the allowed range for both signed and unsigned `int`. Therefore the **Invalid shift operations** check produces a red error.

Correction — Keep shift amount within bounds

One possible correction is to keep the shift amount in the range 0..31 for unsigned integers and 0...30 for signed integers. This correction works if the size of `int` is 32 on the target processor.

```
#include <stdlib.h>
#define shiftAmountSigned 30
#define shiftAmount 31
enum shiftType {
 SIGNED_LEFT,
 SIGNED_RIGHT,
 UNSIGNED_LEFT,
 UNSIGNED_RIGHT
};

enum shiftType getShiftType();

void main() {
  enum shiftType myShiftType = getShiftType();
  int signedInteger = 1;
  unsigned int unsignedInteger = 1;
  switch(myShiftType) {

  case SIGNED_LEFT:
    signedInteger = signedInteger << shiftAmountSigned;
    break;

  case SIGNED_RIGHT:
    signedInteger = signedInteger >> shiftAmountSigned;
    break;

  case UNSIGNED_LEFT:
    unsignedInteger = unsignedInteger << shiftAmount;
     break;

  case UNSIGNED_RIGHT:
    unsignedInteger = unsignedInteger >> shiftAmount;
    break;
  }
}
```

Left operand of left shift is negative

```
void main(void) {
  int x = -200;
  int y;
  y = x << 1;
}
```

In this example, the left operand of the left shift operation is negative.

Correction — Use Polyspace analysis option

You can use left shifts on negative numbers and not produce a red **Invalid shift operations** error. To allow such left shifts, on the **Configuration** pane, under **Check Behavior**, select **Allow negative operand for left shifts**.

```
void main(void) {
```

```
  int x = -200;
  int y;
  y = x << 1;
}
```

Left operand of left shift may be negative

```
short getVal();

int foo(void) {
  long lvar;
  short svar1, svar2;

  lvar = 0;
  svar1 = getVal();
  svar2 = getVal();

  lvar =  (svar1 - svar2) << 10;
  if (svar1 < svar2)  {
    return 1;
  } else {
    return 0;
  }
}

int main(void) {
  return foo();
}
```

In this example, if svar1 < svar2, the left operand of << can be negative. Therefore the **Shift operations** check on << is orange. Following an orange check, execution paths containing the error get truncated. Therefore, following the orange **Invalid shift operations** check, Polyspace assumes that svar1 >= svar2. The branch of the statement, if(svar1 < svar2), is unreachable.

Check Information
Group: Numerical
Language: C | C++
Acronym: SHF

See Also
Allow negative operand for left shifts (-allow-negative-operand-in-shift) |
Consider non finite floats (-allow-non-finite-floats)

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Invalid use of standard library routine

Standard library function is called with invalid arguments

Description

This check on a standard library function call determines whether the function is called with valid arguments.

The check works differently for memory routines, floating point routines or string routines because their arguments can be invalid in different ways. For more information on each type of routines, see the following examples.

Diagnosing This Check

"Review and Fix Invalid Use of Standard Library Routine Checks"

Examples

Invalid use of standard library float routine

```
#include <assert.h>
#include <math.h>

#define LARGE_EXP 710

enum operation {
    ASIN,
    ACOS,
    TAN,
    SQRT,
    LOG,
    EXP,
};

enum operation getOperation(void);
double getVal(void);

void main() {
    enum operation myOperation = getOperation();
    double myVal=getVal(), res;
    switch(myOperation) {
    case ASIN:
        assert( myVal <- 1.0 || myVal > 1.0);
        res = asin(myVal);
        break;
    case ACOS:
        assert( myVal < -1.0 || myVal > 1.0);
        res = acos(myVal);
        break;
    case SQRT:
        assert( myVal < 0.0);
        res = sqrt(myVal);
```

```
            break;
        case LOG:
            assert(myVal <= 0.0);
            res = log(myVal);
            break;
        case EXP:
            assert(myVal > LARGE_EXP);
            res = exp(myVal);
            break;
    }
}
```

In this example, following each `assert` statement, Polyspace considers that `myVal` contains only those values that make the `assert` condition true. For example, following `assert(myVal < 1.0);`, Polyspace considers that `myVal` contains values less than 1.0.

When `myVal` is used as argument in a standard library function, its values are invalid for the function. Therefore, the **Invalid use of standard library routine** check produces a red error.

To learn more about the specifications of this check for floating point routines, see "Invalid Use of Standard Library Floating Point Routines".

Invalid use of standard library memory routine

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void) {
  char str1[10],str2[5];
  printf("Enter string:\n");
  scanf("%s",str1);
  memcpy(str2,str1,6);
  return str2;
}

int main(void) {
  (void*)Copy_First_Six_Letters();
  return 0;
}
```

In this example, the size of string `str2` is 5, but 6 characters of string `str1` are copied into `str2` using the `memcpy` function. Therefore, the **Invalid use of standard library routine** check on the call to `memcpy` produces a red error.

For other examples, see "Assumptions About memset and memcpy" on page 4-24.

Correction — Call function with valid arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void) {
  char str1[10],str2[6];
  printf("Enter string:\n");
```

```
  scanf("%s",str1);
  memcpy(str2,str1,6);
  return str2;
}

int main(void) {
  (void*)Copy_First_Six_Letters();
  return 0;
}
```

Invalid use of standard library string routine

```
#include <stdio.h>
#include <string.h>

char* Copy_String(void)
{
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";
  res=strcpy(gbuffer,text);
  return(res);
}

int main(void) {
  (void*)Copy_String();
}
```

In this example, the string text is larger in size than gbuffer. Therefore, when you copy text into gbuffer. the **Invalid use of standard library routine** check on the call to strcpy produces a red error.

Correction — Call function with valid arguments

One possible correction is to declare the destination string gbuffer with equal or larger size than the source string text.

```
#include <stdio.h>
#include <string.h>

char* Copy_String(void)
{
  char *res;
  char gbuffer[20],text[20]="ABCDEFGHIJKL";
  res=strcpy(gbuffer,text);
  return(res);
}

int main(void) {
  (void*)Copy_String();
}
```

Check Information
Group: Other
Language: C | C++
Acronym: STD_LIB

See Also

`Consider non finite floats (-allow-non-finite-floats)` | `Float rounding mode (-float-rounding-mode)`

Topics

"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"
"Assumptions About memset and memcpy" on page 4-24

Non-initialized local variable

Local variable is not initialized before being read

Description

This check occurs for every local variable read. It determines whether the variable being read is initialized.

Diagnosing This Check

"Review and Fix Non-initialized Local Variable Checks"

Examples

Non-initialized variable used on right side of assignment operator

```
#include <stdio.h>

void main(void) {
   int sum;
   for(int i=1;i <= 10; i++)
      sum+=i;
   printf("The sum of the first 10 natural numbers is %d.", sum);
 }
```

The statement `sum+=i;` is the shorthand for `sum=sum+i;`. Because `sum` is used on the right side of an expression before being initialized, the **Non-initialized local variable** check returns a red error.

Correction — Initialize variable before using on right side of assignment

One possible correction is to initialize `sum` before the `for` loop.

```
#include <stdio.h>

void main(void) {
   int sum=0;
   for(int i=1;i <= 10; i++)
      sum+=i;
   printf("The sum of the first 10 natural numbers is %d.", sum);
 }
```

Non-initialized variable used with relational operator

```
#include <stdio.h>

int getTerm();

void main(void) {
    int count,sum=0,term;

    while( count <= 10  && sum <1000) {
        count++;
        term = getTerm();
```

```
        if(term > 0 && term <= 1000) sum += term;
      }

    printf("The sum of 10 terms is %d.", sum);
 }
```

In this example, the variable `count` is not initialized before the comparison `count <= 10`. Therefore, the **Non-initialized local variable** check returns a red error.

Correction — Initialize variable before using with relational operator

One possible correction is to initialize `count` before the comparison `count <= 10`.

```
#include <stdio.h>

int getTerm();

void main(void) {
    int count=1,sum=0,term;

    while( count <= 10  && sum <1000) {
       count++;
       term = getTerm();
       if(term > 0 && term <= 1000) sum+= term;
      }

    printf("The sum of 10 terms is %d.", sum);
 }
```

Non-initialized variable passed to function

```
#include <stdio.h>

int getShift();
int shift(int var) {
    int shiftVal = getShift();
    if(shiftVal > 0 && shiftVal < 1000)
        return(var+shiftVal);
    return 1000;
}

void main(void) {
    int initVal;
    printf("The result of a shift is %d",shift(initVal));
}
```

In this example, `initVal` is not initialized when it is passed to `shift()`. Therefore, the **Non-initialized local variable** check returns a red error. Because of the red error, Polyspace does not verify the operations in `shift()`.

Correction — Initialize variable before passing to function

One possible correction is to initialize `initVal` before passing to `shift()`. `initVal` can be initialized through an input function. To avoid an overflow, the value returned from the input function must be within bounds.

```
#include <stdio.h>
```

```
int getShift();
int getInit();
int shift(int var) {
    int shiftVal = getShift();
    if(shiftVal > 0 && shiftVal < 1000)
        return(var+shiftVal);
    return 1000;
}

void main(void) {
   int initVal=getInit();
   if(initVal >0 && initVal < 1000)
     printf("The result of a shift is %d",shift(initVal));
   else
     printf("Value must be between 0 and 1000.");
 }
```

Non-initialized array element

```
#include <stdio.h>
#define arrSize 19

void main(void)
{
  int arr[arrSize],indexFront, indexBack;
  for(indexFront = 0,indexBack = arrSize - 1;
    indexFront < arrSize/2;
    indexFront++, indexBack--) {
    arr[indexFront] = indexFront;
    arr[indexBack] = arrSize - indexBack - 1;
  }
  printf("The array elements are: \n");
  for(indexFront = 0; indexFront < arrSize; indexFront++)
    printf("Element[%d]: %d", indexFront, arr[indexFront]);
 }
```

In this example, in the first for loop:

- indexFront runs from 0 to 8.

- indexBack runs from 18 to 10.

Therefore, arr[9] is not initialized. In the second for loop, when arr[9] is passed to printf, the **Non-initialized local variable** check returns an error. The error is orange because the check returns an error only in one of the loop runs.

Due to the orange error in one of the loop runs, a red **Non-terminating loop** error appears on the second for loop.

Correction — Initialize variable before passing to function

One possible correction is to keep the first for loop intact and initialize arr[9] outside the for loop.

```
#include <stdio.h>
#define arrSize 19

void main(void)
{
  int arr[arrSize],indexFront, indexBack;
```

```
  for(indexFront = 0,indexBack = arrSize - 1;
    indexFront < arrSize/2;
    indexFront++, indexBack--) {
    arr[indexFront] = indexFront;
    arr[indexBack] = arrSize - indexBack - 1;
  }
  arr[indexFront] = indexFront;
  printf("The array elements are: \n");
  for(indexFront = 0; indexFront < arrSize; indexFront++)
    printf("Element[%d]: %d", indexFront, arr[indexFront]);
}
```

Non-initialized structure

```
typedef struct S {
    int integerField;
    char characterField;
}S;

void operateOnStructure(S);
void operateOnStructureField(int);

void main() {
  S myStruct;
  operateOnStructure(myStruct);
  operateOnStructureField(myStruct.integerField);
}
```

In this example, the structure myStruct is not initialized. Therefore, when the structure myStruct is passed to the function operateOnStructure, a **Non-initialized local variable** check on the structure appears red.

Correction— Initialize structure

One possible correction is to initialize the structure myStruct before passing it to a function.

```
typedef struct S {
    int integerField;
    char characterField;
}S;

void operateOnStructure(S);
void operateOnStructureField(int);

void main() {
  S myStruct = {0,' '};
  operateOnStructure(myStruct);
  operateOnStructureField(myStruct.integerField);
}
```

Partially initialized structure — All used fields initialized

```
typedef struct S {
    int integerField;
    char characterField;
    double doubleField;
}S;
```

```
int getIntegerField(void);
char getCharacterField(void);

void printIntegerField(int);
void printCharacterField(char);

void printFields(S s) {
 printIntegerField(s.integerField);
 printCharacterField(s.characterField);
}

void main() {
  S myStruct;

  myStruct.integerField = getIntegerField();
  myStruct.characterField = getCharacterField();
  printFields(myStruct);
}
```

In this example, the **Non-initialized local variable** check on `myStruct` is green because:

- The fields `integerField` and `characterField` that are used are both initialized.
- Although the field `doubleField` is not initialized, there is no read or write operation on the field `doubleField` in the code.

To determine which fields are checked for initialization:

1   Select the check on the **Results List** pane or **Source** pane.
2   View the message on the **Result Details** pane.

Note that in the special case where none of the fields are used, the checks for initialization are orange instead of green if all the fields.are uninitialized.

Partially initialized structure — Some used fields initialized

```
typedef struct S {
    int integerField;
    char characterField;
    double doubleField;
}S;

int getIntegerField(void);
char getCharacterField(void);

void printIntegerField(int);
void printCharacterField(char);
void printDoubleField(double);

void printFields(S s) {
 printIntegerField(s.integerField);
 printCharacterField(s.characterField);
 printDoubleField(s.doubleField);
}

void main() {
  S myStruct;
```

3-69

```
    myStruct.integerField = getIntegerField();
    myStruct.characterField = getCharacterField();
    printFields(myStruct);
}
```

In this example, the **Non-initialized local variable** check on `myStruct` is orange because:

- The fields `integerField` and `characterField` that are used are both initialized.
- The field `doubleField` is not initialized and there is a read operation on `doubleField` in the code.

To determine which fields are checked for initialization:

1   Select the check on the **Results List** pane or **Source** pane.
2   View the message on the **Result Details** pane.

Check Information

Group: Data flow
Language: C | C++
Acronym: NIVL

See Also

```
Disable checks for non-initialization (-disable-initialization-checks)|Non-
initialized pointer|Non-initialized variable
```

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Non-initialized pointer

Pointer is not initialized before being read

Description

This check occurs for every pointer read. It determines whether the pointer being read is initialized.

Diagnosing This Check

"Review and Fix Non-initialized Pointer Checks"

Examples

Non-initialized pointer passed to function

```
int assignValueToAddress(int *ptr) {
  *ptr = 0;
}

void main() {
 int* newPtr;
 assignValueToAddress(newPtr);
}
```

In this example, `newPtr` is not initialized before it is passed to `assignValueToAddress()`.

Correction — Initialize pointer before passing to function

One possible correction is to assign `newPtr` an address before passing to `assignValueToAddress()`.

```
int assignValueToAddress(int *ptr) {
  *ptr = 0;
}

void main() {
 int val;
 int* newPtr = &val;
 assignValueToAddress(newPtr);
}
```

Non-initialized pointer to structure

```
#include <stdlib.h>
#define stackSize 25

typedef struct stackElement {
  int value;
  int *prev;
}stackElement;

int input();
```

```
void main() {
 stackElement *stackTop;

 for (int count = 0; count < stackSize; count++) {
    if(stackTop!=NULL) {
        stackTop -> value = input();
        stackTop -> prev = (int*)stackTop;
    }
    stackTop = (stackElement*)malloc(sizeof(stackElement));
 }
}
```

In this example, in the first run of the for loop, stackTop is not initialized and does not point to a valid address. Therefore, the **Non-initialized pointer** check on stackTop!=NULL returns a red error.

Correction — Initialize pointer before dereference

One possible correction is to initialize stackTop through malloc() before the check stackTop! =NULL.

```
#include <stdlib.h>
#define stackSize 25

typedef struct stackElement {
  int value;
  int *prev;
}stackElement;

int input();

void main() {
 stackElement *stackTop;

 for (int count = 0; count < stackSize; count++) {
    stackTop = (stackElement*)malloc(sizeof(stackElement));
    if(stackTop!=NULL) {
        stackTop->value = input();
        stackTop->prev = (int*)stackTop;
    }
 }
}
```

Non-initialized char* pointer used to store string

```
#include <stdio.h>

void main() {
 char *str;
 scanf("%s",str);
}
```

In this example, str does not point to a valid address. Therefore, when the scanf function reads a string from the standard input to str, the **Non-initialized pointer** check returns a red error.

Correction — Use char array instead of char* pointer

One possible correction is to declare `str` as a `char` array. This declaration assigns an address to the `char*` pointer associated with the array name `str`. You can then use the pointer as input to `scanf`.

```
#include <stdio.h>

void main() {
 char str[10];
 scanf("%s",str);
}
```

Non-initialized array of char* pointers used to store variable-size strings

```
#include <stdio.h>

void assignDataBaseElement(char** str) {
 scanf("%s",*str);
}

void main() {
 char *dataBase[20];

 for(int count = 1; count < 20 ; count++) {
    assignDataBaseElement(&dataBase[count]);
    printf("Database element %d : %s",count,dataBase[count]);
 }
}
```

In this example, `dataBase` is an array of `char*` pointers. In each run of the `for` loop, an element of `dataBase` is passed via pointers to the function `assignDataBaseElement()`. The element passed is not initialized and does not contain a valid address. Therefore, when the element is used to store a string from standard input, the **Non-initialized pointer** check returns a red error.

Correction — Initialize char* pointers through calloc

One possible correction is to initialize each element of `dataBase` through the `calloc()` function before passing it to `assignDataBaseElement()`. The initialization through `calloc()` allows the `char` pointers in `dataBase` to point to strings of varying size.

```
#include <stdio.h>
#include <stdlib.h>

void assignDataBaseElement(char** str) {
 scanf("%s",*str);
}
int inputSize();

void main() {
 char *dataBase[20];

 for(int count = 1; count < 20 ; count++) {
    dataBase[count] = (char*)calloc(inputSize(),sizeof(char));
    assignDataBaseElement(&dataBase[count]);
    printf("Database element %d : %s",count,dataBase[count]);
 }
}
```

Check Information

Group: Data flow
Language: C | C++
Acronym: NIP

See Also

`Disable checks for non-initialization (-disable-initialization-checks)`|`Non-initialized local variable`|`Non-initialized variable`

Topics

"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Non-initialized variable

Variable other than local variable is not initialized before being read

Description

This check occurs when you read variables that are not local (global or static variables). It determines whether the variable being read is initialized.

By default, Polyspace considers that global variables are initialized. The verification checks global variables only if you prevent this default initialization. See also "Assumptions About Global Variable Initialization" on page 4-15.

For more examples of initialization of complex data types, see the equivalent checker for local variables, `Non-initialized local variable`.

Diagnosing This Check

"Review and Fix Non-initialized Variable Checks"

Examples

Non-initialized global variable

```
int globVar;
int getVal();

void main() {
 int val = getVal();
 if(val>=0 && val<= 100)
    globVar += val;
}
```

In this example, `globVar` does not have an initial value when incremented. Therefore, the **Non-initialized variable** check produces a red error.

The example uses the option to prevent default initialization of global variables.

Correction — Initialize global variable before use

One possible correction is to initialize the global variable `globVar` before use.

```
int globVar;
int getVal();

void main() {
 int val = getVal();
 globVar = 0;
 if(val>=0 && val<= 100)
    globVar += val;
}
```

Check Information
Group: Data flow
Language: C | C++
Acronym: NIV

See Also
`Disable checks for non-initialization (-disable-initialization-checks)`|`Global variable not assigned a value in initialization code`|`Ignore default initialization of global variables (-no-def-init-glob)`|`Non-initialized local variable`|`Non-initialized pointer`

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"
"Assumptions About Global Variable Initialization" on page 4-15

Non-terminating call

Called function does not return to calling context

Description

This check on a function call appears when the following conditions hold:

- The called function does not return to its calling context. The call leads to a definite run-time error or a process termination function like `exit()` in the function body.
- There are other calls to the same function that do not lead to a definite error or process termination function in the function body.

When only a fraction of calls to a function lead to a definite error, this check helps identify those function calls. In the function body, even though a definite error occurs, the error appears in orange instead of red because the verification results in a function body are aggregated over all function calls. To indicate that a definite error has occurred, a red **Non-terminating call** check is shown *on the function call* instead.

Otherwise, if all the calls to a function lead to a definite error or process termination function in the function body, the **Non-terminating call** error is not displayed. The error appears in red in the function body and a dashed red underline appears on the function calls. However, following the function call, like other red errors, Polyspace does not analyze the remaining code in the same scope as the function call.

You can navigate directly from the function call to the operation causing the run-time error in the function body.

- To find the source of error, on the **Source** pane, place your cursor on the loop keyword and view the tooltip.
- Navigate to the source of error in the function body. Right-click the function call and select **Go to Cause** if the option exists.

  If the error is the result of multiple causes, the option takes you to the first cause in the function body. Multiple causes can occur, for instance, when some values of a function argument trigger one specific error and other values trigger other errors.

Diagnosing This Check

"Review and Fix Non-Terminating Call Checks"

Examples

Dashed red underline on function call

```
#include<stdio.h>
double ratio(int num, int den) {
  return(num/den);
}

void main() {
```

```
    int i,j;
    i=2;
    j=0;
    printf("%.2f",ratio(i,j));
}
```

In this example, a red **Division by zero** error appears in the body of `ratio`. This **Division by zero** error in the body of `ratio` causes a dashed red underline on the call to `ratio`.

Red underline on function call

```
#include<stdio.h>
double ratio(int num, int den) {
    return(num/den);
}

int inputCh();

void main() {
    int i,j,ch=inputCh();
    i=2;

    if(ch==1)  {
        j=0;
        printf("%.2f",ratio(i,j));
    }
    else {
        j=2;
        printf("%.2f",ratio(i,j));
    }
}
```

In this example, there are two calls to `ratio`. In the first call, a **Division by zero** error occurs in the body of `ratio`. In the second call, Polyspace does not find errors. Therefore, combining the two calls, an orange **Division by zero** check appears in the body of `ratio`. A red **Non-terminating call** check on the first call indicates the error.

Red underline on call through function pointer

```
typedef void (*f)(void);
// function pointer type

void f1(void) {
    int x;
    x++;
}

void f2(void) { }
void f3(void) { }

f fptr_array[3] = {f1,f2,f3};
unsigned char getIndex(void);

void main(void) {
 unsigned char index = getIndex() % 3;
 // Index is between 0 and 2

 fptr_array[index]();
```

```
 fptr_array[index]();
}
```

In this example, because `index` can lie between 0 and 2, the first `fptr_array[index]()` can call `f1`, `f2` or `f3`. If `index` is zero, the statement calls `f1`. `f1` contains a red **Non-initialized local variable** error, therefore, a dashed red error appears on the function call. Unlike other red errors, the verification continues.

After this statement, the software considers that `index` is either 1 or 2. An error does not occur on the second `fptr_array[index]()`.

Check Information

Group: Control flow
Language: C | C++
Acronym: NTC

See Also

`Non-terminating loop`

Topics
"Identify Function Call with Run-Time Error"
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Non-terminating loop

Loop does not terminate or contains an error

Description

This check on a loop determines if the loop has one of the following issues:

- The loop definitely does not terminate.

  The check appears only if Polyspace cannot detect an exit path from the loop. For example, if the loop appears in a function and the loop termination condition is met for some function inputs, the check does not appear, even though the condition might not be met for some other inputs.

- The loop contains a definite error in one its iterations.

  Even though a definite error occurs in one loop iteration, because the verification results in a loop body are aggregated over all loop iterations, the error shows as an orange check in the loop body. To indicate that a definite failure has occurred, a red **Non-terminating loop** check is shown on the loop command.

Unlike other checks, this check appears only when a definite error occurs. In your verification results, the check is always red. If the error occurs only in some cases, for instance, if the loop bound is variable and causes an issue only for some values, the check does not appear. Instead, the loop command is shown in dashed red with more information in the tooltip.

The check also does not appear if both conditions are true:

- The loop has a trivial predicate such as `for(;;)` or `while(1)`.
- The loop has an empty body, or a body without an exit statement such as `break`, `goto`, `return` or an exception.

Instead, the loop statement is underlined with red dashes. If you place your cursor on the loop statement, you see that the verification considers the loop as intentional. If you deliberately introduce infinite loops, for instance, to emulate cyclic tasks, you do not have to justify red checks.

Using this check, you can identify the operation in the loop that causes the run-time error.

- To find the source of error, on the **Source** pane, place your cursor on the function call and view the tooltip.
- For loops with fewer iterations, you can navigate to the source of error in the loop body. Select the loop to see the full history of the result. Alternatively, right-click the loop keyword and select **Go to Cause** if the option exists.

Diagnosing This Check

"Review and Fix Non-Terminating Loop Checks"

Examples

Loop does not terminate

```
#include<stdio.h>

void main() {
  int i=0;
  while(i<10) {
    printf("%d",i);
  }
}
```

In this example, in the `while` loop, `i` does not increase. Therefore, the test `i<10` never fails.

Correction — Ensure Loop Termination

One possible correction is to update `i` such that the test `i<10` fails after some loop iterations and the loop terminates.

```
#include<stdio.h>

void main() {
  int i=0;
  while(i < 10) {
    printf("%d",i);
    i++;
  }
}
```

Loop contains an out of bounds array index error

```
void main() {
  int arr[20];
  for(int i=0; i<=20; i++) {
    arr[i]=0;
  }
}
```

In this example, the last run of the `for` loop contains an **Out of bounds array index** error. Therefore, the **Non-terminating loop** check on the `for` loop is red. A tooltip appears on the `for` loop stating the maximum number of iterations including the one containing the run-time error.

Correction — Avoid loop iteration containing error

One possible correction is to reduce the number of loop iterations so that the **Out of bounds array index** error does not occur.

```
void main() {
  int arr[20];
  for(int i=0; i<20; i++) {
    arr[i]=0;
  }
}
```

Loop contains an error in function call

```
int arr[4];
```

```
void assignValue(int index) {
  arr[index] = 0;
}

void main() {
  for(int i=0;i<=4;i++)
    assignValue(i);
}
```

In this example, the call to function `assignValue` in the last `for` loop iteration contains an error. Therefore, although an error does not show in the `for` loop body, a red **Non-terminating loop** appears on the loop itself.

Correction — Avoid loop iteration containing error

One possible correction is to reduce the number of loop iterations so the error in the call to `assignValue` does not occur.

```
int arr[4];

void assignValue(int index) {
  arr[index] = 0;
}

void main() {
 for(int i=0;i<4;i++)
    assignValue(i);
}
```

Loop contains an overflow error

```
#define MAX 1024
void main() {
 int i=0,val=1;
 while(i<MAX) {
   val*=2;
   i++;
  }
}
```

In this example, an **Overflow** error occurs in iteration number 31. Therefore, the **Non-terminating loop** check on the `while` loop is red. A tooltip appears on the `while` loop stating the maximum number of iterations including the one containing the run-time error.

Correction — Reduce loop iterations

One possible correction is to reduce the number of loop iterations so that the overflow does not occur.

```
#define MAX 30
void main() {
 int i=0,val=1;
 while(i<MAX) {
   val*=2;
   i++;
  }
}
```

Check Information

Group: Control flow
Language: C | C++
Acronym: NTL

See Also

`Non-terminating call`

Topics

"Identify Loop Operation with Run-Time Error"
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Null this-pointer calling method

`this` pointer is null during member function call

Description

This check on a `this` pointer dereference determines whether the pointer is `NULL`.

Diagnosing This Check

"Review and Fix Null This-pointer Calling Method Checks"

Examples

Pointer to object is NULL during member function call

```
#include <stdlib.h>
class Company {
 public:
  Company(int initialNumber):numberOfClients(initialNumber) {}
  void addNewClient() {
    numberOfClients++;
  }
 protected:
  int numberOfClients;
};

void main() {
 Company* myCompany = NULL;
 myCompany->addNewClient();
}
```

In this example, the pointer `myCompany` is initialized to `NULL`. Therefore when the pointer is used to call the member function `addNewClient`, the **Null this-pointer calling method** produces a red error.

Correction — Initialize pointer with valid address

One possible correction is to initialize `myCompany` with a valid memory address using the `new` operator.

```
#include <stdlib.h>
class Company {
 public:
  Company(int initialNumber):numberOfClients(initialNumber) {}
  void addNewClient() {
    numberOfClients++;
  }
 protected:
  int numberOfClients;
};

void main() {
```

```
Company* myCompany = new Company(0);
myCompany->addNewClient();
}
```

Check Information
Group: C++
Language: C++
Acronym: NNT

See Also

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Out of bounds array index

Array is accessed outside range

Description

This check on an array element access determines whether the element is outside the array range. The check occurs only when you read an array element using the index notation and not when you take the address of the array element.

Diagnosing This Check

"Review and Fix Out of Bounds Array Index Checks"

Examples

Array index is equal to array size

```
#include <stdio.h>

void fibonacci(void)
{
  int i;
  int fib[10];

  for (i = 0; i < 10; i++)
  {
    if (i < 2)
      fib[i] = 1;
    else
      fib[i] = fib[i-1] + fib[i-2];
    }

  printf("The 10-th Fibonacci number is %i .\n", fib[i]);
}

int main(void) {
  fibonacci();
}
```

In this example, the array `fib` is assigned a size of 10. An array index for `fib` has allowed values of [0,1,2,...,9]. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, when the `printf` statement attempts to access `fib[10]` through `i`, the **Out of bounds array index** check produces a red error.

The check also produces a red error if `printf` uses `*(fib+i)` instead of `fib[i]`.

Correction — Keep array index less than array size

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>
```

```
void fibonacci(void)
{
  int i;
  int fib[10];

  for (i = 0; i < 10; i++)
  {
    if (i < 2)
      fib[i] = 1;
    else
      fib[i] = fib[i-1] + fib[i-2];
    }

  printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}

int main(void) {
  fibonacci();
}
```

Accessing external arrays with undefined size

```
extern int arr[];

int getFifthElement(void) {
    return arr[5];
}
int main(void){
    getFifthElement();
}
```

Code Prover assumes by default that external arrays of undefined size can be safely accessed at any index. The **Out of bounds array index** check on the external array access is green.

To remove this default assumption, use the option `-consider-external-array-access-unsafe`. With this option, the **Out of bounds array index** check is orange.

```
extern int arr[];

int getFifthElement(void) {
    return arr[5];
}
```

Check Information
Group: Static memory
Language: C | C++
Acronym: OBAI

See Also
`Illegally dereferenced pointer` | `-consider-external-array-access-unsafe`

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Overflow

Arithmetic operation causes overflow

Description

This check on an arithmetic operation determines whether the result overflows. The result of this check depends on whether you allow nonfinite float results such as infinity and NaN.

The result of the check also depends on the float rounding mode you specify. By default, the rounding mode is `to-nearest`. See `Float rounding mode (-float-rounding-mode)`.

Nonfinite Floats Not Allowed

By default, nonfinite floats are not allowed. When the result of an operation falls outside the allowed range, an overflow occurs. The check is:

- Red, if the result of the operation falls outside the allowed range.
- Orange, if the result of the operation falls outside the allowed range on some of the execution paths.
- Green, if the result of the operation does not fall outside the allowed range.

To fine tune the behavior of the overflow check, use these options and specify argument `forbid`, `allow`, or `warn-with-wrap-around`:

- `Overflow mode for unsigned integer (-unsigned-integer-overflows)`
- `Overflow mode for signed integer (-signed-integer-overflows)`

The operand data types determine the allowed range for the arithmetic operation. If the operation involves two operands, the verification uses the ANSI C conversion rules to determine a common data type. This common data type determines the allowed range.

For some examples of conversion rules, see "Assumptions About Implicit Data Type Conversions" on page 4-22.

Nonfinite Floats Allowed

If you enable a verification mode that incorporates infinities and specify that the verification must warn about operations that produce infinities, the check is:

- Red, if the operation produces infinity on all execution paths that the software considers, and the operands themselves are not infinite.
- Orange, if the operation produces infinity on some of the execution paths when the operands themselves are not infinite.
- Green, if the operation does not produce infinity unless the operands themselves are infinite.

If you specify that the verification must forbid operations that produce infinities, the check color depends on the result of the operation only. The color does not depend on the operands.

To enable this verification mode, use these options:

- Consider non finite floats (-allow-non-finite-floats)
- Infinities (-check-infinite): Use argument `warn` or `forbid`.

Diagnosing This Check

"Review and Fix Overflow Checks"

Examples

Integer Overflow

```
void main() {
  int i=1;
  i = i << 30; //i = 2^30
  i = 2*i-2;
}
```

In this example, the operation $2*i$ results in a value 2^{31}. The **Overflow** check on the multiplication produces a red error because the maximum value that the type `int` can hold on a 32–bit target is 2^{31}-1.

Overflow Due to Left Shift on Signed Integers

```
void main(void)
 {
   int i;
   int shiftAmount = 1;

   i = 1090654225 << shiftAmount;
 }
```

In this example, an **Overflow** error occurs a left shift is performed on a signed integer.

Float Overflow

```
#include <float.h>

void main() {
 float val = FLT_MAX;
 val = val * 2 + 1.0;
}
```

In this example, FLT_MAX is the maximum value that `float` can represent on a 32-bit target. Therefore, the operation `val * 2` results in an **Overflow** error.

Overflow on Casts from Negative Floats to Unsigned Integers

```
void func(void) {
   float fVal = -2.0f;
   unsigned int iVal = (unsigned int)fVal;
}
```

In this example, a red **Overflow** check appears on the cast from `float` to `unsigned int`. According to the C99 Standard (footnote to paragraph 6.3.1.4), the range of values that can be converted from floating-point values to unsigned integers while keeping the code portable is (-1, MAX + 1). For

floating-point values outside this range, the conversion to unsigned integers is not well-defined. Here, MAX is the maximum number that can be stored by the unsigned integer type.

Even if a run-time error does not occur when you execute the code on your target, the cast might fail on another target.

Correction — Cast to Signed Integer First

One possible solution is to cast the floating-point value to a signed integer first. The signed integer can then be cast to an unsigned integer type. For these casts, the conversion rules are well-defined.

```
void func(void) {
    float fVal = -2.0f;
    int iValTemp = (int)fVal;
    unsigned int iVal = (unsigned int)iValTemp;
}
```

Negative Overflow

```
#define FLT_MAX 3.40282347e+38F

void float_negative_overflow() {
    float min_float = -FLT_MAX;
    min_float = -min_float * min_float;
}
```

In `float_negative_overflow`, `min_float` contains the most negative number that the type `float` can represent. Because the operation `-min_float * min_float` produces a number that is more negative than this number, the type `float` cannot represent it. The **Overflow** check produces a red error.

Overflows on Unsigned Bit Fields

```
#include <stdio.h>

struct
{
  unsigned int dayOfWeek : 2;
} Week;

void main()
{
  unsigned int two = 2, three = 3, four = 4;
  Week.dayOfWeek = two;
  Week.dayOfWeek = three;
  Week.dayOfWeek = four;
}
```

In this example, `dayOfWeek` occupies 2 bits. It can take values in `[0,3]` because it is an unsigned integer. When you assign the value 4 to `dayOfWeek`, the **Overflow** check is red.

To detect overflows on signed and unsigned integers, on the **Configuration** pane, under **Check Behavior**, select `forbid` or `warn-with-wrap-around` for **Overflow mode for signed integer** and **Overflow mode for unsigned integer** .

Nonfinite Floats: Infinity Detected with Red Check

Results in `forbid` mode:

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

In this example, both the operands of the / operation is not infinite but the result is infinity. The **Overflow** check on the - operation is red. In the `forbid` mode, the verification stops after the red check. For instance, a **Non-initialized local variable** check does not appear on x in the `return` statement. If you do not turn on the option **Allow non finite floats**, a **Division by zero** check appears because infinities are not allowed.

Results in `warn-first` mode:

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

In this example, both the operands of the / operation are not infinite but the result is infinity. The **Overflow** check on the - operation is red. The red checks in `warn-first` mode are different from red checks for other check types. The verification does not stop after the red check. For instance, a green **Non-initialized local variable** check appears on x in the `return` statement. In the verification result, if you place your cursor on x, you see that it has the value `Inf`.

Nonfinite Floats: Infinity Detected with Orange Check

Results in `forbid` mode:

```
void func(double arg1, double arg2) {
    double ratio1=arg1/arg2;
    double ratio2=arg1/arg2;
}
```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can have all possible `double` values. For instance, `arg1` can be nonzero and `arg2` can be zero and the result of `ratio1=arg1/arg2` can be infinity. Therefore, an orange **Overflow** check appears on the division operation. Following the check, the verification terminates the execution thread that results in infinity. The verification assumes that `arg2` cannot be zero following the orange check. The **Overflow** check on the second division operation `ratio2=arg1/arg2` is green.

Results in `warn-first` mode:

```
void func(double arg1, double arg2) {
    double ratio1=arg1/arg2;
    double ratio2=arg1/arg2;
}
```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can have all possible `double` values. For instance, `arg1` can be non-

zero and `arg2` can be zero and the result of `ratio1=arg1/arg2` can be infinity. An orange **Overflow** check appears on the division operation. The orange checks in `warn-first` mode are different from orange checks for other check types. Following the check, the verification does not terminate the execution thread that results in infinity. The verification retains the zero value of `arg2` following the orange check. Therefore, the **Overflow** check on the second division operation `ratio2=arg1/arg2` is also orange.

Check Information

Group: Numerical
Language: C | C++
Acronym: OVFL

See Also

`-show-similar-overflows` | `Consider non finite floats (-allow-non-finite-floats)` | `Infinities (-check-infinite)` | `Invalid operation on floats` | `Overflow mode for signed integer (-signed-integer-overflows)` | `Overflow mode for unsigned integer (-unsigned-integer-overflows)` | `Subnormal float`

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"
"Order of Code Prover Run-Time Checks"

Return value not initialized

C function does not return value when expected

Description

This check determines whether a function with a return type other than `void` returns a value. This check appears on every function call.

Diagnosing This Check

"Review and Fix Return Value Not Initialized Checks"

Examples

Function does not return value for given input

```
#include <stdio.h>
int input(void);
int inputRep(void);

int reply(int msg) {
  int rep = inputRep();
  if (msg > 0) return rep;
}

void main(void) {
  int ch = input(), ans;
  if (ch <= 0)
    ans = reply(0);
  else
    ans = reply(ch);
  printf("The answer is %d.",ans);
}
```

In this example, for the function call `reply(0)`, there is no return value. Therefore the **Return value not initialized** check returns a red error. The second call `reply(ch)` always returns a value. Therefore, the check on this call is green.

Correction — Return value for all inputs

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>
int input();
int inputRep();

int reply(int msg) {
  int rep = inputRep();
  if (msg > 0) return rep;
  return 0;
}
```

```
void main(void) {
  int ch = input(), ans;
  if (ch <= 0)
    ans = reply(0);
  else
    ans = reply(ch);
  printf("The answer is %d.",ans);
}
```

Function does not return value for some inputs

```
#include <stdio.h>
int input();
int inputRep(int);

int reply(int msg) {
  int rep = inputRep(msg);
  if (msg > 0) return rep;
}

void main(void) {
  int ch = input(), ans;
  if (ch < 10)
    ans = reply(ch);
  else
    ans = reply(10);
  printf("The answer is %d.",ans);
}
```

In this example, in the first branch of the `if` statement, the value of `ch` can be divided into two ranges:

- `ch < = 0`: For the function call `reply(ch)`, there is no return value.
- `ch > 0` and `ch < 10`: For the function call `reply(ch)`, there is a return value.

Therefore the **Return value not initialized** check returns an orange error on `reply(ch)`.

Correction — Return value for all inputs

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>
int input();
int inputRep(int);

int reply(int msg) {
  int rep = inputRep(msg);
  if (msg > 0) return rep;
  return 0;
}

void main(void) {
  int ch = input(), ans;
  if (ch < 10)
    ans = reply(ch);
  else
    ans = reply(10);
  printf("The answer is %d.",ans);
}
```

Check Information

Group: Data flow
Language: C
Acronym: IRV

See Also

`Disable checks for non-initialization (-disable-initialization-checks)|`
`Function not returning value`

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Subnormal float

Floating-point operation has subnormal results

Description

This check determines if a floating-point operation produces a subnormal result.

Subnormal numbers have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the significand. The presence of subnormal numbers indicates loss of significant digits. This loss can accumulate over subsequent operations and eventually result in unexpected values. Subnormal numbers can also slow down the execution on targets without hardware support.

By default, the results of the check do not appear in your verification results. To see the results of the check, change the default value of the option `Subnormal detection mode (-check-subnormal)`. The results of the check vary based on the detection mode that you specify. In all modes other than `allow`, to identify the subnormal results, look for red or orange **Subnormal float** checks on operations.

Mode	Check Colors	Behavior Following Check
`forbid`: This mode detects the occurrence of a subnormal value. This mode stops the execution path with the subnormal result and prevents subnormal values from propagating further. Therefore, in practice, you see only the first occurrence of the subnormal value.	The color of the check depends only on the result of the operation. The check flags an operation that has subnormal results even if those results come only from subnormal operands. For instance, if `x` is unknown, `x * 2` can be subnormal because `x` can be subnormal. The result of the check is orange.	Blocking check. If the check is red, the verification stops. If the check is orange, the verification removes the execution paths containing the subnormal result from consideration. For instance, the tooltip on the result does not show the subnormal values.
`warn-all`: This mode highlights all occurrences of subnormal values. Even if a subnormal result comes from previous subnormal values, the result is highlighted.	The color of the check depends only on the result of the operation. The check flags an operation that has subnormal results even if those results come only from subnormal operands. For instance, if `x` is unknown, `x * 2` can be subnormal because `x` can be subnormal. The result of the check is orange.	Non-blocking check. The verification continues even if the check is red. If the check is orange, the verification does not remove the execution paths containing the subnormal result from consideration.

Mode	Check Colors	Behavior Following Check
`warn-first:` This mode highlights the first occurrence of a subnormal value. If a subnormal value propagates to further subnormal results, those subsequent results are not highlighted.	The check color depends on the result of the operation and the operand values. The check does not flag a subnormal result if it comes only from subnormal operands. In this mode, the check is: • Red, if the operation produces subnormal results on all execution paths that the software considers, and the operands are not subnormal. • Orange, if the operation produces subnormal results on some of the execution paths when the operands are not subnormal. For instance, if `x` is unknown, `x * 0.5` can be subnormal even if `x` is not subnormal. • Green, if the operation does not produce subnormal results unless the operands are subnormal. For instance, even if `x` is unknown, `x * 2` cannot be subnormal unless `x` is subnormal.	Non-blocking check. The verification continues even if the check is red. If the check is orange, the verification does not remove the execution paths containing the subnormal result from consideration.

If you choose to check for subnormals, you can also identify from the tooltips whether a variable range excludes subnormal values. For instance, if the tooltips show `[-1.0 .. -1.1754E-38]` or `[-0.0..0.0] or [1.1754E-38..1.0]`, you can interpret that the variable does not have subnormal values.

Examples

Subnormal Results Detected with Red Checks

In the following examples, DBL_MIN is the minimum normal value that can be represented using the type `double`.

Results in `forbid` mode:

```
#include <float.h>

void func(){
    double val = DBL_MIN/4.0;
    double val2 = val * 2.0;
}
```

In this example, the first **Subnormal float** check is red because the result of DBL_MIN/4.0 is subnormal. The red check stops the verification. The following operation, val * 2.0, is not verified for run-time errors.

Results in warn-all mode:

```
#include <float.h>

void func(){
    double val = DBL_MIN/4.0;
    double val2 = val * 2.0;
}
```

In this example, both **Subnormal float** checks are red because both operations have subnormal results.

Results in warn-first mode:

```
#include <float.h>

void func(){
    double val = DBL_MIN/4.0;
    double val2 = val * 2.0;
}
```

In this example, DBL_MIN is not subnormal but the result of DBL_MIN/4.0 is subnormal. The first **Subnormal float** check is red. The second **Subnormal float** check is green. The reason is that val * 2.0 is subnormal only because val is subnormal. Through red/orange checks, you see only the first instance where a subnormal value appears. You do not see red/orange checks from those subnormal values propagating to subsequent operations.

Subnormal Results Detected with Orange Checks

In the following examples, arg1 and arg2 are unknown. The verification assumes that they can take all values allowed for the type double.

Results in forbid mode:

```
void func (double arg1, double arg2) {
      double difference1 = arg1 - arg2;
      double difference2 = arg1 - arg2;
      double val1 = difference1 * 2;
      double val2 = difference2 * 2;
}
```

In this example, difference1 can be subnormal if arg1 and arg2 are sufficiently close. The first **Subnormal float** check is orange. Following this check, the verification excludes from consideration the following:

- The close values of arg1 and arg2 that led to the subnormal value of difference1.

  In the subsequent operation arg1 - arg2, the **Subnormal float** check is green and difference2 is not subnormal. The result of the check on difference2 * 2 is green for the same reason.

- The subnormal value of difference1.

  In the subsequent operation difference1 * 2, the **Subnormal float** check is green.

Results in `warn-all` mode:

```
void func (double arg1, double arg2) {
      double difference1 = arg1 - arg2;
      double difference2 = arg1 - arg2;
      double val1 = difference1 * 2;
      double val2 = difference2 * 2;
}
```

In this example, the four operations can have subnormal results. The four **Subnormal float** checks are orange.

Results in `warn-first` mode:

```
void func (double arg1, double arg2) {
      double difference1 = arg1 - arg2;
      double difference2 = arg1 - arg2;
      double val1 = difference1 * 2;
      double val2 = difference2 * 2;
}
```

In this example, if `arg1` and `arg2` are sufficiently close, `difference1` and `difference2` can be subnormal. The first two **Subnormal float** checks are orange. `val1` and `val2` cannot be subnormal unless `difference1` and `difference2` are also subnormal. The last two **Subnormal float** checks are green. Through red/orange checks, you see only the first instance where a subnormal value appears. You do not see red/orange checks from those subnormal values propagating to subsequent operations.

Conversion of Floating Point Literals

```
void main() {
    float d = 1e-38;
    float e = 1e-38 - 1e-39;
}
```

In this example, the two red checks appear in both `warn-first` and `warn-all` mode (the `forbid` mode prevents analysis after the first red check).

Literal constants such as `1e-38` have the data type `double`. If you assign a literal constant to a variable with narrower type `float`, the constant might not be representable in this type. This issue is indicated with the red checks. The checks flag the conversion from `double` to `float` during assignment.

Result Information
Group: Numerical
Language: C | C++
Acronym: SUBNORMAL

See Also
`Invalid operation on floats` | `Overflow` | `Subnormal detection mode (-check-subnormal)`

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"
"Order of Code Prover Run-Time Checks"

Introduced in R2016b

Uncaught exception

Exception propagates uncaught to the `main` or another entry-point function

Description

This check looks for the following issues:

- An uncaught exception propagates to the `main` or another entry-point function.
- An exception is thrown in the constructor of a global variable and not caught.
- An exception is thrown in a destructor call or `delete` expression.
- An exception is thrown before a previous throw expression is handled by a `catch` statement, for instance, when constructing a `catch` statement parameters.
- A `noexcept` specification is violated. For instance, a function declared with `noexcept(true)` is not supposed to throw any exceptions but an exception is thrown in the function body.

In these situations, according to the C++ standard, the `std::terminate` function is called and can cause unexpected results.

Note that the **Uncaught exception** check on functions from the Standard Template Library is green, even though Polyspace stubs these functions and does not check if a function throws an exception.

Diagnosing This Check

"Review and Fix Uncaught Exception Checks"

Examples

Exception in call to function

```
#include <vector>
using namespace std;

class error {};

class initialVector {
private:
  int sizeVector;
  vector<int> table;
public:
  initialVector(int size) {
    sizeVector = size;
    table.resize(sizeVector);
    Initialize();
  }
  void Initialize();
  int getValue(int number) throw(error);
};

void initialVector::Initialize() {
  for(int i=0; i<table.size(); i++)
```

```
    table[i]=0;
  }

int initialVector::getValue(int index) throw(error) {
  if(index >= 0 && index < sizeVector)
    return table[index];
  else throw error();
}

void main() {
    initialVector *vectorPtr = new initialVector(5);
    vectorPtr->getValue(5);
}
```

In this example, the call to method `initialVector::getValue` throws an exception. This exception propagates uncaught to the `main` function resulting in a red **Uncaught exception** check.

Exception handled through `try/catch` construct

```
class error {
public:
  error() {   }
  error(const error&) { }
};


void funcNegative() {
  try {
    throw error() ;
  } catch (error NegativeError) {
  }
}

void funcPositive() {
  try {
  }
  catch (error PositiveError) {
  /* Gray code */
  }
}

int input();
void main()
{
    int val=input();
    if(val < 0)
        funcNegative();
    else
        funcPositive();
}
```

In this example:

- The call to `funcNegative` throws an exception. However, the exception is placed inside a `try` block and is caught by the corresponding handler (`catch` clause). The **Uncaught exception** check on the `main` function appears green because the exception does not propagate to the `main`.
- The call to `funcPositive` does not throw an exception in the `try` block. Therefore, the `catch` block following the `try` block appears gray.

Exception in call to destructor

```
class error {
};

class X
{
public:
  X() {
      ;
   }
   ~X() {
     throw error();
   }
};

int main() {
  try {
    X * px = new X ;
    delete px;
  } catch (error) {
    assert(1) ;
  }
}
```

In this example, the `delete` operator calls the destructor `X::~X()`. The destructor throws an exception that appears as a red error on the destructor body and dashed red on the `delete` operator. The exception does not propagate to the `catch` block. The code following the exception is not verified. This behavior enforces the requirement that a destructor must not throw an exception.

The black `assert` statement suggests that the exception has not propagated to the `catch` block.

Exception in infinite loop

```
#include<stdio.h>
#define SIZE 100

int arr[SIZE];
int getIndex();

int runningSum() {
  int index, sum=0;
  while(1) {
    index=getIndex();
    if(index < 0 || index >= SIZE)
      throw int(1);
    sum+=arr[index];
  }
}

void main() {
    printf("The sum of elements is: %d",runningSum());
}
```

In this example, the `runningSum` function throws an exception only if `index` is outside the range `[0,SIZE]`. Typically, an error that occurs due to instructions in an `if` statement is orange, not red. The error is orange because an alternate execution path that does not involve the `if` statement does

not produce an error. Here, because the loop is infinite, there is no alternate execution path that goes outside the loop. The only way to go outside the loop is through the exception in the `if` statement. Therefore, the **Uncaught exception** error is red.

Rethrow outside `catch` block

```
#include <string>

void f() { throw; }        //rethrow not allowed - an error is raised here
void main() {
    try {
        throw std::string("hello");
     }
    catch (std::string& exc) {
        f();
    }
}
```

In this example, an exception is rethrown in the function `f()` outside a `catch` block. A rethrow occurs when you call `throw` by itself without an exception argument. A rethrow is typically used *inside* a `catch` block to propagate an exception to an outer `try-catch` sequence. Polyspace Code Prover does not support a rethrow *outside* a `catch` block and produces a red **Uncaught exception** error.

Check Information
Group: C++
Language: C++
Acronym: EXC

See Also

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Unreachable code

Code cannot be reached during execution

Description

Unreachable code uses statement coverage to determine whether a section of code can be reached during execution. Statement coverage checks whether a program statement is executed. If a statement has test conditions, and at least one of them occurs, the statement is executed and reachable. The test conditions that do not occur are not considered dead code unless they have a corresponding code branch. If all the test conditions do not occur, the statement is not executed and each test condition is an instance of unreachable code. For example, in the `switch` statements of this code, `case 3` never occurs:

```
void test1 (int a) {
    int tmp = 0;
    if ((a!=3)) {
        switch (a){
            case 1:
                tmp++;
                break;
            default:
                tmp = 1;
                break;
/* case 3 falls through to
   case 2, no dead code */
            case 3:
            case 2:
                tmp = 100;
                break;
        }
    }
}

void test2 (int a) {
    int tmp = 0;
    if ((a!=3)) {
        switch (a){
            case 1:
                tmp++;
                break;
            default:
                tmp = 1;
                break;
// Dead code on case 3
            case 3:
                break;
            case 2:
                tmp = 100;
                break;
        }
    }
}
```

In `test1()`, `case` 3 falls through to `case` 2 and the check shows no dead code. In `test2()`, the check shows dead code for `case` 3 because the `break` statement on the next line is not executed.

Other examples of unreachable code include:

- If a test condition always evaluates to false, the corresponding code branch cannot be reached. On the **Source** pane, the opening brace of the branch is gray.
- If a test condition always evaluates to true, the condition is redundant. On the **Source** pane, the condition keyword, such as `if`, appears gray.
- The code follows a `break` or `return` statement.

If an opening brace of a code block appears gray on the **Source** pane, to highlight the entire block, double-click the brace.

The check operates on code inside a function. The checks **Function not called** and **Function not reachable** determine if the function itself is not called or called from unreachable code.

Diagnosing This Check

"Review and Fix Unreachable Code Checks"

Examples

Test in `if` Statement Always False

```
#define True 1
#define False 0

typedef enum {
  Intermediate, End, Wait, Init
} enumState;

enumState input();
enumState inputRef();
void operation(enumState, int);

int checkInit (enumState stateval)  {
  if (stateval == Init)
    return True;
  return False;
}

int checkWait (enumState stateval)  {
  if (stateval == Wait)
    return True;
  return False;
}

void main()  {
  enumState myState = input(),refState = inputRef() ;
  if(checkInit(myState)){
    if(checkWait(myState)) {
      operation(myState,checkInit(refState));
    } else {
      operation(myState,checkWait(refState));
```

```
      }
    }
}
```

In this example, the `main` enters the branch of `if(checkInit(myState))` only if `myState = Init`. Therefore, inside that branch, Polyspace considers that `myState` has value `Init`. `checkWait(myState)` always returns `False` and the first branch of `if(checkWait(myState))` is unreachable.

Correction — Remove Redundant Test

One possible correction is to remove the redundant test `if(checkWait(myState))`.

```
#define True 1
#define False 0

typedef enum {
  Intermediate, End, Wait, Init
} enumState;

enumState input();
enumState inputRef();
void operation(enumState, int);

int checkInit (enumState stateval)  {
  if (stateval == Init)
    return True;
  return False;
}

int checkWait (enumState stateval)  {
  if (stateval == Wait) return True;
  return False;
}

void main()  {
  enumState myState = input(),refState = inputRef() ;
  if(checkInit(myState))
    operation(myState,checkWait(refState));
}
```

Test in `if` Statement Always True

```
#include <stdlib.h>
#include <time.h>

int roll() {
  return(rand()%6+1);
}

void operation(int);

void main()    {
    srand(time(NULL));
    int die = roll();
    if(die >= 1 && die <= 6)
      /*Unreachable code*/
      operation(die);
  }
```

In this example, `roll()` returns a value between 1 and 6. Therefore the `if` test in `main` always evaluates to true and is redundant. If there is a corresponding `else` branch, the gray error appears on the `else` statement. Without an `else` branch, the gray error appears on the `if` keyword to indicate the redundant condition.

Correction — Remove Redundant Test

One possible correction is to remove the condition `if(die >= 1 && die <=6)`.

```
#include <stdlib.h>
#include <time.h>

int roll() {
  return(rand()%6+1);
}

void operation(int);

void main()   {
  srand(time(NULL));
  int die = roll();
  operation(die);
}
```

Test in `if` Statement Unreachable

```
#include <stdlib.h>
#include <time.h>
#define True 1
#define False 0

int roll1() {
  return(rand()%6+1);
}

int roll2();
void operation(int,int);

void main()    {
  srand(time(NULL));
  int die1 = roll1(),die2=roll2();
  if((die1>=1 && die1<=6) ||
     (die2>=1 && die2 <=6))
  /*Unreachable code*/
    operation(die1,die2);
}
```

In this example, `roll1()` returns a value between 1 and 6. Therefore, the first part of the `if` test, `if((die1>=1) && (die1<=6))` is always true. Because the two parts of the `if` test are combined with `||`, the `if` test is always true irrespective of the second part. Therefore, the second part of the `if` test is unreachable.

Correction — Combine Tests with &&

One possible correction is to combine the two parts of the `if` test with && instead of ||.

```
#include <stdlib.h>
#include <time.h>
```

```
#define True 1
#define False 0

int roll1() {
  return(rand()%6+1);
}

int roll2();
void operation(int,int);

void main()    {
  srand(time(NULL));
  int die1 = roll1(),die2=roll2();
  if((die1>=1 && die1<=6) &&
     (die2>=1 && die2<=6))
    operation(die1,die2);
}
```

Check Information

Group: Data flow
Language: C | C++
Acronym: UNR

See Also

`Function not called` | `Function not reachable`

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"

User assertion

`assert` statement fails

Description

This check determines whether the argument to an `assert` macro is true.

The argument to the `assert` macro must be true when the macro executes. Otherwise the program aborts and prints an error message. Polyspace models this behavior by treating a failed `assert` statement as a run-time error. This check allows you to detect failed `assert` statements before program execution.

Diagnosing This Check

"Review and Fix User Assertion Checks"

Examples

Red assert on array index

```
#include<stdio.h>
#define size 20

int getArrayElement();

void initialize(int* array) {
 for(int i=0;i<size;i++)
   array[i] = getArrayElement();
}

void printElement(int* array,int index) {
 assert(index < size);
 printf("%d", array[index]);
}

int getIndex() {
 int i = size;
 return i;
}

void main() {
 int array[size];
 int index;

 initialize(array);
 index = getIndex();
 printElement(array,index);

}
```

In this example, the `assert` statement in `printElement` causes program abort if `index >= size`. The `assert` statement makes sure that the array index is not outside array bounds. If the code does

not contain exceptional situations, the `assert` statement must be green. In this example, `getIndex` returns an index equal to `size`. Therefore the `assert` statement appears red.

Correction — Correct cause of `assert` failure

When an `assert` statement is red, investigate the cause of the exceptional situation. In this example, one possible correction is to force `getIndex` to return an index equal to `size-1`.

```c
#include<stdio.h>
#define size 20

int getArrayElement();

void initialize(int* array) {
 for(int i=0;i<size;i++)
  array[i] = getArrayElement();
}

void printElement(int* array,int index) {
 assert(index < size);
 printf("%d", array[index]);
}

int getIndex() {
 int i = size;
 return (i-1);
}

void main() {
 int array[size];
 int index;

 initialize(array);
 index = getIndex();
 printElement(array,index);

}
```

Orange assert on `malloc` return value

```c
#include <stdlib.h>

void initialize(int*);
int getNumberOfElements();

void main() {
 int numberOfElements, *myArray;

 numberOfElements = getNumberOfElements();

 myArray = (int*)malloc(numberOfElements);
 assert(myArray!=NULL);

 initialize(myArray);
}
```

In this example, `malloc` can return `NULL` to `myArray`. Therefore, `myArray` can have two possible values:

- `myArray == NULL`: The `assert` condition is false.
- `myArray != NULL`: The `assert` condition is true.

Combining these two cases, the **User assertion** check on the `assert` statement is orange. After the orange `assert`, Polyspace considers that `myArray` is not equal to `NULL`.

Correction — Check return value for NULL

One possible correction is to write a customized function `myMalloc` where you always check the return value of `malloc` for `NULL`.

```
#include <stdio.h>
#include <stdlib.h>

void initialize(int*);
int getNumberOfElements();

void myMalloc(int **ptr, int num) {
 *ptr = (int*)malloc(num);
 if(*ptr==NULL) {
    printf("Memory allocation error");
    exit(1);
  }
}

void main() {
 int numberOfElements, *myArray=NULL;

 numberOfElements = getNumberOfElements();

 myMalloc(&myArray,numberOfElements);
 assert(myArray!=NULL);

 initialize(myArray);
}
```

Imposing constraint through orange assert

```
#include<stdio.h>
#include<math.h>

float getNumber();
void squareRootOfDifference(float firstNumber, float secondNumber) {
    assert(firstNumber > secondNumber);
    if(firstNumber > 0 && secondNumber > 0)
    printf("Square root = %.2f",sqrt(firstNumber-secondNumber));
}

void main() {
    double firstNumber = getNumber(), secondNumber = getNumber();
    squareRootOfDifference(firstNumber,secondNumber);
}
```

In this example, the `assert` statement in `squareRootOfDifference()` causes program abort if `firstNumber` is less than `secondNumber`. Because Polyspace does not have enough information about `firstNumber` and `secondNumber`, the `assert` is orange.

Following the `assert`, all execution paths that cause assertion failure terminate. Therefore, following the `assert`, Polyspace considers that `firstNumber >= secondNumber`. The **Invalid use of standard library routine** check on `sqrt` is green.

Use `assert` statements to help Polyspace determine:

- Relationships between variables
- Constraints on variable ranges

Check Information
Group: Other
Language: C | C++
Acronym: ASRT

See Also

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"
"Code Prover Analysis Following Red and Orange Checks"

Approximations Used During Verification

Why Polyspace Verification Uses Approximations

Polyspace Code Prover uses *static verification* to prove the absence of run-time errors. Static verification derives the dynamic properties of a program without actually executing it. Static verification differs significantly from other techniques such as run-time debugging because the verification does not rely on a specific test case or set of test cases. The properties obtained from static verification are true for *all* executions of your program[1].

Static verification uses representative approximations of software operations and data. For instance, consider the following code:

```
for (i=0 ; i<1000 ; ++i)  {
    tab[i] = foo(i);
}
```

To check that the variable i never overflows the range of `tab`, one approach can be to consider each possible value of i. This approach requires a thousand checks.

In static verification, the software models a variable by its domain. In this case, the software models that i belongs to the static interval, [0..999]. Depending on the complexity of the data, the software uses more elaborate models such as convex polyhedrons or integer lattices for this purpose.

An approximation, by definition, leads to information loss. For instance, the verification loses the information that i is incremented by one every cycle in the loop. However, even without this information, it is possible to ensure that the range of i is smaller than the range of `tab`. Only one check is required to establish this property. Therefore, static verification is more efficient compared to traditional approaches.

When performing approximations, the verification does not compromise with exhaustiveness. The reason is that the approximations performed are upper approximations or over-approximations. In other words, the computed domain of a variable is a superset of its actual domain.

---

1. The properties obtained from static verification hold true only if you execute your program under the same conditions that you specified through the analysis options. For instance, the default verification assumes that pointers obtained from external sources are non-null. Unless you specify the option `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`, the verification results are obtained under this assumption. They might not hold true during program execution if the assumption is invalidated and a null pointer is obtained from an external source.

Sources of Orange Checks

The **Orange Sources** pane shows unconstrained sources such as volatile variables and stubbed functions that can lead to multiple orange checks (unproven results) in a Code Prover analysis. If you constrain an orange source, you can address several orange checks together. To see the **Orange Sources** pane, click the ⟨?⟩ button on the **Result Details** pane.

The sources essentially indicate variables whose values cannot be determined from your code. The variables can be inputs to functions whose call context is unknown or return values of undefined functions. Code Prover assumes that these variables take the full range of values allowed by their data type. This broad assumption can lead to one or more orange checks in the subsequent code.

Source Type	Name	File	Line	Max Oranges	Suggestion
stubbed function	get_bus_status()			1	Add Constraints
stubbed function	random_float()			3	Add Constraints
stubbed function	random_int()			1	Add Constraints
local volatile variable	tmps32	single_file_analysis.c	29	2	Open Editor
local volatile variable	tmpu16	single_file_analysis.c	31	2	Open Editor
local volatile variable	vol_i	example.c	27	2	Open Editor
local volatile variable	PORT_A	main.c	48	3	Open Editor
local volatile variable	PORT_B	main.c	49	3	Open Editor

For instance, in this example, if the function `random_float` is not defined, you see three orange **Overflow** checks.

```
static void Close_To_Zero(void)
{
    float xmin = random_float();
    float xmax = random_float();
    float y;

    if ((xmax - xmin) < 1.0E-37f) { /* Overflow 1 */
        y = 1.0f;
    } else {
        /* division by zero is impossible here */
        y = (xmax + xmin) / (xmax - xmin); /* Overflows 2 and 3 */
    }
}
```

The function `random_float` is therefore an orange source that causes at most three orange checks.

Using the **Orange Sources** pane, you can:

- Review all orange checks originating from the same source.

  In the preceding example, if you select the function `random_float`, the results list shows only the three orange checks caused by this source. See "Filter Using Orange Sources".

- Constrain variable ranges by specifying external constraints or through additional code. Constraining the range of an orange source can remove several orange checks that come from overapproximation. The remaining orange checks indicate real issues in your code.

In the preceding example, you can constrain the return value of `random_float`.

For efficient review, click the **Max Oranges** column header to sort the orange sources by number of orange checks that result from the source. Constrain the sources with more orange checks before tackling the others.

Constrain Orange Sources

How you constrain variable ranges and work around the default Polyspace assumptions depends on the type of orange source:

Stubbed function

If the definition of a function is not available to the Polyspace analysis, the function is stubbed. The analysis makes several assumptions about stubbed functions. For instance, the return value of a stubbed function can take any value allowed by its data type.

See "Assumptions About Stubbed Functions" on page 4-7 for assumptions about stubbed functions and how to work around them.

Volatile variable

If a variable is declared with the `volatile` specifier, the analysis assumes that the variable can take any value allowed by its data type at any point in the code.

See "Assumptions About Volatile Variables" on page 4-19 to work around around this assumption.

Extern variable

If a variable is declared with the `extern` specifier but not defined elsewhere in the code, the analysis assumes that the variable can take any value within its data type range before it is first assigned.

Determine where the variable is defined and why the definition is not available to the analysis. For instance, you might have omitted an include folder from the analysis.

Function called by the main generator

If your code does not contain a `main` function, a `main` function is generated for the analysis. By default, the generated `main` function calls uncalled functions with inputs that can take any value allowed by their data type.

See:

- "Constrain Function Inputs" to constrain the function inputs.
- "Verify C Application Without main Function" or "Verify C++ Classes" to change which functions are called by the generated `main`.

Variable initialized by the main generator

If your code does not contain a `main` function, a `main` function is generated for the analysis. By default, in each function called by the generated `main`, a global variable can take any value within its data type range before it is first assigned.

See "Assumptions About Global Variable Initialization" on page 4-15 for how the generated `main` initializes global variables.

Variable set in a permanent range by the main generator

If you explicitly constrain a global variable to a specific range in the permanent mode, the analysis assumes that the variable can take any value within this range at any point in the code.

See "External Constraints for Polyspace Analysis" for more information on how a variable gets a permanent range. Check if you assigned a permanent range by mistake or your range must be narrower to reflect real-world values.

Absolute address

If a pointer is assigned an absolute address, the analysis assumes that the pointer dereference leads to a range of potential values determined by the pointer data type.

See `Absolute address usage` for examples of absolute address usage and corresponding Code Prover assumptions. To remove this assumption and flag all uses of absolute address, use the option `-no-assumption-on-absolute-addresses`.

Sometimes, more than one orange source can be responsible for an orange check. If you plug an orange source but do not see the expected disappearance of an orange check, consider if another source is also responsible for the check.

See Also

More About

- "Orange Checks in Code Prover"
- "Filter Using Orange Sources"
- "Reduce Orange Checks"

Assumptions About Variable Ranges

If Polyspace cannot determine a variable value from the code, it assumes that the variable has a full range of values allowed by its type.

For instance, for a variable of integer type, to determine the minimum and maximum value allowed, Polyspace uses the following criteria:

- The C standard specifies that the range of a signed *n*-bit integer-type variable must be at least $[-(2^{n-1}-1), 2^{n-1}-1]$.

  The **Target processor type** that you specify determines the number of bits allocated for a certain type. For more information, see `Target processor type (-target)`.

- Polyspace assumes that your target uses the two's complement representation for signed integers. The software uses this representation to determine the exact range of a variable. In this representation, the range of a signed *n*-bit integer-type variable is $[-2^{n-1}, 2^{n-1}-1]$.

For example, for an `i386` processor:

- A `char` variable has 8 bits. The C standard specifies that the range of the `char` variable must be at least [-127,127].
- Using the two's complement representation, Polyspace assumes that the exact range of the `char` variable is [-128,127].

To determine the range that Polyspace assumes for a certain type:

1. Run verification on this code. Replace *type* with the type name such as `int`.

   ```
   type getVal(void);
   void main() {
           type val = getVal();
   }
   ```

2. Open your verification results. On the **Source** pane, place your cursor on `val`.

   The tooltip provides the range that Polyspace assumes for *type*. Since `getVal` is not defined, Polyspace assumes that the return value of `getVal` has full range of values allowed by *type*.

Assumptions About Stubbed Functions

The verification stubs functions that are not defined in your source code or that you choose to stub. For a stubbed function:

- The verification makes certain assumptions about the function return value and other side effects of the function.

  You can fine-tune the assumptions by specifying constraints.
- The verification ignores the function body if it exists. Operations in the function body are not checked for run-time errors.

If the verification of a function body is imprecise and causes many orange checks when you call the function, you can choose to stub the function. To reduce the number of orange checks, you stub the function, and then constrain the return value of the function and specify other side effects.

To stub functions, you can use these options:

- `Functions to stub (-functions-to-stub)`: Specify functions that you want stubbed.
- `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`: Stub functions that contain lookup tables in code generated from models using Embedded Coder.
- `-code-behavior-specifications`: Stub functions that correspond to a standard function that Polyspace recognizes.

If you use the first option to stub a function, you constrain the function return value and model other side effects by specifying constraints. If you want to specify constraints more fine-grained than the ones available through the Polyspace constraint specification interface, define your own stubs. If you use the other options to stub functions, the software itself constrains the function return value and models its side effects appropriately.

The verification makes the following assumptions about the arguments of stubbed functions.

Function Return Value

Assumptions

The verification assumes that:

- The variable returned by the function takes the full range of values allowed by its data type.

  If the function returns an `enum` variable, the variable value is in the range of the enum. For instance, if an `enum` type takes values {0,5,-1,32} and a stubbed function has that return type, the verification assumes that the function returns values in the range -1..32.
- If the function returns a pointer, the pointer is not NULL and safe to dereference. The pointer does not point to dynamically allocated memory or another variable in your code.
- C++ specific assumptions: The operator `new` returns allocated memory. Operators such as `operator=`, `operator+=`, `operator--`(prefixed version) or `operator<<` returns:

  - A reference to `*this`, if the operator is part of a class definition.

    For instance, if an operator is defined as:

```
class X {
  X& operator=(const X& arg) ;
};
```

It returns a reference to *this (the object that calls the operator). The object that calls the operator or its data members have the full range of values allowed by their type.

- The first argument, if the operator is not part of a class definition.

  For instance, if an operator is defined as:

  ```
  X& operator+=(X& arg1, const X& arg2) ;
  ```

  It returns arg1. The object that arg1 refers to or its data members have the full range of values allowed by their type.

Functions declared with __declspec(no_return) (Visual Studio) or __attribute__ ((noreturn)) (GCC) do not return.

How to Change Assumptions

You can change the default assumptions about the function return value.

- If the function returns a non-pointer variable, you can constrain its range. Use the option Constraint setup (-data-range-specifications).

  Through the constraint specification interface, you can specify an absolute range [*min..max*]. To specify more complicated constraints, write a function stub.

  For instance, an undefined function has the prototype:

  ```
  int func(int ll, int ul);
  ```

  Suppose you know that the function return value lies between the first and the second arguments. However, the software assumes full range for the return value because the function is not defined. To model the behavior that you want and reduce orange checks from the imprecision, write a function stub as follows:

  ```
  int func(int ll, int ul) {
      int ret;
      assert(ret>=ll && ret <=ul);
      return ret;
  }
  ```

  Provide the function stub in a separate file for verification. The verification uses your stub as the function definition.

  If the definition of func exists in your code and you want to override the definition because the verification of the function body is imprecise, embed the actual definition and the stub in a #ifdef statement:

  ```
  #ifdef POLYSPACE
  int func(int ll, int ul) {
      int ret;
      assert(ret>=ll && ret <=ul);
      return ret;
  }
  #else
  ```

```
int func(int ll, int ul) {
    /*Your function body */
}
#endif
```

Define the macro `POLYSPACE` by using the option `Preprocessor definitions (-D)`. The verification uses your stub instead of the actual function definition.

- If the function returns a pointer variable, you can specify that the pointer might be NULL.

    - To specify this assumption for all stubbed functions, use the option `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.

    - To specify this assumption for specific stubbed functions, use the option `Constraint setup (-data-range-specifications)`.

Function Arguments That are Pointers

Assumptions

The verification assumes that:

- If the argument is a pointer, the function can write any value to the object that the pointer points to. The range of values is constrained by the argument data type alone.

    For instance, in this example, the verification assumes that the stubbed function `stubbedFunc` writes any possible value to `val`. Therefore, the assertion is orange.

    ```
    void stubbedFunc(int*);

    void main() {
        int val=0, *ptr=&val;
        stubbedFunc(ptr);
        assert(val==0);
    }
    ```

- If the argument is a pointer to a structure, the function can write any value to the structure fields. The range of values is constrained only by the data type of the fields.

    In C++ code, only first level data members of a structure can be written via a pointer to the structure. For instance, in this example, the analysis has knowledge of what `pb->j` points to, but not what `pb->pa->i` points to. So, after the call to `Foo`, `b.j` appears as initialized but `a.i` is not initialized.

    ```
    struct A {
        int i;
    };

    struct B {
        A* pa;
        int j;
    };

    void Foo(B*);

    void main() {
        A a;
        B b;
    ```

```
        b.pa = &a;
        Foo(&b);
        int var1 = b.j;
        int var2 = a.i;
    }
```

- If the argument is a pointer to another pointer, the function can write any value to the object that the second pointer points to (C code only). This assumption continues to arbitrary depths of a pointer hierarchy.

  For instance, suppose that a pointer **pp points to another pointer *p, which points to an int variable var. If a stubbed function takes **p as argument, the verification assumes that following the function call, var has any int value. *p can point to anywhere in allocated memory or can point to var but does not point to another variable in the code.

- If the argument is a function pointer, the function that it points to gets called (C code only).

  For instance, in this example, the stubbed function stubbedFunc takes a function pointer funcPtr as argument. funcPtr points to func, which gets called when you call stubbedFunc.

  ```
  typedef int (*typeFuncPtr) (int);

  int func(int x){
        return x;
  }

  int stubbedFunc(typeFuncPtr);

  void main() {
      typeFuncPtr funcPtr = (typeFuncPtr)(&func);
      int result = stubbedFunc(funcPtr);
  }
  ```

  If the function pointer takes another function pointer as argument, the function that the second function pointer points to gets stubbed.

How to Change Assumptions

You can constrain the range of the argument that is passed by reference. Use the option Constraint setup (-data-range-specifications).

Through the constraint specification interface, you can specify an absolute range [*min..max*]. To specify more complicated constraints, write a function stub.

For instance, an undefined function has the prototype:

```
void func(int *x, int ll, int ul);
```

Suppose you know that the value written to x lies between the second and the third arguments. However, the software assumes full range for the value of *x because the function is not defined. To model the behavior that you want and reduce orange checks from the imprecision, write a function stub as follows:

```
void func(int *x, int ll, int ul) {
    assert(*x>=ll && *x <=ul);
}
```

Provide the function stub in a separate file for verification. The verification uses your stub as the function definition.

If the definition of `func` exists in your code and you want to override the definition because the verification of the function body is imprecise, embed the actual definition and the stub in a `#ifdef` statement:

```
#ifdef POLYSPACE
void func(int *x, int ll, int ul) {
    assert(*x>=ll && *x <=ul);
}
#else
void func(int *x, int ll, int ul) {
    /* Your function body */
}
#endif
```

Define the macro `POLYSPACE` by using the option `Preprocessor definitions (-D)`. The verification uses your stub instead of the actual function definition.

Global Variables

Assumptions

The verification assumes that the function stub does not modify global variables.

How to Change Assumptions

To model write operations on a global variable, write a function stub.

For instance, an undefined function has the prototype:

```
void func(void);
```

Suppose you know that the function writes the value 0 or 1 to a global variable `glob`. To model the behavior that you want, write a function stub as follows:

```
void func(void) {
    volatile int randomVal;
    if(randomVal)
        glob = 0;
    else
        glob = 1;
}
```

Provide the function stub in a separate file for verification. The verification uses your stub as the function definition.

If the definition of `func` exists in your code and you want to override the definition because the verification of the function body is imprecise, embed the actual definition and the stub in a `#ifdef` statement as follows:

```
#ifdef POLYSPACE
void func(void) {
    volatile int randomVal;
    if(randomVal)
        glob = 0;
    else
        glob = 1;
}
```

```
#else
void func(void) {
    /* Your function body */
}
#endif
```

Define the macro `POLYSPACE` using the option `Preprocessor definitions (-D)`. The verification uses your stub instead of the actual function definition.

Assumptions About main Function

A C/C++ program that compiles into a complete executable contains a `main` function. A Code Prover analysis treats the `main` function differently from other functions.

main Function as Top of Call Hierarchy

Code Prover considers the `main` function as the starting point of verification. If you do not provide a `main` function, for instance, when verifying a library, the verification generates one. By default, the generated `main` calls functions that are not called anywhere else.

The verification then proceeds from the `main` function onwards into the functions called from `main` and so on down the call hierarchy.

To adjust the verification time or precision:

- You can change the content of the generated `main` using analysis options.

  See "Code Prover Verification".

- You can write your own `main` function that calls only the functions that you want to verify.

main Function Arguments

The `main` function can have one of three forms:

- The no-argument form:

  `int main() {}`

- The two-argument form:

  `int main(int argc, char* argv[]) {}`

- Any other implementation-defined form.

In keeping with the C/C++ Standard specifications, the verification makes certain assumptions on the `main` function arguments. If the `main` function has arguments with data types that match the second form, Code Prover assumes this form and imposes corresponding restrictions on the arguments. In particular:

- If the first argument of the `main` function is an integer (or a `typdef` to integer), the verification assumes that the argument is nonnegative. This argument denotes the number of additional arguments to the program from its external environment.

  The assumption holds true even when the `main` function has only that one argument.

- If the first argument is an integer and the second argument is a pointer to a pointer (or a `typedef` to one), the verification assumes that the second argument is allocated a buffer of size equal to the first argument, `argc`. Each element of the buffer, `argv[0]`, `argv[1]`,...,`argv[argc-1]` is also assumed to be an initialized pointer. This argument stores the additional arguments to the program from its external environment.

See Also

More About

- "Code Prover Verification"

Assumptions About Global Variable Initialization

Global variables are variables that are visible throughout the program (unless shadowed by local variables). A Code Prover analysis makes specific assumptions about the initialization of global variables.

Global Variable Initialization When main Function Exists

If your code contains a `main` function, a Code Prover verification considers that global variables are initialized according to ANSI C standards. The default values are:

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

and so on.

Sometimes, you might want to check if global variables are explicitly initialized in the code. For instance:

- In a warm reboot, to save time, the bss segment of a program, which might hold variable values from a previous state, is not loaded. Instead, the program is supposed to explicitly initialize all non-const variables without default values before execution. You can delimit this initialization code and verify that all non-const global variables are indeed initialized in a warm reboot.

  To delimit a section of code as initialization code, enter the pragma `polyspace_end_of_init` in the `main` function. The initialization code begins from the `main` function and continues up to this pragma. Use these options to check the initialization code only and determine whether all global variables are initialized in this section of the code:

  - `Check that global variables are initialized after warm reboot (-check-globals-init)`
  - `Verify initialization section of code only (-init-only-mode)`

  The Code Prover analysis reports non-initialized variables using red or orange results in the initialization code for the checks:

  - `Global variable not assigned a value in initialization code`
  - `Non-initialized variable`

- To only check if global variables are explicitly initialized at the point of use, use the option `Ignore default initialization of global variables (-no-def-init-glob)`.

  The Code Prover analysis reports non-initialized variables using red or orange results for the check `Non-initialized variable`.

Global Variable Initialization When main Function Does Not Exist

If your code does not have a `main` function, Code Prover begins verifying each *uncalled* function with the assumption that global variables have full range value, constrained only by their data type. See also "Assumptions About Variable Ranges" on page 4-6.

For instance, consider this example:

```
int glob;
void func1_callee();

void func1() {
    int loc =  glob;
    if(!glob)
        func1_callee();
}

void func1_callee() {
    int loc = glob;
}

void func2() {
    int loc = glob;
}
```

In both `func1` and `func2`, the global variable `glob` and consequently the local variable `loc` has full range of `int` values.

However, only uncalled functions begin with full-range values of global variables. The function `func1_callee` is called in `func1` after the value of `glob` is constrained to zero. In `func1_callee`, the global variable `glob` and consequently the local variable `loc` has the constrained value zero.

How Code Prover Implements Assumption About Global Variable Initialization

The software uses the dummy function `_init_globals()` to initialize global variables. The `_init_globals()` function is the first function implicitly called in the `main` function (or generated `main` function if there is no `main`).

Consider the following code in the application `gv_example.c`.

```
extern int func(int);

int garray[3] = {1, 2, 3};
int gvar = 12;

int main(void) {
    int i, lvar = 0;
    for (i = 0; i < 3; i++)
        lvar += func(garray[i] + gvar);
    return lvar;
}
```

After verification:

- On the **Results List** pane, if you select **File** from the list, under the node `gv_example.c`, you see `_init_globals`.

- On the **Variable Access** pane, `gv_example._init_globals` represents the initialization of the global variable. The **Values** column shows the value of the global variable immediately after initialization.

What Initialization Means for Complex Data Types

The following table lists what is checked for each data type to determine initialization. The check happens at the time of read operations for the check **Non-initialized variable** and at the end of the initialization section for the check **Global variable not assigned a value in initialization code**.

Data Type	What Green Check for Initialization Means
Fundamental types (`int`, `double`, etc.)	The variable is written at least once.
Array data types	Every array element is written at least once.
Structured data types	Every structure field that is used is written at least once. If you check initialization code only using the option `Verify initialization section of code only (-init-only-mode)`, the analysis checks for initialization of all structure fields, whether used or not. In the special case where none of the fields are used, the checks for initialization are orange instead of green if all the fields.are uninitialized.

Data Type	What Green Check for Initialization Means
Pointers	The pointer is written at least once. However, Code Prover does not check for initialization of the pointed buffer (till you dereference the pointer).
Enumerations	The enum variable is written at least once. However, Code Prover does not check if the variable has one of the enum values.

See Also

Check that global variables are initialized after warm reboot (-check-globals-init)|Global variable not assigned a value in initialization code| Non-initialized variable|Verify initialization section of code only (-init-only-mode)

Assumptions About Volatile Variables

The values of volatile variables can change without explicit write operations.

For local volatile variables:

- Polyspace assumes that the variable has a full range of values allowed by its type.
- Unless you explicitly initialize the variable, when you read the variable, Polyspace produces an orange **Non-initialized local variable** check.

In this example, Polyspace assumes that `val1` is potentially noninitialized but `val2` is initialized. Polyspace considers that the + operation can cause an overflow because it assumes both variables to have all possible values allowed by their data types.

```
int func (void)
{
    volatile int val1, val2=0;
    return( val1 + val2);
}
```

For global volatile variables:

- Polyspace assumes that the variable has a full range of values allowed by its type.

  You can constrain the range externally. See "Constrain Global Variable Range".
- Even if you do not explicitly initialize the variable, when you read the variable, Polyspace produces a green **Non-initialized variable** check.

If the root cause of an orange check is a local volatile variable, you cannot override the default assumptions and constrain the values of the volatile variables. Try one of the following:

- If the volatile variable represents hardware-supplied data, see if you can use a function call to model this data retrieval. For example, replace `volatile int port_A` with `int port_A = read_location()`. You do not have to define the function. Polyspace stubs the undefined functions. You can then specify constraints on the function return values using the option `Constraint setup (-data-range-specifications)`.
- See if you can copy the contents of the volatile variable to a global nonvolatile variable. You can then constrain the global variable values throughout your code. See "Constrain Global Variable Range".
- Replace the volatile variable with a stubbed function, but only for verification. Before verification, specify constraints on the stubbed functions.

  1   Write a Perl script that replaces each volatile variable declaration with a nonvolatile declaration where you obtain the variable value from a function call.

      For example, if your code contains the line `volatile s8 PORT_A`, your Perl script can contain this substitution:

      `$line=~ s/^\s*volatile\s*s8\s*PORT_A;/s8 PORT_A = random_s8();/g;`

  2   Specify the location of this Perl script for the analysis option `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

  3   In an include file, provide the function declaration. For example, for a function `random_s8`, the include file can contain the following declaration:

```
#ifndef POLYSPACE_H
#define POLYSPACE_H
signed char random_s8(void);
#endif
```

4 Insert a `#include` directive for your include file in the relevant source files

Instead of a manual insertion, specify the location of your include file for the analysis option `Include (-include)`.

Assumptions About Variable and Function Definitions and Declarations

The definition and declaration of a variable are two different but related operations.

Definition

- If you define a function it means that the body of the function is written: `int f(void) { return 0; }`
- If you define a variable, it means that a part of memory is reserved for the variable: `int x;` or `extern int x=0;`

When a variable is not defined, the software considers the variable to be initialized, and to have potentially any value in its full range.

When a function is not defined, the software stubs the function.

Declaration

- Function declaration: `int f(void);`
- Variable declaration: `extern int x;`

A declaration provides information about the type of the function or variable. If you use the function or variable in a file where it has not been declared, a compilation error results.

Assumptions About Implicit Data Type Conversions

If an operation involves two operands, the verification assumes that before the operation takes place, the operands can undergo implicit data type conversion. Whether this conversion happens depends on the original data types of the operands.

Following are the conversion rules that apply if the operands in a binary operation have the same data type. Both operands can be converted to `int` or `unsigned int` type before the operation is performed. This conversion is called integer promotion. The conversion rules are based on the ANSI C99 Standard.

- `char` and `signed short` variables are converted to `int` variables.
- `unsigned short` variables are converted to `int` variables only if an `int` variable can represent all possible values of an `unsigned short` variable.

  For targets where the size of `int` is the same as size of `short`, `unsigned short` variables are converted to `unsigned int` variables. For more information on data type sizes, see `Target processor type (-target)`.
- Types such as `int`, `long` and `long long` remain unchanged.

Following are some of the conversion rules that apply when the operands have different data types. The rules are based on the ANSI C99 Standard.

- If both operands are `signed` or `unsigned`, the operand with a lower-ranked data type is converted to the data type of the operand with the higher-ranked type. The rank increases in the order `char`, `short`, `int`, `long`, and `long long`.
- If one operand is `unsigned` and the other `signed`, and the `unsigned` operand data type has a rank higher or the same as the `signed` operand data type, the `signed` operand is converted to the `unsigned` operand type.

  For instance, if one operand has data type `int` and the other has type `unsigned int`, the `int` operand is converted to `unsigned int`.

Implicit Conversion When Operands Have Same Data Type

This example shows implicit conversions when the operands in a binary operation have the same data type. If you run verification on the examples, you can use tooltips on the **Source** pane to see the conversions.

In the first addition, `i1` and `i2` are not converted before the addition. Their sum can have values outside the range of an `int` type because `i1` and `i2` have full-range values. Therefore, the Overflow on page 3-88 check on the first addition is orange.

In the second addition, `c1` and `c2` are promoted to `int` before the addition. The addition does not overflow because an `int` variable can represent all values that result from the sum of two `char` variables. The **Overflow** check on the second addition is green. However, when the sum is assigned to a `char` variable, an overflow occurs during the conversion from `int` back to `char`. An orange **Overflow** check appears on the = operation.

```
extern char input_char(void);
extern int input_int(void);
```

```
void main(void) {
    char c1 = input_char();
    char c2 = input_char();
    int i1 = input_int();
    int i2 = input_int();

    i1 = i1 + i2;
    c1 = c1 + c2;
}
```

Implicit Conversion When Operands Have Different Data Types

The following examples show implicit conversions that happen when the operands in a binary operation have different data types. If you run verification on the examples, you can use tooltips on the **Source** pane to see the conversions.

In this example, before the `<=` operation, `x` is implicitly converted to `unsigned int`. Therefore, the User assertion on page 3-110 check is red.

```
#include <assert.h>
int func(void) {
    int x = -2;
    unsigned int y = 5;
    assert(x <= y);
}
```

In this example, in the first `assert` statement, x is implicitly converted to `unsigned int` before the operation x `<=` y. Because of this conversion, in the second `assert` statement, x is greater than or equal to zero. The User assertion on page 3-110 check on the second `assert` statement is green.

```
int input(void);

void func(void) {
    unsigned int y = 7;
    int x = input();
    assert ( x >= -7 && x <= y );
    assert ( x >=0 && x <= 7);
}
```

Assumptions About memset and memcpy

In this section...
"Polyspace Specifications for memcpy" on page 4-24
"Polyspace Specifications for memset" on page 4-25

Polyspace Specifications for memcpy

Syntax:

```
#include <string.h>
void * memcpy ( void * destinationPtr, const void * sourcePtr, size_t num );
```

If your code uses the `memcpy` function, see the information in this table.

Specification	Example
Polyspace runs a **Invalid use of standard library routine** check on the function. The check determines if the memory block that `sourcePtr` or `destinationPtr` points to is greater than or equal in size to the memory assigned to them through `num`.	```#include <string.h>typedef struct { char a; int b; } S;void func(int);void main() { S s; int d; memcpy(&d, &s, sizeof(S));}``` In this code, Polyspace produces a red **Invalid use of standard library routine** error because: • `d` is an `int` variable. • `sizeof(S)` is greater than `sizeof(int)`. • A memory block of size `sizeof(S)` is assigned to &d.

Specification	Example
Polyspace does not check if the memory that `sourcePtr` points to is itself initialized. Following the use of `memcpy`, Polyspace considers that the variables that `destinationPtr` points to can have any value allowed by their type.	```#include <string.h>
typedef struct {
    char a;
    int b;
} S;

void func(int);

void main() {
  S s, d={'a',1};
  int val;
  val = d.b; // val=1

  memcpy(&d, &s, sizeof(S));
  val = d.b;
  // val can have any int value
}```

In this code, when the `memcpy` function copies `s` to `d`, Polyspace does not produce a red **Non-initialized local variable** error. Following the copy, the verification considers that the fields of `d` can have any value allowed by their type. For instance, `d.b` can have any value in the range allowed for an `int` variable. |
| Polyspace raises a red **Invalid use of standard library routine** check if the source and destination arguments overlap. Overlapping assignments are forbidden by the C Standard. | A red check is produced for this memory assignment:

```#include <string.h>

int main() {
  char arr[4];
  memcpy (arr, arr + 3, sizeof(int));
}``` |

Polyspace Specifications for memset

Syntax:

```
#include <string.h>
void * memset ( void * ptr, int value, size_t num );
```

If your code uses the `memset` function, see the information in this table.

Specification	Example
Polyspace runs a **Invalid use of standard library routine** check on the function. The check determines if the memory block that `ptr` points to is greater than or equal in size to the memory assigned to them through `num`.	```#include <string.h>
typedef struct {
    char a;
    int b;
} S;

void main() {
 int val;
 memset(&val,0,sizeof(S));
}```

In this code, Polyspace produces a red **Invalid use of standard library routine** error because:

• `val` is an `int` variable.

• `sizeof(S)` is greater than `sizeof(int)`.

• A memory block of size `sizeof(S)` is assigned to `&val`. |
| If `value` is 0, following the use of `memset`, Polyspace considers that the variables that `ptr` points to have the value 0. | ```#include <string.h>
typedef struct {
    char a;
    int b;
} S;

void main() {
 S s;
 int val;
 memset(&s,0,sizeof(S));
 val=s.b; //val=0
}```

In this code, Polyspace considers that following the use of `memset`, each field of `s` has value 0. |

Specification	Example

Following the use of `memset`, if `value` is anything other than 0, Polyspace considers that:
 	
• The variables that `ptr` points to can be noninitialized.
 	
• If initialized, the variables can have any value that their type allows.
	``c
	#include <string.h>
	typedef struct {
	    char a;
	    int b;
	} S;
	
	void main() {
	 S s;
	 int val;
	 memset(&s,1,sizeof(S));
	 val=s.b;
	 // val can have any int value
	}
	``
	
	In this code, Polyspace considers that following the use of `memset`, each field of `s` has any value that its type allows. For instance, `s.b` can have any value in the range allowed for an `int` variable.
	
	Following the `memset`, the structure fields can have different values depending on the structure packing and padding bits. Therefore, structure field assignments with `memset` are implementation-dependent. Code Prover performs this part of the analysis in an implementation-independent way. The analysis allows all possible paddings and therefore full range of values for the structure fields.

Assumptions About #pragma Directives

The verification ignores most `#pragma` directives, because they do not carry information relevant to the verification.

However, the verification takes into account the behavior of these pragmas.

Pragma	Effect on Verification

`#pragma asm` and `#pragma endasm`
 	
`#asm` and `#endasm`
	The verification ignores the content between the pragmas.
	
	If you use `#pragma inline_asm` *func*, the verification considers that the function *func* contains assembly level instructions and ignores the function body.

`#pragma hdrstop`	For Visual C++ compilers, the verification stops processing precompiled headers at the point where it encounters the pragma.
`#pragma once`	The verification allows the current source file to be included only once in a compilation.
`#pragma pack(n)`, `#pragma pack(push[,n])`, `#pragma pack(pop)`	The verification takes into account the boundary alignment specified in the pragmas. `#pragma pack` without an argument is treated as `#pragma pack(1)`. For more information, see the following example.
`#pragma inline global` *func* or `#pragma inline` *func*	The verification considers the function *func* as an inline function. In particular, by default, the Code Prover generated `main` does not call these functions directly with the assumption that they are called in other functions.
`_Pragma("inline=never")` *func*	The verification does not inline function *func*.
`#error` *message*	The verification stops if it encounters the directive. For more information, see "Error Related to #error Directive".

For more information on the pragmas, see your compiler documentation. If the verification does not take into account a certain pragma from the preceding list, see if you specified the right compiler for your verification. For more information, see `Compiler (-compiler)`.

For instance, in this code, the directives `#pragma pack(`*n*`)` force a new alignment boundary in the structure. The User assertion on page 3-110 checks in the `main` function are green because the verification takes into account the behavior of the directives. The verification uses these options:

- Target processor type (`-target`): `i386` (`char`: 1 byte, `int`: 4 bytes)
- Compiler (`-compiler`): `gnu4.9`

```
#include <assert.h>

#pragma pack(2)

struct _s6 {
    char c;
    int i;
} s6;

#pragma pack() /* Restores default packing: pack(4) */

struct _sb {
    char c;
    int i;
} sb;

#pragma pack(1)

struct _s5 {
    char c;
    int i;
} s5;


int main(void) {
    assert(sizeof(s6) == 6);
    assert(sizeof(sb) == 8);
    assert(sizeof(s5) == 5);
    return 0;
}
```

Assumptions About Standard Library Float Routines

For some two-argument standard library float routines, the verification can ignore the function arguments and assume that the function returns all possible values in its range.

In this code, the first `assert` statement is true and the second `assert` statement is false. However, because the verification assumes that `fmodf` and `nextafterf` return full-range values, it considers that the `assert` statements are false but only on a fraction of possible execution paths. Therefore, the `User assertion` checks on the `assert` statements are orange.

```
#include <math.h>
int main() {
 float val1=10.0, val2=3.0,res;
 res = fmodf(val1/val2);
 assert(res==1.0);

 res = nextafterf(val2,val1);
 assert(res<3.0);
}
```

Assumptions About Unions

In some situations, unions can help you construct efficient code. However, if you write a union member and read back a different union member, the behavior depends on the member sizes and can be implementation-dependent. You have to determine the following for your implementation:

- **Padding** – Padding can be inserted at the end of a union.
- **Alignment** – Members of structures within a union can have different alignments.
- **Endianness** – Whether the most significant byte of a word is stored at the lowest or highest memory address.
- **Bit-order** – Bits within bytes can have both different numbering and allocation to bit fields.

When you use unions in your code, because of these issues, Polyspace verification can lose precision.

If you write a union member and read back another union member, Polyspace considers that the latter member can have any value that its type allows. In this code, the member b of X is written, but a is read. Polyspace considers that a can have any `int` value and both branches of the `if-else` statement are reachable.

```
typedef union _u {
    int a;
    char b[4];
} my_union;

void main() {
    my_union X;

    X.b[0] = 1;
    X.b[1] = 1;
    X.b[2] = 1;
    X.b[1] = 1;
    if (X.a == 0x1111) {
    }
    else {
    }
}
```

To avoid using unions in your code, check for violations of `MISRA C:2012 Rule 19.2`.

Note If you initialize a union using a static initializer, following ANSI C standard, Polyspace considers that the union member appearing first in the declaration list gets initialized.

Assumptions About Variables Cast as Void Pointers

The C language allows the use of statements that cast a variable as a void pointer. However, Polyspace verification of these statements entails a loss of precision.

Consider:

```
1   typedef struct {
2   int x1;
3   } s1;
4
5   s1 object;
6
7   void g(void *t) {
8   int x;
9   s1 *p;
10
11  p = (s1 *)t;
12  x = p->x1;    // x should be assigned value 5 but p->x1 is full-range
13  }
14
15  void main(void) {
16  s1 * p;
17
18  object.x1 = 5;
19  p = &object;
20  g((void *)p); // p cast as void pointer
21  }
```

On line 12, the variable x must be assigned the value 5. However, the software assumes that `p->x1` has full range of values allowed by its type.

Assumptions About Assembly Code

Polyspace recognizes most inline assemblers as introduction of assembly code. The verification ignores the assembly code but accounts for the fact that the assembly code can modify variables in the C code.

If introduction of assembly code causes compilation errors:

1   Embed the assembly code between a #pragma *my_asm_begin* and a #pragma *my_asm_end* statement.

2   Specify the analysis option -asm-begin *my_asm_begin* -asm-end *my_asm_end*.

For more information, see -asm-begin -asm-end.

Recognized Inline Assemblers

Polyspace recognizes these inline assemblers as introduction of assembly code.

- asm

    Examples:

    - int f(void)
      ```
      {
          asm ("% reg val; mtmsr val;");
          asm("\tmove.w #$2700,sr");
          asm("\ttrap #7");
          asm(" stw r11,0(r3) ");
          assert (1); // is green
          return 1;
      }
      ```
    - int other_ignored2(void)
      ```
      {
          asm "% reg val; mtmsr val;";
          asm mtmsr val;
          assert (1); // is green
          asm ("px = pm(0,%2); \
              %0 = px1; \
              %1 = px2;"
              : "=d" (data_16), "=d" (data_32)
              : "y" ((UI_32 pm *)ram_address):
      "px");
          assert (1); // is green
      }
      ```
    - int other_ignored4(void)
      ```
      {
          asm {
              port_in: /* byte = port_in(port); */
              mov EAX, 0
              mov EDX, 4[ESP]
                  in AL, DX
                  ret
                  port_out: /* port_out(byte,port); */
              mov EDX, 8[ESP]
              mov EAX, 4[ESP]
      ```

```
                    out DX, AL
                    ret }
          assert (1); // is green
          }
```

- __asm__

  Examples:

  - int other_ignored6(void)
    ```
    {
    #define A_MACRO(bus_controller_mode) \
        __asm__ volatile("nop"); \
        __asm__ volatile("nop"); \
        __asm__ volatile("nop"); \
        __asm__ volatile("nop"); \
        __asm__ volatile("nop"); \
        __asm__ volatile("nop")
            assert (1); // is green
            A_MACRO(x);
            assert (1); // is green
            return 1;
    }
    ```

  - int other_ignored1(void)
    ```
    {
        __asm
            {MOV R8,R8
            MOV R8,R8
            MOV R8,R8
            MOV R8,R8
            MOV R8,R8}
        assert (1); // is green
    }
    ```

  - int GNUC_include (void)
    ```
    {
        extern int __P (char *__pattern, int __flags,
        int (*__errfunc) (char *, int),
        unsigned *__pglob) __asm__ ("glob64");
        __asm__ ("rorw $8, %w0" \
            : "=r" (__v) \
            : "0" ((guint16) (val)));
        __asm__ ("st g14,%0" : "=m" (*(AP)));
        __asm("" \
            : "=r" (__t.c) \
            : "0" ((((union { int i, j; } *) (AP))++)->i));
        assert (1); // is green
        return (int) 3 __asm__("% reg val");
    }
    ```

  - int other_ignored3(void)
    ```
    {
        __asm {ldab 0xffff,0;trapdis;};
    __asm {ldab 0xffff,1;trapdis;};
        assert (1); // is green
        __asm__ ("% reg val");
        __asm__ ("mtmsr val");
        assert (1); // is green
        return 2;
    }
    ```

- #pragma asm #pragma endasm

   Examples:

   - ```
     int pragma_ignored(void)
     {
         #pragma asm
             SRST
         #pragma endasm
             assert (1); // is green
     }
     ```
   - ```
     void test(void)
     {
       #asm
         mov _as:pe, reg
             jre _nop
       #endasm
         int r;
         r=0;
         r++;
     }
     ```

Single Function Containing Assembly Code

The software stubs a function that is preceded by `asm`, even if a body is defined.

```
asm int h(int tt)               // function h is stubbed even if body is defined
{
  % reg val;                    // ignored
  mtmsr val;                   // ignored
  return 3;                      // ignored
};

void f(void) {
  int x;
  x = h(3);                    // x is full-range
}
```

Multiple Functions Containing Assembly Code

The functions that you specify through the following pragma are stubbed automatically, even if function bodies are defined.

```
#pragma inline_asm(list of functions)
```

Code examples:

```
#pragma inline_asm(ex1, ex2)
   // The functions ex1 and ex2 are
   // stubbed, even if their bodies are defined

int ex1(void)
{
  % reg val;
  mtmsr val;
  return 3;                         // ignored
};

int ex2(void)
```

```
{
  % reg val;
  mtmsr val;
  assert (1);                      // ignored
  return 3;
};


#pragma inline_asm(ex3)  // the definition of ex3 is ignored

int ex3(void)
{
  % reg val;
  mtmsr val;                 // ignored
  return 3;
};

void f(void) {
  int x;

  x = ex1();                     // ex1 is stubbed : x is full-range
  x = ex2();                     // ex2 is stubbed : x is full-range
  x = ex3();                     // ex3 is stubbed : x is full-range
}
```

Local Variables in Functions with Assembly Code

The verification ignores the content of assembly language instructions, but following the instructions, it makes some assumptions about:

- *Uninitialized local variables*: The assembly instructions can initialize these variables.

- *Initialized local variables*: The assembly instructions can write any possible value to the variables allowed by the variable data types.

For instance, the function f has assembly code introduced through the asm statement.

```
int f(void) {
    int val1, val2 = 0;
    asm("mov 4%0,%%eax"::"m"(val1));
    return (val1 + val2);
}
```

On the return statement, the **Non-initialized local variable** check has the following results:

- val1: The check is orange because the assembly instruction can initialize val1.

- val2: The check is green. However, val2 can have any int value.

If the variable is static, the assumptions are true anywhere in the function body, even before the assembly instructions.

Determination of Program Stack Usage

The Polyspace Code Prover analysis can estimate stack usage of each function in your program and compute the entire program stack usage. The analysis uses the function call hierarchy of your program to estimate stack usage. The stack usage of a function is the sum of local variable sizes in the function plus the maximum stack usage from function callees. The stack usage of the function at the top of the call hierarchy is the program stack usage.

For instance, for this call hierarchy, the stack usage of `func` is the size of local variables in `func` plus the maximum stack usage from `func1` and `func2` (unless they are called in mutually exclusive branches of a conditional statement).

For details, see:

- Function metrics: `Maximum Stack Usage` and `Minimum Stack Usage`
- Project metrics: `Program Maximum Stack Usage` and `Program Minimum Stack Usage`

Investigate Possible Stack Overflow

If your stack usage exceeds available stack space, you can identify which function is responsible. Begin at the `main` function and navigate your program call tree. During navigation, look for the function that has an unreasonable size of local variables. If you cannot identify such a function, look for a call sequence that is unreasonably long. The detailed steps for navigation are:

1 On the **Source** pane, select the `main` function. On the **Call Hierarchy** pane, you see the functions called from `main` (callees). To see the full hierarchy, right-click a function and expand all nodes.

 If the **Call Hierarchy** pane is not open by default, select **Window > Show/Hide View > Call Hierarchy**.

2   To navigate to the callee definition in your source, on the **Call Hierarchy** pane, double-click each callee name. Then, click the callee name on the **Source** pane. The **Result Details** pane shows the higher estimate of local variable size and stack usage by the callee.

Stack Usage Not Computed

For function stack usage to be computed, the analysis must be able to reach the end of the function. The following can prevent the computation of function stack usage:

- Red checks.

  If a definite run-time error occurs in a function or one of its callees, the analysis does not compute its stack usage. The reason is that code following a red check is not analyzed. If the unanalyzed code contains function calls, any stack usage estimate for the caller function is inaccurate.

  In this example, the stack usage of `func` is not computed because following the red overflow, the remainder of the function is not analyzed. If the stack usage was computed, function calls in the unanalyzed code, such as the call to `func2`, would not be part of the computation.

  ```
  #include <limits.h>
  void func(void) {
      int val=INT_MAX;
      val++;
  ```

```
        func2();
    }
```

- Recursive functions.

    If a function calls itself directly or indirectly, its stack usage and the stack usage of all functions that call this function are not computed.

    If a program contains recursive functions, the program minimum stack usage might be computed even if the maximum stack usage is not. In this case, the program minimum stack usage computation uses an execution path that bypasses the call to the recursive function, and might not be an accurate representation of the stack usage.

If the program stack usage appears as not computed, make sure that the stack usage of all functions are computed. In the **Information** column on the **Results List** pane, check if a function stack usage result shows the value Not computed.

Stack Usage Assumptions

If a function is called but not defined in the code that you provide to Polyspace, the stack usage determination does not take the function call into account.

This assumption applies to:

- Implicit C++ constructors.

    For instance, in this example, func calls the constructor of class myClass when myObj is defined. Stack usage determination does not consider the constructor as a callee of func.

    ```
    class myClass {std::string str;};

    void func() {
        myClass myObj;
    }
    ```

- Standard library functions or other functions whose definitions are missing from the code in your Polyspace project.

    For instance, in this example, func calls the standard library function cos. Unless you provide the definition of cos, stack usage determination does not consider it as a callee of func.

    ```
    #include <math.h>

    double func(double arg) {
        return cos(arg);
    }
    ```

Limitations of Polyspace Verification

Code verification has certain limitations. The *Polyspace Code Prover Limitations* document describes known limitations of the code verification process.

This document is stored as `codeprover_limitations.pdf` in the following folder:

polyspaceroot`\polyspace\verifier\code_prover_desktop`

Here, *polyspaceroot* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2021a`.

Functions, Classes, Methods, Properties, and Apps

polyspace-autosar

(DOS/UNIX) Run Polyspace Code Prover on code implementation of AUTOSAR software components

Syntax

```
polyspace-autosar -create-project projectFolder -arxml-dir arxmlFolder -
sources-dir codeFolder [-sources-dir codeFolder] [OPTIONS]
polyspace-autosar -create-project projectFolder -select-arxml-files
arxmlFiles [-select-arxml-files arxmlFiles] -select-source-files codeFiles [-
select-source-files codeFiles] [OPTIONS]
```

```
polyspace-autosar -update-project prevProjectFile [OPTIONS]
```

```
polyspace-autosar -update-and-clean-project prevProjectFile [OPTIONS]
```

```
polyspace-autosar -help
```

Description

`polyspace-autosar -create-project projectFolder -arxml-dir arxmlFolder -sources-dir codeFolder [-sources-dir codeFolder] [OPTIONS]` checks the code implementation of AUTOSAR software components for run-time errors and violation of data constraints in the corresponding AUTOSAR XML specifications. The analysis parses the AUTOSAR XML specifications (`.arxml` files) in `arxmlFolder`, modularizes the code implementation (`.c` files) in `codeFolder` based on the specifications, and runs Code Prover on each module for the checks. The Code Prover results are stored in `projectFolder`. After analysis, you can open the project `psar_project.psprj` from `projectFolder` in the Polyspace user interface. You can view the results for each software component individually or upload them to Polyspace Metrics for an overview.

You can use additional options for troubleshooting, for instance, to only perform certain parts of the update and track down an issue or to provide extra header files or define macros.

`polyspace-autosar -create-project projectFolder -select-arxml-files arxmlFiles [-select-arxml-files arxmlFiles] -select-source-files codeFiles [-select-source-files codeFiles] [OPTIONS]` creates a Polyspace project from AUTOSAR specifications as in the preceding syntax but allows you to exclude specific files or folders from analysis using shell patterns or regular expressions.

`polyspace-autosar -update-project prevProjectFile [OPTIONS]` updates the Code Prover analysis results based on changes in ARXML files or C source code since the last analysis. The update uses the html file `prevProjectFile` from the previous analysis and only reanalyzes the code implementation of software components that changed since that analysis.

You can use additional options for troubleshooting.

`polyspace-autosar -update-and-clean-project prevProjectFile [OPTIONS]` updates the Code Prover analysis results based on changes in ARXML files or C source code since the last analysis. The update only reanalyzes the code implementation of software components that changed since the previous analysis. A clean update also removes information about software components that

are out of date. For instance, if you use an additional option to force the update for specific software components and other SWC-s have also changed, a clean update removes those other SWC-s from the Polyspace project.

You can use additional options for troubleshooting.

`polyspace-autosar -help` shows all options available for `polyspace-autosar`.

Examples

Run Code Prover on All Software Components

Suppose your ARXML files are in a folder `arxml` and your C source files in a folder `code` in the current folder.

Run Code Prover on all software components defined in your ARXML files. Store the results in a folder `polyspace` in the current folder.

```
polyspace-autosar -create-project polyspace -arxml-dir arxml -sources-dir code
```

The analysis creates a Polyspace project with several modules. Each module collects the C code implementation of a software component. The analysis runs Code Prover on each module and checks the code for run-time errors or mismatch with ARXML specifications.

After analysis, you can open the results in several ways. See "Review Polyspace Results on AUTOSAR Code".

Update an ARXML or code file. For instance, in Linux, you can `touch` a file to indicate an update. Check if the updates affected results of the Code Prover analysis. For an updated analysis, provide the project file `psar_project.html` created in the previous step.

```
polyspace-autosar -update-project polyspace\psar_project.xhtml
```

If you update an ARXML file, the entire analysis is repeated. If you update your source code, the analysis is repeated only for software components whose code implementation was updated.

Run Code Prover on Specific Software Components

Instead of running Code Prover on all software components, check specific software components only.

For instance, suppose a software component has the fully qualified path `pkg.component.bhv`. You can run Code Prover only on this software component.

```
polyspace-autosar -create-project polyspace -arxml-dir arxml -sources-dir code
    -autosar-behavior pkg.component.bhv
```

You can run Code Prover on all software components but later choose to update the analysis for specific software components only.

```
polyspace-autosar -update-project polyspace\psar_project.xhtml
    -autosar-behavior pkg.component.bhv
```

If you do not reanalyze a software component that has been updated, the analysis shows that the software component might be out of date.

You can also update the analysis for specific software components and remove all traces of other software components.

```
polyspace-autosar -update-and-clean-project polyspace\psar_project.xhtml
    -autosar-behavior pkg.component.bhv
```

Run Code Prover and Upload Results to Polyspace Metrics

In the Polyspace user interface, you can see the results for individual software components. To see an overview of Code Prover results for all software components analyzed, upload the results to Polyspace Metrics.

If you perform verification on a server, you can specify before verification that all results must be uploaded to Polyspace Metrics. Specify remote verification and results upload using these additional options:

- Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`)
- Upload results to Polyspace metrics (`-add-to-results-repository`)

You can specify additional options with the flag `-extra-project-options`.

For instance:

```
polyspace-autosar -create-project polyspace -arxml-dir arxml -sources-dir code
  -extra-project-options "-add-to-results-repository -batch -scheduler localhost
  -prog polyspace_project -verif-version 1.0"
```

Here `localhost` indicates that the same computer serves as the server and client. Replace it with the name of your server. The argument of `-prog` can be the same as that of `-create-project`. You use the options `-prog` and `-verif-version` to set the project name and version number as it appears on Polyspace Metrics.

Alternatively, you can run Code Prover and upload each result using the `polyspace-results-repository` command.

Input Arguments

projectFolder — Folder to store Polyspace results
string

Folder name, specified as a string (in double quotes). If the folder exists, it must be empty.

After analysis, the folder contains two project files `psar_project.psprj` and `psar_project.html`.

- To see the results, open the file `psar_project.psprj` in the Polyspace user interface or the file `psar_project.html` in a web browser.
- For subsequent updates on the command line, use the file `psar_project.html`.

See also "Review Polyspace Results on AUTOSAR Code".

Example: `"C:\Polyspace_Projects\proj_swc1"`

arxmlFolder — Root folder containing ARXML files
string

Folder name, specified as a string (in double quotes). You can omit the double quotes if your folder paths do not contain spaces.

UNC paths are not supported for the folder name.

Example: "C:\arxml_swc1"

codeFolder — Root folder containing C files
string

Folder name, specified as a string (in double quotes). You can omit the double quotes if your folder paths do not contain spaces.

To specify multiple root folders containing sources, repeat the `-sources-dir` option. If you specify multiple root folders, they must not overlap. For instance, one root folder cannot be a subfolder of the other.

UNC paths are not supported for the folder name.

Example: "C:\code_swc1"

arxmlFiles — Root folder containing ARXML files along with files and folders to exclude
string

Root folder containing ARXML files followed by file and folder inclusions and exclusions, specified as a string. To create this string:

1   Use the Linux `find` command to search for the files and folders to include and exclude.
2   Copy the `find` options and enclose them in double quotes.

For examples, see "Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis".

codeFiles — Root folder containing source files along with files and folders to exclude
string

Root folder containing code (`.c` and `.h`) files followed by file and folder inclusions and exclusions, specified as a string. To create this string:

1   Use the Linux `find` command to search for the files and folders to include and exclude.
2   Copy the `find` options and enclose them in double quotes.

For examples, see "Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis".

prevProjectFile — Path to `psar_project.html`
string

Path to the previously created project file `psar_project.html`, specified as a string (in double quotes). You can omit the double quotes if your folder paths do not contain spaces.

Example: "C:\Polyspace_Projects\proj1\psar_project.html"

[OPTIONS] — Options to control project creation
string

Options to control creation of Polyspace project and subsequent analysis. You primarily use the options for troubleshooting, for instance, to only perform certain parts of the update and narrow down an issue or to provide extra header files or define macros.

General options

Option	Description
`-verbose`	Save additional information about the various phases of command execution (verbose mode). The file `psar_project.log` and other auxiliary files store this additional information.
	If an error occurs in command execution, the error message is stored in a separate file, irrespective of whether you enable verbose mode. Running in verbose mode only stores the various phases of execution. You can use this information to see when an error was introduced.
`-options-file OPTION_FILE`	Use an options file to supplement or replace the command line options. In the options file, specify each option on a separate line. Begin a line with `#` to indicate comments.
	An options file `opts.txt` can look like this:

	`# Store Polyspace results`
	`-create-project polyspace`
	`# ARXML Folder`
	`-arxml-dir arxml`
	`# SOURCE Folder`
	`-sources-dir code`

	You can run `polyspace-autosar` as:
	`polyspace-autosar -options-file opts.txt`
	If an option that is directly specified with the `polyspace-autosar` command conflicts with an option in the options file, the directly specified option is used. For instance, in this example, the folder `proj` is used to save the Polyspace project.
	`polyspace-autosar -create-project proj` `                   -options-file opts.txt`
	You typically use an options file to store and reuse options that are common to multiple projects.

Options to control update of project

If you update a project, by default, the analysis results are updated for all AUTOSAR SWCs behaviors with respect to any change in the arxml files or C source code since the last analysis. These options allow you to control the update.

Option	Description

`-autosar-behavior` *AUTOSAR_QUALIFIED_NAME*	Check the implementation of software components whose internal behavior-s are specified by *AUTOSAR_QUALIFIED_NAME*. The default analysis considers all software components present in the ARXML specifications.
	
	To specify multiple software components, repeat the option. Alternatively, you can do one of the following:
	
	• Use shell patterns similar to the patterns used with `-select-arxml-files` and `-select-source-files`.
	
	  For examples, see "Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis".
	
	• Use regular expressions to specify a group of software components under the same package.
	
	  For instance:
	
	    • To specify the software component whose internal behavior has the fully qualified name `pkg.component.bhv`, use:
	
	      `-autosar-behavior pkg.component.bhv`
	
	    • To specify the software components whose internal behavior-s have fully qualified names beginning with `pkg.component`, use:
	
	      `-autosar-behavior pkg.component\..*`
	
	      The `\.` represents the package name separator `.` (dot) and the `.*` represents any number of characters.

`-do-not-update-autosar-prove-environment`	Do not read the ARXML specifications. Use ARXML specifications stored from the previous analysis. Use this option during project updates to compare the code against previous specifications. Unless you use this option, project updates read the entire ARXML specifications again.

Option	Description
`-do-not-update-extract-code`	Do not read the C source code. Use source code stored from the previous analysis. Use this option during project updates to compare the previous source code against ARXML specifications. Unless you use this option, project updates consider all changes to the source code since the previous analysis.
`-do-not-update-verification`	Read the ARXML specifications and C code implementation only but do not run the Code Prover analysis. Use this option during project updates to investigate errors introduced in the ARXML specifications or compilation errors introduced in the source code. You can first fix these issues and then run the Code Prover analysis.

Options to control parsing of ARXML specifications

Option	Description

`-autosar-datatype`
AUTOSAR_QUALIFIED_NAME
	Import definition of AUTOSAR data types specified by *AUTOSAR_QUALIFIED_NAME*. The default analysis only imports data types specified in the internal behavior of software components that you verify.
	
	To specify multiple data types, repeat the option. Alternatively, you can do one of the following:
	
	• Use shell patterns similar to the patterns used with `-select-arxml-files` and `-select-source-files`.
	
	  For examples, see "Select AUTOSAR XML (ARXML) and Code Files for Polyspace Analysis".
	
	• Use regular expressions to specify all data types under the same package.
	
	  For instance:
	
	  • To specify a data type that has the fully qualified name `pkg.datatypes.type`, use:
	
	    `-autosar-datatype pkg.datatypes.type`
	
	  • To specify data types that have fully qualified names beginning with `pkg.datatypes`, use:
	
	    `-autosar-datatype pkg.datatypes\..*`
	
	    The `\.` represents the package name separator `.` (dot) and the `.*` represents any number of characters.
	
	  • To force import of all data types, use:
	
	    `-autosar-datatype .*\..*`

Option	Description

`-Eautosar-xmlReaderSameUuidForDifferentElements`
 	
`-Eno-autosar-xmlReaderSameUuidForDifferentElements`
	If multiple elements in the ARXML specifications have the same universal-unique-identifier (uuid), use these options to toggle between a warning and an error.
	
	The default analysis stops with an error if the issue happens. To convert to a warning, use `-Eno-autosar-xmlReaderSameUuidForDifferentElements`. For conflicting UUID-s, the analysis stores the last element read and continues with a warning.
	
	The subsequent executions continue to use the warning mode.To revert back to an error, use `-Eautosar-xmlReaderSameUuidForDifferentElements`.

`-Eautosar-xmlReaderTooManyUuids`
 	
`-Eno-autosar-xmlReaderTooManyUuids`
	If the same element in the ARXML specifications has different universal-unique-identifiers (uuid-s), use these options to toggle between a warning and an error.
	
	The default analysis stops with an error if the issue happens. To convert to a warning, use `-Eno-autosar-xmlReaderTooManyUuids`. For conflicting UUID-s, the analysis stores the last element read and continues with a warning.
	
	The subsequent executions continue to use the warning mode. To revert back to an error, use `-Eautosar-xmlReaderTooManyUuids`.

Options to control reading of C source code

Option	Description
`-include` *USER_RTE_TYPE_H*	Define additional data types and macros that are not part of your ARXML specifications, but needed for analysis of the code implementation. Add the data type and macro definitions to a file *USER_RTE_TYPE_H*. These definitions are appended to a header file `Rte_Type.h` that is used in the analysis. The file that you provide must itself not be named `Rte_Type.h`. You can provide the file with data type and macro definitions only during project creation. For subsequent updates, you can change the contents of this file but not provide a new file. Also, this file must not be in the same folder as the Polyspace project and results. If you additionally define macros or undefine them using the options `-D` or `-U`, for definitions that conflict with the ones in *USER_RTE_TYPE_H*, the `-D` or `-U` specifications prevail.

`-I` *INCLUDE_FOLDER*	Specify folders containing header files. The analysis looks for `#include`-d files in this folder. The folder must be a subfolder of your source code folder.
	
	Repeat the option for multiple folders. The analysis looks for header files in these folders in the order in which you specify them.
	
	If you want to specify folders that are not in the source code folder, use the option:
	
	`-extra-project-options`
	`"-I `*INCLUDE_FOLDER*`"`

`-D` *DEFINE*	Specify macros that the analysis must consider as defined. For instance, if you specify: `-D _WIN32` the preprocessor conditional `#ifdef _WIN32` succeeds and the corresponding branch is executed.

Option	Description
`-U` *UNDEFINE*	Specify macros that the analysis must consider as undefined. For instance, if you specify: `-U _WIN32` the preprocessor conditional `#ifndef _WIN32` succeeds and the corresponding branch is executed.

Options to control Code Prover checks

Option	Description
`-extra-project-options` *POLYSPACE_OPTIONS*	Specify additional options for the Code Prover analysis. The options that you specify do not apply to the ARXML parsing or code extraction, but only to the subsequent Code Prover analysis. Use this method to specify analysis options that you use with the `polyspace-code-prover` command. See "Analysis Options in Polyspace Code Prover". Note that these options of `polyspace-code-prover` do not need to be specified: • `-sources`: `polyspace-autosar` extracts the required source files. • `-I`: You specify include folders with the `-I` option of `polyspace-autosar`. • "Inputs and Stubbing" options such as `-data-range-specifications`: External data constraints in your ARXML files are extracted automatically with `polyspace-autosar`. You cannot specify constraints explicitly. • "Multitasking" options such as `-entry-points`: You cannot perform a multitasking analysis with `polyspace-autosar`. To detect data races, create a separate project for the entire application and explicitly add your source folders. Specify the ARXML files relevant for multitasking and run Bug Finder. For more information, see `ARXML files selection (-autosar-multitasking)`. • "Code Prover Verification" options associated with `main` generation: A `main` function is generated (in the file `psar_prove_main.c`) when you create a Polyspace project from an AUTOSAR description. The `main` function calls functions that implement runnable entities in the software components. The generated `main` is needed for the Code Prover analysis. You cannot change the properties of this `main` function.

Option	Description

`-extra-options-file OPTIONS_FILE`	Specify additional options for the Code Prover analysis in an options file. The options that you specify do not apply to the ARXML parsing or code extraction, but only to the subsequent Code Prover analysis.
	
	For instance, you can trace your build command to gather compiler options, macro definitions and paths to include folders, and provide this information in an options file for analysis of code implementation of AUTOSAR software components.
	
	1 Trace your build command (for instance, `make`) with `polyspace-configure` and generate an options file for subsequent Code Prover analysis. Suppress inclusion of sources in the options file with the `-no-sources` option.
	
	    `polyspace-configure \`
	    `-output-options-file options.txt \`
	    `-no-sources make`
	
	2 Run Code Prover on AUTOSAR code with `polyspace-autosar`. Provide your ARXML folder, source folders and other options. In addition, provide the earlier generated options file with the `-extra-options-file` option.
	
	    `polyspace-autosar ... \`
	    `-extra-options-file options.txt`
	
	See also "Run Polyspace on AUTOSAR Code Using Build Command".

`-show-prove AUTOSAR_QUALIFIED_NAME`	After analysis, open results for a specific software component whose internal behavior is specified by *AUTOSAR_QUALIFIED_NAME*.

See Also

ARXML files selection (-autosar-multitasking) | AUTOSAR runnable not implemented | Invalid result of AUTOSAR runnable implementation | Invalid use of AUTOSAR runtime environment function

Topics
"Run Polyspace on AUTOSAR Code"
"Review Polyspace Results on AUTOSAR Code"
"Troubleshoot Polyspace Analysis of AUTOSAR Code"
"Benefits of Polyspace for AUTOSAR"
"Using Polyspace in AUTOSAR Software Development"

Introduced in R2018a

polyspace-code-prover

(DOS/UNIX) Run a Code Prover verification from Windows, Linux, or other command line

Syntax

`polyspace-code-prover [OPTIONS]`

`polyspace-code-prover -sources sourceFiles [OPTIONS]`

`polyspace-code-prover -sources-list-file listOfSources [OPTIONS]`

`polyspace-code-prover -options-file optFile`

`polyspace-code-prover -h[elp]`

Description

`polyspace-code-prover [OPTIONS]` runs a Code Prover verification if your current folder contains a `sources` subfolder with source files (`.c` or `.cxx` files). The verification considers files in `sources` and all subfolders under `sources`. You can customize the verification with additional options.

`polyspace-code-prover -sources sourceFiles [OPTIONS]` runs a Code Prover verification on the source file(s) `sourceFiles`. You can customize the verification with additional options.

`polyspace-code-prover -sources-list-file listOfSources [OPTIONS]` runs a Code Prover verification on the source files listed in the text file `listOfSources`. You can customize the verification with additional options.

`polyspace-code-prover -options-file optFile` runs a Code Prover verification with the options specified in the option file.

`polyspace-code-prover -h[elp]` lists a summary of possible analysis options.

Examples

Run Verification by Directly Specifying Options

Run a local Code Prover verification by specifying analysis options in the command itself. This example uses source files from a demo Polyspace Code Prover example. To run this example, replace *polyspaceroot* with the path to your Polyspace installation, for example `C:\Program Files\Polyspace\R2019a`.

Run a verification on `numerical.c` and `programming.c`, checking for MISRA C:2012 mandatory rules and using GNU 4.7 compiler settings. This example command is split by ^ characters for readability. In practice, you can put all commands on one line.

```
polyspaceroot\polyspace\bin\polyspace-code-prover -lang C^
  -sources polyspaceroot\polyspace\examples\cxx\Code_Prover_Example\sources\*.c,^
```

```
 -I polyspaceroot\polyspace\examples\cxx\Code_Prover_Example\sources\^
 -compiler generic -misra3 mandatory^
 -author jlittle -prog myProject -results-dir C:\Polyspace_Workspace\Results\
```

Open the results.

polyspaceroot\polyspace\bin\polyspace C:\Polyspace_Workspace\Results\^
ps_results.pscp

To rerun the verification, you must rerun it from the command line.

Run Verification with Options File

Run a verification by using an options file to specify your source files and analysis options. To run this example, replace *polyspaceroot* with the path to your Polyspace installation, for example C:\Program Files\Polyspace\R2019a.

Save this text to a text file called myOptsFile.txt.

```
# Polyspace analysis options
-I polyspaceroot\polyspace\examples\cxx\Code_Prover_Example\sources
-verif-version 1.0
-sources polyspaceroot\polyspace\examples\cxx\Code_Prover_Example\sources\*.c
-lang C
-target i386
-compiler generic
-dos
-do-not-generate-results-for all-headers
-misra3 mandatory-required
CustomRulesDefinition.txt
-entry-points proc1,proc2,server1,server2,tregulate
-critical-section-begin Begin_CS:Cs10
-critical-section-end End_CS:Cs10
-temporal-exclusions-file polyspaceroot\^
polyspace\examples\cxx\Code_Prover_Example\sources\temporal_exclusions.txt
-float-rounding-mode to-nearest
-signed-integer-overflows forbid
-unsigned-integer-overflows allow
-uncalled-function-checks none
-check-subnormal allow
-O2
-to Software Safety Analysis level 2
-context-sensitivity-auto
-path-sensitivity-delta 0
-author jlittle
-prog myProject
-results-dir C:\Polyspace_Workspace\Results\
```

Run the verification with the options specified in the text file.

polyspaceroot\polyspace\bin\polyspace-code-prover -options-file myOptsFile.txt

Open the results.

polyspaceroot\polyspace\bin\polyspace C:\Polyspace_Workspace\Results\^
ps_results.pscp

To rerun the verification, you must rerun it from the command line.

Input Arguments

`sourceFiles` — Comma-separated names of C or C++ files to analyze
source file name or path

Comma-separated C or C++ source file names, specified as a string. If the files are not in the current folder (`pwd`), `sourceFiles` must include a full or relative path. To avoid errors because of paths with spaces, add quotes `"  "` around the path. For more information, see `-sources`.

If your current folder contains a `sources` subfolder with the source files, you can omit the `-sources` flag. The verification considers files in `sources` and all subfolders under `sources`.

Example: `myFile.c`, `"C:\mySources\myFile1.c,C:\mySources\myFile2.c"`

`listOfSources` — Text file listing names of C or C++ files to analyze
sources list file name or path

Text file which lists the name of C or C++ files, specified as a string. If the files are not in the current folder (`pwd`), `listOfSources` must include a full or relative path. To avoid errors because of paths with spaces, add quotes `"  "` around the path. For more information, see `-sources-list-file`.

Example: `filename.txt`, `"C:\ps_analysis\source_files.txt"`

`[OPTIONS]` — Analysis option and corresponding value
command-line flag with optional value

Analysis options and their corresponding values, specified by the option name and if applicable value. For syntax specifications, see the individual analysis option reference pages.

Example: `-lang C-CPP`, `-target i386`

`optFile` — Text file listing analysis options and values
options file name or path

Text file listing analysis options and values, specified as a string. For more information, see `-options-file`.

Example: `opts.txt`, `"C:\ps_analysis\options.txt"`

See Also
`polyspaceCodeProver`

Topics
"Run Polyspace Analysis from Command Line"
"Analysis Options in Polyspace Code Prover"

Introduced in R2013b

polyspace-configure

(DOS/UNIX) Create Polyspace project from your build system at the DOS or UNIX command line

Syntax

`polyspace-configure buildCommand`

`polyspace-configure [OPTIONS] buildCommand`

`polyspace-configure [OPTIONS] -compilation-database jsonFile`

Description

`polyspace-configure buildCommand` traces your build system and creates a Polyspace project with information gathered from your build system.

`polyspace-configure [OPTIONS] buildCommand` traces your build system and uses `-option value` to modify the default operation of `polyspace-configure`. Specify the modifiers before `buildCommand`, otherwise they are considered as options in the build command itself.

`polyspace-configure [OPTIONS] -compilation-database jsonFile` creates a Polyspace project with information gathered from the JSON compilation database file `jsonFile` that you provide. You do not need to specify a build command or trace your build system. For more on JSON compilation databases, see JSON Compilation Database.

Examples

Create Polyspace Project from Makefile

This example shows how to create a Polyspace project if you use the command `make targetName buildOptions` to build your source code.

Create a Polyspace project specifying a unique project name. Use the `-B` or `-W makefileName` option with `make` so that the all prerequisite targets in the makefile are remade.

```
polyspace-configure  -prog myProject \
make -B targetName buildOptions
```

Open the Polyspace project in the Polyspace user interface.

Create Polyspace Options File from JSON Compilation Database

This example shows how to create a Polyspace options file from a JSON compilation database that you generate with the CMake build system generator. CMake generates build instructions for the build tool you specify, such as a Unix Makefiles for `make` or project files for Microsoft Visual Studio. CMake supports the generation of a JSON compilation database only for Makefile generators and Ninja generator. For more information, see makefile generators.

Generate a JSON compilation database for your CMake project. For an example of a Cmake project, see `polyspaceroot\help\toolbox\bugfinder\examples\compilation_database` where `polyspaceroot` is your Polyspace installation folder.

Navigate to the root of your project source tree. This folder contains the file `CMakeLists.txt` which CMake uses as an input to generate build instructions. Enter these commands:

```
mkdir JSON_cdb
cd JSON_cdb
cmake -G "Unix Makefiles" -DCMAKE_EXPORT_COMPILE_COMMANDS=1 ../
```

The last command generates a Unix makefile with build instructions for the `make` build tool. The command also outputs file `compile_commands.json`. This file lists the compiler calls for every translation unit in your project.

Generate a Polyspace options file from the compilation database that you generated in the previous step.

```
polyspace-configure -compilation-database compile_commands.json \
-output-options-file options.txt
```

You do not need to specify a build command and `polyspace-configure` does not trace your build. Polyspace extracts information about your build system from the JSON compilation database.

Pass the options file to Polyspace to run an analysis, for instance:

```
polyspace-code-prover -options-file options.txt
```

Create Projects That Have Different Source Files from Same Build Trace

This example shows how to create different Polyspace projects from the same trace of your build system. You can specify which source files to include for each project.

Trace your build system without creating a Polyspace project by specifying the option `-no-project`. To ensure that all the prerequisite targets in your makefile are remade, use the appropriate `make` build command option, for instance `-B`.

```
polyspace-configure -no-project make -B
```

`polyspace-configure` stores the cache information and the build trace in default locations inside the current folder. To store the cache information and build trace in a different location, specify the options `-cache-path` and `-build-trace`.

Generate Polyspace projects by using the build trace information from the previous step. Specify a project name and use the `-include-sources` or `-exclude-sources` option to select which files to include for each project.

```
polyspace-configure -no-build -prog myProject \
-include-sources "glob_pattern"
```

glob_pattern is a glob pattern that corresponds to folders or files you filter in or out of your project. To ensure the shell does not expand the glob patterns you pass to `polyspace-configure`, enclose them in double quotes. For more information on the supported syntax for glob patterns, see "polyspace-configure Source Files Selection Syntax".

If you specified the options `-build-trace` and `-cache-path` in the previous step, specify them again.

Delete the trace file and cache folder.

```
rm -r polyspace_configure_cache polyspace_configure_built_trace
```

If you used the options `-build-trace` and `-cache-path`, use the paths and file names from those options.

Run Command-Line Polyspace Analysis from Makefile

This example shows how to run Polyspace analysis if you use the command `make targetName buildOptions` to build your source code. In this example, you use `polyspace-configure` to trace your build system but do not create a Polyspace project. Instead you create an options file that you can use to run Polyspace analysis from command-line.

Create a Polyspace options file specifying the `-output-options-file` command. Use the `-B` or `-W makefileName` option with `make` so that all prerequisite targets in the makefile are remade.

```
polyspace-configure -output-options-file\
 myOptions make -B targetName buildOptions
```

Use the options file that you created to run a Polyspace analysis at the command line:

```
polyspace-code-prover -options-file myOptions
```

Input Arguments

buildCommand — Command for building source code
build command

Build command specified exactly as you use to build your source code.

Example: `make -B`, `make -W makefileName`

[OPTIONS] — Options for changing default operation of polyspace-configure
single option starting with -, followed by argument | multiple space-separated option-argument pairs

Basic Options

Option	Argument	Description
`-prog`	Project name	Project name that appears in the Polyspace user interface. The default is `polyspace`. If you do not use the option `-output-project`, the `-prog` argument also sets the project name. **Example:** `-prog myProject` creates a project that has the name `myProject` in the user interface. If you do not use the option `-output-project`, the project name is also `myProject.psrprj`.

Option	Argument	Description
`-author`	Author name	Name of project author. **Example:** `-author jsmith`
`-output-project`	Path	Project file name and location for saving project. The default is the file `polyspace.psprj` in the current folder. **Example:** `-output-project ../myProjects/project1` creates a project `project1.psprj` in the folder with the relative path `../myProjects/`.
`-output-options-file`	File name	Option to create a Polyspace analysis options file. Use this file for command-line analysis using one of these commands: • `polyspace-bug-finder` • `polyspace-code-prover` • `polyspace-bug-finder-server` • `polyspace-code-prover-server` • `polyspace-bug-finder-access`

`-allow-build-error`	None	Option to create a Polyspace project even if an error occurs in the build process.
		
		If an error occurs, the build trace log shows the following message:
		
		`polyspace-configure (polyspaceConfigure)`
		    `ERROR: build command`
		    command_name `fail [status=`*status_value*`]`
		
		command_name is the build command name that you use and *status_value* is the non-zero exit status or error level that indicates which error occurred in your build process.
		
		This option is ignored when you use `-compilation-database`.

`-allow-overwrite`	None	Option to overwrite a project with the same name, if it exists. By default, `polyspace-configure (polyspaceConfigure)` throws an error if a project with the same name already exists in the output folder. Use this option to overwrite the project.

Option	Argument	Description

`-no-console-output`
 	
`-silent` (default)
 	
`-verbose`
	None	Option to suppress or display additional messages from running `polyspace-configure` (`polyspaceConfigure`).
		
		• `-no-console-output` – Suppress all outputs including errors and warnings.
		• `-silent` (default) – Show only errors and warnings.
		• `-verbose` – Show all messages.
		
		If you specify more than one of these options, the most verbose option is applied.
		
		These options are ignored if they are used in combination with `-easy-debug`.

`-help`	None	Option to display the full list of `polyspace-configure` (`polyspaceConfigure`) commands
`-debug`	None	Option to store debug information for use by MathWorks technical support. This option has been superseded by the option `-easy-debug`.
`-easy-debug`	Path	Option to store debug information for use by MathWorks technical support. After a `polyspace-configure` (`polyspaceConfigure`) run, the path provided contains a zipped file ending with `pscfg-output.zip`. If the run fails to create a complete Polyspace project or options file, send this zipped file to MathWorks Technical Support for further debugging. The zipped file does not contain source files traced in the build. See also "Errors in Project Creation from Build Systems".

Options to Create Multiple Modules

These options are not compatible with `-compilation-database`.

Option	Argument	Description
`-module`	None	Option to create a separate options file for each binary created in build system.
		You can only create separate options files for different binaries. You cannot create multiple modules in a Polyspace project (for running in the Polyspace user interface).
		Use this option only for build systems that use GNU and Visual C++ compilers.
		See also "Modularize Polyspace Analysis by Using Build Command".
`-output-options-path`	Path name	Location where generated options files are saved. Use this option together with the option `-module`.
		The options files are named after the binaries created in the build system.

Advanced Options

Option	Argument	Description
`-compilation-database`	Path and file name	Location and name of JSON compilation database (JSON CDB) file. You generate this file from your build system, for instance by using the flag `-DCMAKE_EXPORT_COMPILE_COMMANDS=1` with `cmake`. The file contains compiler calls for all the translation units in you projects. For more information, see JSON Compilation Database. `polyspace-configure` uses the content of this file to get information about your build system. The extracted compiler paths in the JSON CDB must be accessible from the path where you run `polyspace-configure`. You do not specify a build command when you use this option. The build systems and compilers support the generation of a JSON CDB: • CMake • Bazel • Clang • Ninja • Qbs • waf This option is not compatible with `-no-project` and with the options to create multiple modules. The cache control options, `-allow-build-error`, and `-no-build` are ignored when you use this option.
`-compiler-config`	Path and file name	Location and name of compiler configuration file. The file must be in a specific format. For guidance, see the existing configuration files in *polyspaceroot*\polyspace\configure\compiler_configuration\. For information on the contents of the file, see "Compiler Not Supported for Project Creation from Build Systems". **Example:** `-compiler-configuration myCompiler.xml`

Option	Argument	Description
`-no-project`	None	Option to trace your build system without creating a Polyspace project and save the build trace information. Use this option to save your build trace information for a later run of `polyspace-configure (polyspaceConfigure)` with the `-no-build` option. This option is not compatible with `-compilation-database`.
`-no-build`	None	Option to create a Polyspace project using previously saved build trace information. To use this option, you must have the build trace information saved from an earlier run of `polyspace-configure (polyspaceConfigure)` with the `-no-project` option. If you use this option, you do not need to specify the `buildCommand` argument. This option is ignored when you use `-compilation-database`.

Option	Argument	Description

`-no-sources`	None	Option to create a Polyspace options file that does not contain the source file specifications.
		
		Use this option when you intend to specify the source files by other means. For instance, you can use this option when:
		
		• Running Polyspace on AUTOSAR-specific code.
		
		You want to create an options file that traces your build command for the compiler options:
		
		`-output-options-file options.txt -no-sources`
		
		You later append this options file when extracting source file names from ARXML specifications and running the subsequent Code Prover analysis with `polyspace-autosar`
		
		`-extra-options-file options.txt`
		
		See also "Run Polyspace on AUTOSAR Code Using Build Command".
		
		• Running Polyspace in Eclipse™.
		
		Your source files are already specified in your Eclipse project. When running a Polyspace analysis, you want to specify an options file that has the compilation options only.

Option	Argument	Description

`-extra-project-options`	Options to use for subsequent Polyspace analysis. For instance, `"-stubbed-pointers-are-unsafe"`.	Options that are used for subsequent Polyspace analysis.
		
		Once a Polyspace project is created, you can change some of the default options in the project. Alternatively, you can pass these options when tracing your build command. The flag `-extra-project-options` allows you to pass additional options.
		
		Specify multiple options in a space separated list, for instance `"-allow-negative-operand-in-shift -stubbed-pointers-are-unsafe"`.
		
		Suppose you have to set the option `-stubbed-pointers-are-unsafe` for every Polyspace project created. Instead of opening each project and setting the option, you can use this flag when creating the Polyspace project:
		
		`-extra-project-options`
		`    "-stubbed-pointers-are-unsafe"`
		
		For the list of options available, see:
		
		• "Analysis Options in Polyspace Code Prover"
		
		If you are creating an options file instead of a Polyspace project from your build command, do not use this flag.

`-tmp-path`	Path	Location of folder where temporary files are stored.
`-build-trace`	Path and file name	Location and name of file where build information is stored. The default is `./polyspace_configure_build_trace.log`. **Example:** `-build-trace ../build_info/trace.log`

`-include-sources`
 	
`-exclude-sources`
	Glob pattern	Option to specify which source files `polyspace-configure` (`polyspaceConfigure`) includes in, or excludes from, the generated project. You can combine both options together.
		
		A source file is included if the file path matches the glob pattern that you pass to `-include-sources`.
		
		A source file is excluded if the file path matches the glob pattern that you pass to `-exclude-sources`.

Option	Argument	Description

`-print-included-sources`
 	
`-print-excluded-sources`
	None	Option to print the list of source files that `polyspace-configure (polyspaceConfigure)` includes in, or excludes from, the generated project. You can combine both options together. The output displays the full path of each file on a separate line.
		
		Use this option to troubleshoot the glob patterns that you pass to `-include-sources` or `-exclude-sources`. You can see which files match the pattern that you pass to `-include-sources` or `-exclude-sources`.

`-compiler-cache-path`	Folder path	Specify a folder path where `polyspace-configure` looks for or stores the compiler cache files. If the folder does not exist, `polyspace-configure` creates it.
		
		By default, Polyspace looks for and stores compiler caches under these folder paths:
		
		• **Windows**
		
		`%appdata%\Mathworks\R20xxY\Polyspace`
		• **Linux**
		
		`~/.matlab/R20xxY/Polyspace`
		• **Mac**
		
		`~/Library/Application Support/ MathWorks/MATLAB/R20xxY/Polyspace`
		
		R20xxY is the release version of your Polyspace product, for instance R2020b.

`-no-compiler-cache`	None	Use this option if you do not want Polyspace to cache your compiler configuration information or to use an existing cache for your compiler configuration. By default, the first time you run `polyspace-configure` with a particular compiler configuration, Polyspace queries your compiler for the size of fundamental types, compiler macro definitions, and other compiler configuration information then caches this information. Polyspace reuses the cached information in subsequent runs of `polyspace-configure` for builds that use the same compiler configuration.
`-reset-compiler-cache-entry`	None	Use this option to query the compiler for the current configuration and to refresh the entry in the cache file that corresponds to this configuration. Other compiler configuration entries in the cache are not updated.

Option	Argument	Description
`-clear-compiler-cache`	None	Use this option to delete all compiler configurations stored in the cache file. If you also specify a build command or `-compilation-database`, polyspace-configure computes and caches the compiler configuration information of the current run, except if you specify `-no-project` or `-no-compiler-cache`.

`-import-macro-definitions`	none
	
	`from-whitelist`
	
	`from-source-tokens`
		Use this option to specify how `polyspace-configure` queries the compiler for macro definitions.
		
		You can specify:
		
		• `none` — Polyspace does not query the compiler for macro definitions. You must provide the macro definitions manually.
		
		• `from-whitelist` — Polyspace uses an internal white list to query the compiler for macro definitions.
		
		Polyspace uses the white list by default when you use the option `-compilation-database`.
		
		• `from-source-tokens` (default, except if you use `-compilation-database`) — Polyspace uses every non-keyword token in your source code to query your compiler for macro definitions.

`-options-for-sources-delimiter`	A single character	Specify an option separator to use when multiple analysis options are associated with one source file using the `-options-for-sources` option. Typically, the `-options-for-sources` option uses a semicolon as separator. See also `-options-for-sources`.

Cache Control Options

These options are primarily useful for debugging. Use the options if `polyspace-configure (polyspaceConfigure)` fails and MathWorks Technical Support asks you to use the option and provide the cached files. Starting R2020a, the option `-easy-debug` provides an easier way to provide debug information. See "Contact Technical Support About Issues with Running Polyspace".

These options are ignored when you use `-compilation-database`.

Option	Argument	Description

`-no-cache`
 	
`-cache-sources` (default)
 	
`-cache-all-text`
 	
`-cache-all-files`
	None	Option to perform one of the following:
		
		• `-no-cache`: Not create a cache
		• `-cache-sources`: Cache text files temporarily created during build for later use by `polyspace-configure` (polyspaceConfigure).
		• `-cache-all-text`: Cache all text files including sources and headers.
		• `-cache-all-files`: Cache all files including binaries.
		
		Typically, you cache temporary files created by your build command to debug issues in tracing the command.

`-cache-path`	Path	Location of folder where cache information is stored. When tracing a Visual Studio build (`devenv.exe`), if you see the error: `path is too long` try using a shorter path for this option to work around the error. **Example:** `-cache-path ../cache`
`-keep-cache` `-no-keep-cache` (default)	None	Option to preserve or clean up cache information after `polyspace-configure` (polyspaceConfigure) completes execution. If `polyspace-configure` (polyspaceConfigure) fails, you can provide this cache information to technical support for debugging purposes.

See Also

Topics
"Requirements for Project Creation from Build Systems"
"Compiler Not Supported for Project Creation from Build Systems"
"Modularize Polyspace Analysis by Using Build Command"

Introduced in R2013b

polyspace-report-generator

(DOS/UNIX) Generate reports for Polyspace analysis results stored locally or on Polyspace Access

Syntax

```
polyspace-report-generator -template outputTemplate [OPTIONS]
polyspace-report-generator -generate-results-list-file [-results-dir
resultsFolder] [-set-language-english]
polyspace-report-generator -generate-variable-access-file [-results-dir
resultsFolder] [-set-language-english]
```

```
polyspace-report-generator -template outputTemplate -host hostName -run-id
runID> [ACCESS_OPTIONS] [OPTIONS]
polyspace-report-generator -generate-results-list-file -host hostName -run-id
runID [ACCESS_OPTIONS] [-set-language-english]
polyspace-report-generator -generate-variable-access-file -host hostName -
run-id runID [ACCESS_OPTIONS] [-set-language-english]
polyspace-report-generator -configure-keystore
```

Description

`polyspace-report-generator -template outputTemplate [OPTIONS]` generates a report by using the template `outputTemplate` for the local analysis results that you specify with `OPTIONS`.

By default, reports for results from `project-name` are stored as `project-name_report-name` in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

`polyspace-report-generator -generate-results-list-file [-results-dir resultsFolder] [-set-language-english]` exports the analysis results stored locally in `resultsFolder` to a tab-delimited text file. The file contains the result information available on the **Results List** pane in the user interface. For more information on the exported results list, see "View Exported Results".

By default, the results file for results from `project-name` is stored in the `PathToFolder \Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

For exporting results to a tab-delimited text file, the `polyspace-results-export` command is preferred.

`polyspace-report-generator -generate-variable-access-file [-results-dir resultsFolder] [-set-language-english]` exports the list of global variables in your code from the Code Prover analysis stored locally in `FOLDER` to a tab-delimited text file. The file contains the information available on the **Variable Access** pane in the user interface. For more information on the exported variables list, see "View Exported Variable List".

By default, the variables file for results from `project-name` is stored in the `PathToFolder \Polyspace-Doc` folder. `PathToFolder` is the results folder of `project-name`.

`polyspace-report-generator -template outputTemplate -host hostName -run-id runID> [ACCESS_OPTIONS] [OPTIONS]` generates a report by using the template

`outputTemplate` for the analysis results corresponding to run `runID` on Polyspace Access. `hostName` is the fully qualified host name of the machine that hosts Polyspace Access.

By default, reports for results from `project-name` are stored as `project-name_report-name` in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

`polyspace-report-generator -generate-results-list-file -host hostName -run-id runID [ACCESS_OPTIONS] [-set-language-english]` exports the analysis results corresponding to run `runID` on Polyspace Access to a tab-delimited text file. The file contains the result information available on the **Results List** pane in the Polyspace Access web interface. `hostName` is the fully qualified host name of the machine that hosts Polyspace Access. For more information on the exported results list, see "Results List" (Polyspace Code Prover Access).

By default, the results file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

For exporting results to a tab-delimited text file, the `polyspace-results-export` command is preferred.

`polyspace-report-generator -generate-variable-access-file -host hostName -run-id runID [ACCESS_OPTIONS] [-set-language-english]` exports the list of global variables in your code from the Code Prover analysis corresponding to run `runID` on Polyspace Access to a tab-delimited text file. The file contains the information available on the **Variable Access** pane in the Polyspace Access web interface. `hostName` is the fully qualified host name of the machine that hosts Polyspace Access. For more information on the exported variables list, see "View Exported Variable List".

By default, the variables file for results from `project-name` is stored in the `PathToFolder\Polyspace-Doc` folder. `PathToFolder` is the path from which you call the command.

`polyspace-report-generator -configure-keystore` configures the report generator to communicate with Polyspace Access over HTTPS.

Run this one-time configuration step if Polyspace Access is configured to use the HTTPS protocol and you do not have a Polyspace Bug Finder desktop license, or you have a desktop license but you have not configured the desktop UI to communicate with Polyspace Access over HTTPS. Before running this command, generate a client keystore to store the SSL certificate that Polyspace Access uses for HTTPS. See "Generate a Client Keystore" (Polyspace Code Prover Access).

Examples

Generate PDF Reports for Analysis Results Stored Locally

You can generate multiple reports for analysis results that you store locally.

Create a variable `template_path` to store the path to the report templates and create a variable `report_templates` to store a comma-separated list of templates to use.

```
SET template_path="C:\Program Files"\Polyspace\R2019a\toolbox\polyspace^
\psrptgen\templates\
SET report_templates=%template_path%\Developer.rpt,^
%template_path%\CodingStandards.rpt
```

Generate the reports from the templates that you specified in `report_templates` for analysis results of Polyspace project `myProject`.

```
 polyspace-report-generator -template %report_templates% ^
-results-dir C:\Polyspace_Workspace\myProject\Module_1\CP_Result ^
-format PDF
```

The command generates two PDF reports, `myProject_Developer.PDF` and `myProject_CodingStandards.PDF`. The reports are stored in `C:\Polyspace_Workspace\myProject\Module_1\CP_Result\Polyspace-Doc`. For more information on the content of the reports, see `Bug Finder and Code Prover report (-report-template)`.

Configure Report Generator with Client Keystore

If you configure Polyspace Access to use the HTTPS protocol, you must generate a client keystore where you store the SSL certificate that Polyspace Access uses, and configure `polyspace-report-generator` to use that keystore. See "Generate a Client Keystore" (Polyspace Code Prover Access). This one-time configuration enables the report generator to communicate with Polyspace Access over HTTPS.

To configure the report generator with a client keystore, use the `polyspace-report-generator -configure-keystore` command. Follow the prompts to provide the URL you use to log into Polyspace Access, the full path to the keystore file you generated, and the keystore password.

```
polyspace-report-generator -configure-keystore
Location: US, user name: jsmit, id: 62600@us-jsmith, print mode: false
Enter the Polyspace Access URL using form  http[s]://<host>:<port> :
https://myAccessServer:9443
Enter full path to client keystore file :
C:\R2019b\ssl\client-cert.jks
Enter client keystore password :

The keystore has been configured
```

You must run the keystore configuration command again if:

- The Polyspace Access URL changes, for instance if you use a different port number.
- The path to the keystore file changes.
- The keystore password changes.

Generate Report and Variables List from Polyspace Access

Note To generate reports of results on Polyspace Access at the command line, you must have a Polyspace Bug Finder Server or Polyspace Code Prover Server installation.

Suppose that you want to generate a report and export the variables list for the results of a Code Prover analysis stored on the Polyspace Access database.

To connect to Polyspace Access, provide a host name and your login credentials including your encrypted password. To encrypt your password, use the `polyspace-access` command and enter your user name and password at the prompt.

```
polyspace-access -encrypt-password
login: jsmith
```

```
password:
CRYPTED_PASSWORD LAMMMEACDMKEFELKMNDCONEAPECEEKPL
Command Completed
```

Store the login and encrypted password in a credentials file and restrict read and write permission on this file. Open a text editor, copy these two lines in the editor, then save the file as `myCredentials.txt` for example.

```
 -login jsmith
 -encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL
```

To restrict the file permissions, right-click the file and select the **Permissions** tab on Linux or the **Security** tab on Windows.

To specify project results on Polyspace Access, specify the run ID of the project. To obtain a list of projects with their latest run IDs, use the `polyspace-access` with option `-list-project`.

```
polyspace-access -host myAccessServer -credentials-file myCredentials.txt -list-project
Connecting to https://myAccessServer:9443
Connecting as jsmith
Get project list with the last Run Id
Restricted/Code_Prover_Example (Code Prover) RUN_ID 14
public/Bug_Finder_Example (Bug Finder) RUN_ID 24
public/CP/Code_Prover_Example (Polyspace Code Prover) RUN_ID 16
public/Polyspace (Code Prover) RUN_ID 28
Command Completed
```

For more information on this command, see `polyspace-access`.

Generate a `Developer` report for results with run ID `16` from the Polyspace Access instance with host name `myAccessServer`. The URL of this instance of Polyspace Access is `https://myAccessServer:9443`.

```
SET template_path=^
"C:\Program Files\Polyspace\R2019a\toolbox\polyspace\psrptgen\templates"

polyspace-report-generator -credentials-file myCredentials.txt ^
-template %template_path%\Developer.rpt ^
-host myAccessServer ^
-run-id 16 ^
-output-name myReport
```

The command creates report `myReport.docx` by using the template that you specify. The report is stored in folder `Polyspace-Doc` on the path from which you called the command.

Generate a tab-delimited text file that contains a list of global variables in your code for the specified analysis results.

```
polyspace-report-generator -credentials-file myCredentials.txt^
-generate-variable-access-file ^
-host myAccessServer ^
-run-id 16
```

The list of global variables `Variable_View.txt` is stored in the same folder as the generated report. For more information on the exported variables list, see "View Exported Variable List".

Input Arguments

outputTemplate — path to report template file
string

Path to the report template that you use to generate an analysis report. To generate multiple reports, specify a comma-separated list of report template paths (do not put a space after the commas). The templates are available in *polyspaceroot*\toolbox\polyspace\psrptgen\templates\ as .rpt files. Here, *polyspaceroot* is the Polyspace installation folder. For more information on the available templates, see `Bug Finder and Code Prover report (-report-template)`.

This option is not compatible with `-generate-variable-access-file` and `-generate-results-list-file`.

Example: `C:\Program Files\Polyspace\R2019a\toolbox\polyspace\psrptgen\templates\Developer.rpt`

Example: `TEMPLATE_PATH\BugFinder.rpt,TEMPLATE_PATH\CodingStandards.rpt`

resultsFolder — Analysis results folder path
string

Path to a folder containing Polyspace analysis results (`.psbf` or `.pscp` file). To generate a report for multiple verifications, specify a comma-separated list of folder paths (do not put a space after the commas). If you do not specify a folder path, the command generates a report for analysis results in the current folder.

Example: `C:\Polyspace_Workspace\My_project\Module_1\results`

Example: `C:\Polyspace_Workspace\My_project\Module_2\results,C:\Polyspace_Workspace\My_project\Module_3\other_results`

hostName — Polyspace Access machine host name
string

Fully qualified host name of the machine that hosts the Polyspace Access **Gateway API** service. You must specify a host name to generate a report for results on the Polyspace Access database.

Example: `my-company-server`

runID — Polyspace Access run ID
integer

Run ID of the project findings for which you generate a report. Polyspace assigns a unique run ID to each analysis run that you upload to the Polyspace Access.

You can see the run ID of a project in the Polyspace Access web interface. To get the run ID, use the command `polyspace-access` with option `-list-project`.

OPTIONS — Options for generated report
string

Option	Description
`-format` *HTML* | *PDF* | *WORD*	File format of the report that you generate. By default, the command generates a WORD document. To generate reports in multiple formats, specify a comma-separated list of formats. (Do not put a space after the commas). For instance, `-format PDF,HTML`. This option is not compatible with `-generate-variable-access-file` and `-generate-results-list-file`.
`-output-name` *outputName*	Name of the generated report or folder name if you generate multiple reports. The report or exported file is saved on the path from which you call the command. To save in a different folder, specify the full path to the folder, for instance `-output-name C:\PathTo\OtherFolder`.
`-results-dir` *FOLDER_1,...,FOLDER_N*	Path to the locally stored results folder. To generate reports for multiple analyses, specify a comma-separated list of folder path. (Do not put a space after the commas). For example: `-results-dir folderPath1,folderPath2`
`-set-language-english`	Generate the report in English. Use this option if your display language is set to another language.
`-h`	Display the help information.

ACCESS_OPTIONS — Options for Polyspace Access
string

Option	Description
`-host` *HOST_NAME*	HOST_NAME corresponds to the host name that you specify in the URL of the Polyspace Access interface, for example `https://`*HOST_NAME*`:`*port*`/metrics/index.html`. If you are unsure about which host name to use, contact your Polyspace Access administrator. The default host name is `localhost`. This option is mandatory when you generate reports for results stored on the Polyspace Access database.

Option	Description
`-run-id` *RUN_ID*	Run ID of the project. Polyspace assigns a unique run ID to each analysis run that you upload. To get the last run ID of a project, use the `-list-project` option of the `polyspace-access` command. For more information on the command, see `polyspace-access`. This option is mandatory when you generate reports for results stored on the Polyspace Access database.
`-all-units`	Specify this option to generate a report for all units from a `unit by unit` analysis. When you use this option, specify the run ID of only one unit with `-run-id`. The command includes the other units from the analysis in the report.
`-port` *portNumber*	`portNumber` corresponds to the port number that you specify in the URL of the Polyspace Access interface, for example `https://`*hostname*`:`*portNumber*`/metrics/index.html`. If you are unsure about which port number to use, contact your Polyspace Access administrator. The default port number is `9443`.
`-protocol` *http*	*https*	HTTP protocol used to connect to Polyspace Access. Default value is `https`.

Option	Description

`-credentials-file` *file_path*	Full path to the text file where you store your login credentials. Use this option if, for instance, you use a command that requires your Polyspace Access credentials in a script but you do not want to store your credentials in that script. While the script runs, someone inspecting currently running processes cannot see your credentials.
	
	You can store only one set of credentials in the file, either as `-login` and `-encrypted-password` entries on separate lines, for instance:
	
	`-login jsmith`
	`-encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL`
	
	or as an `-api-key` entry:
	
	`-api-key keyValue123`
	
	Make sure that you restrict the read and write permissions on the file where you store your credentials. For example, to restrict read and write permissions on file `login.txt` in Linux, use this command:
	
	`chmod go-rwx login.txt`

`-api-key` *keyValue*	API key you use as a login credential instead of providing your login and encrypted password. To assign an API key to a user, see or contact your Polyspace Access administrator. Use the API key if, for instance, you use a command that requires your Polyspace Access login credentials as part of an automation script with a CI tool like Jenkins. If a user updates his or her password, you do not need to update the API key associated with that user in your scripts. It is recommended that you store the API key in a file and pass that file to the command by using `-credentials-file`.
`-login` *username* `-encryted-password` *ENCRYPTED_PASSWD*	Credentials that you use to log into Polyspace Access. The argument of `-encrypted-password` is the output of the `polyspace-access -encrypt-password` command. For more information on the command, see `polyspace-access`.

Tips

You cannot use the `polyspace-report-generator` command with results generated with Polyspace as You Code. Use the `polyspace-results-export` command instead. See `polyspace-results-export`.

See Also

`polyspace-results-export`

Introduced in R2013b

polyspace-results-export

(DOS/UNIX) Export Polyspace results to external formats such as CSV or JSON

Syntax

```
polyspace-results-export -format exportFormat -results-dir resultsFolder [
OPTIONS]
```

```
polyspace-results-export -format exportFormat -host hostName -run-id runID [
OPTIONS] [ACCESS_OPTIONS]
```

Description

`polyspace-results-export -format exportFormat -results-dir resultsFolder [OPTIONS]` exports Polyspace analysis results stored locally in `resultsFolder` to an external format `exportFormat`.

`polyspace-results-export -format exportFormat -host hostName -run-id runID [OPTIONS] [ACCESS_OPTIONS]` exports Polyspace analysis results stored in Polyspace Access to an external format `exportFormat`. Specify the Polyspace Access instance by using `hostName` and the project on Polyspace Access by using `runID`.

Examples

Export Polyspace Results Stored Locally

Export results from a project `myProject` in the Polyspace user interface to JSON format.

Suppose your project is stored in `C:\Polyspace_Workspace`. To export results from a specific module in the project, specify the path to the folder that directly contains results from the module.

```
polyspace-results-export -format json-sarif ^
-results-dir C:\Polyspace_Workspace\myProject\Module_1\BF_Result
-output-name C:\Polyspace_Workspace\reports\myProject\myProject.json
```

Export Results Stored in Polyspace Access

Note To generate reports of results on Polyspace Access at the command line, you must have a Polyspace Bug Finder Server or Polyspace Code Prover Server installation.

Suppose that you want to export the results of a project on Polyspace Access to JSON format.

To connect to Polyspace Access, provide a host name and your login credentials including your encrypted password. To encrypt your password, use the `polyspace-access` command and enter your user name and password at the prompt.

```
polyspace-access -encrypt-password
login: jsmith
password:
CRYPTED_PASSWORD LAMMMEACDMKEFELKMNDCONEAPECEEKPL
Command Completed
```

Store the login and encrypted password in a credentials file and restrict read and write permission on this file. Open a text editor, copy these two lines in the editor, then save the file as `myCredentials.txt` for example.

```
 -login jsmith
 -encrypted-password LAMMMEACDMKEFELKMNDCONEAPECEEKPL
```

To restrict the file permissions, right-click the file and select the **Permissions** tab on Linux or the **Security** tab on Windows.

To specify project results on Polyspace Access, specify the run ID of the project. To obtain a list of projects with their latest run IDs, use the `polyspace-access` with option `-list-project`.

```
polyspace-access -host myAccessServer -credentials-file myCredentials.txt -list-project
Connecting to https://myAccessServer:9443
Connecting as jsmith
Get project list with the last Run Id
Restricted/Code_Prover_Example (Code Prover) RUN_ID 14
public/Bug_Finder_Example (Bug Finder) RUN_ID 24
public/CP/Code_Prover_Example (Polyspace Code Prover) RUN_ID 16
public/Polyspace (Code Prover) RUN_ID 28
Command Completed
```

For more information on this command, see `polyspace-access`.

If Polyspace Access uses the HTTPS protocol, use the `polyspace-report-generator` binary to configure `polyspace-results-export` to enable communications with Polyspace Access over HTTPS. See "Configure Report Generator with Client Keystore" on page 5-34.

Export results from the project with run ID 16 to JSON format.

```
polyspace-results-export -credentials-file myCredentials.txt
-format json-sarif^
-host myAccessServer ^
-run-id 16
```

Input Arguments

exportFormat — Output format for results
csv | json-sarif | console

Format in which the Polyspace results are exported: `csv` for CSV output or `json-sarif` for JSON output. If you use the `polyspace-bug-finder-access` command for single-file analysis (Polyspace as You Code), you can also export the results to the console using the format `console`.

Each result consists of information such as result name, family, and so on. Both CSV and JSON formats result in almost the same content being exported but the exported content might refer to the same type of information by different names.

* In the CSV format, each result consists of tab-separated information in columns such as `ID`, `Family`, `Group`, `Color`, `Check`, and so on.

  To package and potentially filter your result data, use the CSV format. For instance, you can import the CSV file to Microsoft Excel® and use Excel filters on the results.

- In the JSON format, each result consists of almost the same information as JSON object properties. The properties shown for a result sometimes use a name that is different from the name used in the CSV format. For instance, to get the full rule checker name for a result, use the `ruleId` property of a result in combination with the `id` and `name` property of a rule. The reason for the difference is that the JSON format follows the standard notation provided by the OASIS Static Analysis Results Interchange Format (SARIF).

    The JSON format contains some additional information such as the checker short name and the full message that accompanies a result. Use the JSON format if you want to use this short name or message. You can also use this format for a more standardized reporting of results. For instance, if you use several static analysis tools and want to report their results in one interface by using a single parsing algorithm, you can export all the results to the standard SARIF JSON format.

The console output is preformatted in a form similar to compiler errors and warnings, and contains less information than the other formats. In particular, if you baseline Polyspace as You Code results using integration results in the Polyspace Access web interface, use the JSON or CSV format for maximum benefits from the baselining. See "Baseline Polyspace as You Code Results on Command Line" (Polyspace Bug Finder Access).

resultsFolder — Result folder path
string

Path to a folder containing Polyspace analysis results (`.psbf` or `.pscp` file). If you do not specify a folder path, the command generates a report for analysis results in the current folder.

Example: `C:\Polyspace_Workspace\My_project\Module_1\results`

hostName — Polyspace Access machine host name
string

`hostName` corresponds to the host name that you specify in the URL of the Polyspace Access interface, for example `https://hostName:port/metrics/index.html`. If you are unsure about which host name to use, contact your Polyspace Access administrator. The default host name is `localhost`. You must specify a host name to generate a report for results on the Polyspace Access database.

Example: `my-company-server`

runID — Polyspace Access run ID
integer

Run ID of the project findings that you export. A unique run ID is assigned to each analysis run that you upload to Polyspace Access.

You can see the run ID of a project in the Polyspace Access web interface. To get the run ID of a project at the command line, use the command `polyspace-access` with option `-list-project`.

OPTIONS — Additional options for exporting results
string

Option	Description
`-output-name` *outputName*	Name of the exported file. The default name is `results_list.txt` (CSV) or `results_list.json` (JSON). The file is saved on the path from which you call the command. To save the file in a different folder, specify the full path to the folder, for instance `-output-name C:\PathTo\OtherFolder`.
`-set-language-english`	Use this option if your display language is set to a language other than English but you want the exported results in English.

To see options available with this command, enter `polyspace-results-export -h`.

ACCESS_OPTIONS — Additional options for exporting results from Polyspace Access
string

Option	Description
`-port` *portNumber*	`portNumber` corresponds to the port number that you specify in the URL of the Polyspace Access interface, for example `https://`*hostname*`:`*portNumber*`/metrics/index.html`. If you are unsure about which port number to use, contact your Polyspace Access administrator. The default port number is `9443`.
`-protocol` *http* | *https*	HTTP protocol to connect to Polyspace Access. Default value is `https`.

Option	Description

`-credentials-file` *file_path*	Full path to the text file where you store your login credentials. Use this option if, for instance, you use a command that requires your Polyspace Access credentials in a script but you do not want to store your credentials in that script. While the script runs, someone inspecting currently running processes cannot see your credentials.
	
	You can store only one set of credentials in the file, either as `-login` and `-encrypted-password` entries on separate lines, for instance:
	
	`-login jsmith`
	`-encrypted-password`
	`  LAMMMEACDMKEFELKMNDCONEAPECEEKPL`
	
	or as an `-api-key` entry:
	
	`-api-key keyValue123`
	
	Make sure that you restrict the read and write permissions on the file where you store your credentials. For example, to restrict read and write permissions on file `login.txt` in Linux, use this command:
	
	`chmod go-rwx login.txt`

`-api-key` *keyValue*	API key you use as a login credential instead of providing your login and encrypted password. To assign an API key to a user, see or contact your Polyspace Access administrator. Use the API key if, for instance, you use a command that requires your Polyspace Access login credentials as part of an automation script with a CI tool like Jenkins. If a user updates his or her password, you do not need to update the API key associated with that user in your scripts. It is recommended that you store the API key in a file and pass that file to the command by using `-credentials-file`.
`-login` *username* `-encryted-password` *ENCRYPTED_PASSWD*	Credentials that you use to log into Polyspace Access. The argument of `-encrypted-password` is the output of the `polyspace-access -encrypt-password` command. For more information on the command, see `polyspace-access`.

See Also

Introduced in R2020b

polyspace-results-repository

(DOS/UNIX) Upload, download and otherwise interact with results in the Polyspace Metrics repository

Syntax

```
polyspace-results-repository -upload resultsFolder -product productName -prog
projectName -verif-version versionNumber [OPTIONS]

polyspace-results-repository -download resultsFolder -product productName -
prog projectName -verif-version versionNumber [OPTIONS]

polyspace-results-repository -get-projects-list -product productName
polyspace-results-repository -get-versions-list -product productName -prog
projectName
polyspace-results-repository -get-run-numbers-list -product productName -prog
projectName -verif-version versionNumber
polyspace-results-repository -get-files-list -product productName -prog
projectName -verif-version versionNumber [OPTIONS]

polyspace-results-repository -get-sqo-id -product productName -prog
projectName -verif-version versionNumber [OPTIONS]
polyspace-results-repository -set-sqo-id SQOLevel -product productName -prog
projectName -verif-version versionNumber [OPTIONS]

polyspace-results-repository -delete -product productName -prog projectName -
verif-version versionNumber [OPTIONS]
polyspace-results-repository -rename -product productName -new-prog
newProjectName -new-verif-version newVersionNumber -prog projectName -verif-
version versionNumber [OPTIONS]
```

Description

`polyspace-results-repository -upload resultsFolder -product productName -prog projectName -verif-version versionNumber [OPTIONS]` uploads Polyspace results in `resultsFolder` to the Polyspace Metrics web repository.

You can customize the default upload with additional options.

`polyspace-results-repository -download resultsFolder -product productName -prog projectName -verif-version versionNumber [OPTIONS]` downloads Polyspace results from the Polyspace Metrics web repository to `resultsFolder`.

You can customize the default download with additional options.

`polyspace-results-repository -get-projects-list -product productName` displays the Bug Finder or Code Prover projects currently in the Polyspace Metrics web repository.

`polyspace-results-repository -get-versions-list -product productName -prog projectName` displays the versions of a project currently in the Polyspace Metrics web repository. If the project involves file-by-file verification in Code Prover, add the `-unit-by-unit` option.

`polyspace-results-repository -get-run-numbers-list -product productName -prog projectName -verif-version versionNumber` displays the run numbers of a project version currently in the Polyspace Metrics web repository.

The option is useful only if multiple results with the same project name and version number have been uploaded to Polyspace Metrics.

`polyspace-results-repository -get-files-list -product productName -prog projectName -verif-version versionNumber [OPTIONS]` displays the files involved in the results for a certain project and version.

`polyspace-results-repository -get-sqo-id -product productName -prog projectName -verif-version versionNumber [OPTIONS]` displays the Software Quality Objectives being applied to a certain project and version.

`polyspace-results-repository -set-sqo-id SQOLevel -product productName -prog projectName -verif-version versionNumber [OPTIONS]` applies Software Quality Objectives specified by `SQOLevel` to a certain project and version.

`polyspace-results-repository -delete -product productName -prog projectName -verif-version versionNumber [OPTIONS]` deletes a certain project version from the Polyspace Metrics web repository.

`polyspace-results-repository -rename -product productName -new-prog newProjectName -new-verif-version newVersionNumber -prog projectName -verif-version versionNumber [OPTIONS]` renames a certain project version to another project and version.

Examples

Upload Results to Polyspace Metrics

Suppose you want to upload Code Prover results from the folder `C:\My_Results` to the Polyspace Metrics server `localhost:12427`. You want the project name to appear as `Polyspace_Project` and the version number `1.0`.

Upload the results using this information.

```
polyspace-results-repository -upload "C:\My_Results" \
                             -prog "Polyspace_Project" \
                             -verif-version "1.0" \
                             -server "localhost:12427" \
                             -product "CodeProver"
```

Download Results from Polyspace Metrics

Suppose you want to download Bug Finder results in version `1.0` of the project `Polyspace_Project` from the Polyspace Metrics server `localhost:12427`. You want the results to be downloaded to the folder `C:\My_Results`.

Download the results using this information.

```
polyspace-results-repository -download "C:\My_Results" \
                             -prog "Polyspace_Project" \
                             -verif-version "1.0" \
                             -server "localhost:12427" \
                             -product "BugFinder"
```

Upload Results of Multiple Modules to Polyspace Metrics

If a Polyspace project consists of multiple modules, you can upload the analysis results for all modules to the Polyspace Metrics interface.

For instance, if you run `polyspace-autosar`, a separate module is created for each AUTOSAR Software Component. You can write a shell script (`.sh` file) like this (or a Windows `.bat` file) to collect result files in subfolders of the project folder and upload them to Polyspace Metrics. Code Prover result files use extension `.pscp`.

```
#! /bin/bash
# Upload all results from a polyspace-autosar run to a Metrics server.
MODULES_DIR=`find "$RESULTS_DIR" -name ps_results.pscp -printf '%h\n'`
IFS='
'
for module in $MODULES_DIR; do
    # extract module name from its path foo/bar/behavior_name
    module_name=${module#*AUTOSAR/}
    # transform it to foo.bar.behavior_name
    module_name=${module_name//\//.}
    polyspace-results-repository          \
        -f                                \
        -server localhost                 \
        -upload "$module"                 \
        -prog APPLICATION_NAME            \
        -module $module_name              \
        -verif-version "$RESULTS_VERSION"
done
```

Input Arguments

`resultsFolder` — Folder containing Polyspace results
string

Folder name, specified as a string (in double quotes). The folder must contain a Bug Finder result file (`.psbf`) or a Code Prover file (`.pscp`).

Example: `"C:\Polyspace_Projects\Proj1\Module_1\BF_Result"`, `"C:\AUTOSAR\Demo\polyspace\AUTOSAR\pkg\tst002\swc002\bhv\verification"`

`projectName` — Name of Polyspace project
string

Name of Polyspace project, as it appears on Polyspace metrics.

Project	Product	Mode	Language	Latest Version	Date	Status
	▼	▼	▼	▼		▼
AUTOSAR	Code Prover	Integration	C	1.1	Dec 20, 2017	completed (PASS2)
Crypto	Bug Finder		C/C++	1.0	Dec 27, 2017	completed

Example: "Polyspace_project"

newProjectName — Name of Polyspace project
string

New name of Polyspace project, as it appears on Polyspace metrics.

Example: "Polyspace_project_1"

versionNumber — Version number of Polyspace project
string

Version number of Polyspace project, as it appears on the **Runs** tab of Polyspace metrics.

					Projects	Runs
ID	Project	Product ᴬ	Mode	Language	Version	Date
		▼	▼	▼		
20	AUTOSAR	Code Prover	Integration	C	1.1	Dec 27, 2017
9	⊞ AUTOSAR	Code Prover	Integration	C	1.0	Dec 12, 2017

Example: "1.0"

newVersionNumber — Version number of Polyspace project
string

New version number of Polyspace project, as it appears on the **Runs** tab of Polyspace metrics.

Example: "1.1"

productName — Name of product used for analysis
"CodeProver" (default) | "BugFinder"

Name of product used for producing the results, specified as "BugFinder" or "CodeProver".

SQOLevel — SQO Level or BF-QO Level to be applied to analysis results
"SQO-1" | "SQO-2" | "SQO-3" | "SQO-4" | "SQO-5" | "SQO-6" | "BF-QO-1" | "BF-QO-2" | "BF-QO-3" | "BF-QO-4" | "BF-QO-5" | "BF-QO-6" | "Exhaustive"

Quality levels applied to analysis results. The quality levels consist of a set of criteria based on which the analysis results are assigned a status of **PASS** or **FAIL**. Use the SQO levels for Code Prover results and BF-QO level for Bug Finder results.

See:

• "Software Quality Objectives"
• "Bug Finder Quality Objective Levels"

[OPTIONS] — Options to customize upload or download
option name

Option	Description
-server *serverName:portNumber*	Explicitly specify a server name and port number for upload or download, for instance, "localhost:12427". By default, results are uploaded to or downloaded from the server that you configured in Polyspace preferences. See "Set Up Polyspace Metrics".
-f	Use this option in scripts so that the polyspace-results-repository command does not require user interaction. By default, the command asks for confirmation before transferring results from your local folder to Polyspace Metrics or vice versa.
-password *password_value*	Specify the password for uploading or download a password-protected result in Polyspace Metrics.
-module *module_name*	Specify that the result belongs to a module in the current Polyspace project. Specify a module name. Use this option to upload results from a project with multiple modules. In Polyspace Metrics, all modules with the same -prog value appear under the same project. When you upload the results of multiple modules in the same project, they appear as separate modules in Polyspace Metrics. When you download the result of a specific module, the result appears in a subfolder of the download folder.
-run-number	If you uploaded multiple results with the same project name and version number, they appear as separate runs in Polyspace metrics. Use this option to upload or download the results for a specific run.
-integration or -unit-by-unit	If you run a file-by-file verification, use -unit-by-unit to upload or download all results together. Otherwise, use -integration. For more information on file-by-file verification, see Verify files independently (-unit-by-unit). By default, the command assumes one result for each upload or download.

Introduced in R2013b

polyspace-comments-import

(DOS/UNIX) Import review information from previous Polyspace analysis

Syntax

```
polyspace-comments-import -diff-rte prevResultsFolder currentResultsFolder [-
print-new-results] [-overwrite-destination-comments]
```

Description

`polyspace-comments-import -diff-rte prevResultsFolder currentResultsFolder [-print-new-results] [-overwrite-destination-comments]` imports review information from a results file in `prevResultsFolder` to `currentResultsFolder`. The review information includes the severity, status and additional notes that you assign to a result.

Besides importing the review information, the command also shows the number of results where review information could not be imported either because the result changed or the result already had new review information. If you use the option `-print-new-results`, you see this information:

- Number of new results in current results folder, that is, results not present in previous results folder
- Number of results removed from previous results folder, that is, results no longer present in the current results folder
- Number of results in current results folder that do not have associated review information

You can also use this command to create a baseline for the analysis results. In the Polyspace user interface, if you click the **New** button, only the analysis results that are new compared to the baseline remain in the results list.

If you use the option `-overwrite-destination-comments`, newer review information on previously existing results are overwritten with previous review information. For instance, if the same result has a different status in the current and previous results folder, after using the `polyspace-comments-import` command:

- Without the option, the result in the current results folder retains its status.
- With the option, the status of the result in the current results folder is overwritten with the status from the previous results folder.

Examples

Import Review Information from Previous Polyspace Results

Run Bug Finder on a sample file and add some review information. Then, run Bug Finder a second time and import the information from the previous run.

Copy the file `numerical.c` from *polyspaceroot*\polyspace\examples\cxx \Bug_Finder_Example\sources to a writable folder. Open a command window and navigate to the folder (using `cd`). Run Bug Finder on the file and save results in the subfolder `Run_1`:

```
polyspace-bug-finder -sources numerical.c -results-dir Run_1/
```

Depending on the product installed, you can also run `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server`.

Open the results file in the `Run_1` subfolder:

```
polyspace Run_1/ps_results.psbf
```

Select a result. On the **Result Details** window, select a **Severity** and **Status** and add some notes. You will import this review information to results from a later analysis.

Run Bug Finder again, but save the results in a different subfolder `Run_2`:

```
polyspace-bug-finder -sources numerical.c -results-dir Run_2/
```

You can open the results file in `Run_2` and see that there is no review information.

Import the review information from the results file in the `Run_1` subfolder to the `Run_2` subfolder. Add the option `-print-new-results` to see the number of new, removed and unreviewed results.

```
polyspace-comments-import -diff-rte Run_1/ Run_2/ -print-new-results
```

Open the results file in the `Run_2` subfolder:

```
polyspace Run_2/ps_results.psbf
```

You see the review information imported from the results file in the `Run_1` subfolder.

Input Arguments

prevResultsFolder — Folder containing previous Polyspace results with review information
string

Path to a folder containing a Polyspace results file (`.psbf` file for Bug Finder results and `.pscp` file for Code Prover results). The results are presumably from an earlier Polyspace analysis and contain review information that will be imported to a later results file.

Example: `"C:\Polyspace\Project_1_Run_25"`

currentResultsFolder — Folder containing later Polyspace results
string

Path to a folder containing Polyspace results (`.psbf` file for Bug Finder results and `.pscp` file for Code Prover results). The results are presumably from a later Polyspace analysis and have no review information or review information for new results only. You want to import review information from an earlier Polyspace analysis to these results.

Example: `"C:\Polyspace\Project_1_Run_26"`

See Also
`-import-comments`

Topics
"Import Review Information from Previous Polyspace Analysis"

Introduced in R2013b

pslinkfun

Manage model analysis at the command line

Syntax

```
pslinkfun('annotations','type',typeValue,'kind',kindValue,Name,Value)
```

```
pslinkfun('openresults',systemName)
```

```
pslinkfun('settemplate',psprjFile)
prjTemplate = pslinkfun('gettemplate')
```

```
pslinkfun('advancedoptions')
pslinkfun('enablebacktomodel')
pslinkfun('help')
pslinkfun('metrics')
pslinkfun('jobmonitor')
pslinkfun('stop')
```

Description

`pslinkfun('annotations','type',typeValue,'kind',kindValue,Name,Value)` adds an annotation of type `typeValue` and kind `kindValue` to the selected block in the model. You can specify a different block using a `Name,Value` pair argument. You can also add notes about a severity classification, an action status, or other comments using `Name,Value` pairs.

In the generated code associated with the annotated block, Polyspace adds code comments before and after the lines of code. Polyspace reads these comments and marks Polyspace results of the specified `kind` with the annotated information.

Syntax limitations:

- You can have only one annotation per block. If a block produces both a rule violation and an error, you can annotate only one type.

- Even though you apply annotations to individual blocks, the scope of the annotation can be larger. The generated code from one block can overlap with another, causing the annotation to also overlap.

    For example, consider this model. The first summation block has a Polyspace annotation, but the second does not.

    However, the associated generated code adds all three inputs in one line of code.

```
/* polyspace:begin<RTE:OVFL:Medium:To Fix>*/
annotate_y.Out1=(annotate_u.In1+annotate_U.In2)+annotate_U.In3;
/* polyspace:end<RTE:OVFL:Medium:To Fix> */
```

Therefore, the annotation justifies both summations.

`pslinkfun('openresults',systemName)` opens the Polyspace results associated with the model or subsystem `systemName` in the Polyspace environment.

`pslinkfun('settemplate',psprjFile)` sets the configuration file for new verifications.

`prjTemplate = pslinkfun('gettemplate')` returns the template configuration file used for new analyses.

`pslinkfun('advancedoptions')` opens the advanced verification options window to configure additional options for the current model.

`pslinkfun('enablebacktomodel')` enables the back-to-model feature of the Simulink plug-in. If your Polyspace results do not properly link to back to the model blocks, run this command.

`pslinkfun('help')` opens the Polyspace documentation in a separate window. Use this option for only pre-R2013b versions of MATLAB.

`pslinkfun('metrics')` opens the Polyspace Metrics interface.

`pslinkfun('jobmonitor')` opens the Polyspace Job Monitor to display remote verifications in the queue.

`pslinkfun('stop')` kills the code analysis that is currently running. Use this option for local analyses only.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Examples

Annotate a Block and Run a Polyspace Code Prover Verification

Use the Polyspace annotation function to annotate a block and see the annotation in the verification results.

In the example model WhereAreTheErrors, set the current block to the division block of the `10* x // (x-y)` subsystem. Then, add an annotation to the current block to mark division by zero (DIV) errors as justified with the annotation.

```
model = 'WhereAreTheErrors';
open(model)
gcb = 'WhereAreTheErrors/10* x // (x-y)/Divide';
pslinkfun('annotations','type','RTE','kind','ZDV','status',...
    'justified','comment','verified not an error')
```

In Simulink, the division block of the `10* x // (x-y)` subsystem now has a Polyspace annotation.

At the command line, generate code for the model and run a verification. After the analysis is finished, open the result in the Polyspace environment:

```
slbuild(model)
pslinkrun(model)
pslinkfun('openresults',model)
```

If you look at the orange division by zero error, the check is justified and includes the status and comments from your annotation.

Add Batch Options to Default Configuration Template

Change advanced Polyspace options and set the new configuration as a template.

Load the model `WhereAreTheErrors` and open the advanced options window.

```
model = 'WhereAreTheErrors';
load_system(model)
pslinkfun('advancedoptions')
```

In the **Run Settings** pane, select the options **Run Code Prover analysis on a remote cluster** and **Upload results to Polyspace Metrics**.

Set the configuration template for new Polyspace analyses to have these options.

```
pslinkfun('settemplate',fullfile(cd,'pslink_config',...
    'WhereAreTheErrors_config.psprj'))
```

View the current Polyspace template.

```
template = pslinkfun('gettemplate')

template =
C:\ModelLinkDemo\pslink_config\WhereAreTheErrors_config.psprj
```

View Polyspace Queue and Metrics

Run a remote analysis, view the analysis in the queue, and review the metrics.

Before performing this example, check that your Polyspace configuration is set up for remote analysis and Polyspace Metrics.

Build the model `WhereAreTheErrors`, create a Polyspace options object, set the verification mode, and open the advanced options window.

```
model = 'WhereAreTheErrors';
load_system(model)
slbuild(model)
opts = pslinkoptions(model);
opts.VerificationMode = 'CodeProver';
pslinkfun('advancedoptions')
```

In the **Run Settings** pane, select the options **Run Code Prover analysis on a remote cluster** and **Upload results to Polyspace Metrics**.

Run Polyspace, then open the Job Monitor to monitor your remote job.

```
pslinkrun(model,opts)
pslinkfun('jobmonitor')
```

After your job is finished, open the metrics server to see your job in the repository.

```
pslinkfun('metrics')
```

Input Arguments

typeValue — type of result
'RTE' | 'MISRA-C' | 'MISRA-AC-AGC' | 'MISRA-CPP' | 'JSF'

The type of result with which to annotate the block, specified as:

- 'RTE' for run-time errors.
- 'MISRA-C' for MISRA C coding rule violations (C code only).
- 'MISRA-AC-AGC' for MISRA C coding rule violations (C code only).
- 'MISRA-CPP' for MISRA C++ coding rule violations (C++ code only).
- 'JSF' for JSF C++ coding rule violations (C++ code only).

Example: 'type','MISRA-C'

kindValue — specific check or coding rule
check acronym | rule number

The specific check or coding rule specified by the acronym of the check or the coding rule number. For the specific input for each type of annotation, see the following table.

type Value	kind Values
'RTE'	Use the abbreviation associated with the type of check that you want to annotate. For example, 'UNR' – Unreachable Code. For the list of possible checks, see: "Run-Time Checks".
'MISRA-C'	Use the rule number that you want to annotate. For example, '2.2'. For the list of supported MISRA C rules and their numbers, see "Supported MISRA C:2004 and MISRA AC AGC Rules".
'MISRA-AC-AGC'	Use the rule number that you want to annotate. For example, '2.2'. For the list of supported MISRA AC AGC rules and their numbers, see "Supported MISRA C:2004 and MISRA AC AGC Rules".
'MISRA-CPP'	Use the rule number that you want to annotate. For example, '0-1-1'. For the list of supported MISRA C++ rules and their numbers, see "MISRA C++:2008 Rules".
'JSF'	Use the rule number that you want to annotate. For example, '3'. For the list of supported JSF C++ rules and their numbers, see "Supported JSF C++ Coding Rules".

Example: `pslinkfun('annotations','type','MISRA-CPP','kind','1-2-3')`

Data Types: `char`

systemName — Simulink model
system | subsystem

Simulink model specified by the system or subsystem name.

Example: `pslinkfun('openresults','WhereAreTheErrors')`

psprjFile — Polyspace project file
standard Polyspace template (default) | absolute path to `.psprj` file

Polyspace project file specified as the absolute path to the `.psprj` project file. If `psprjFile` is empty, Polyspace uses the standard Polyspace template file. New Polyspace projects start with this project configuration.

Example: `pslinkfun('settemplate', fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', 'Bug_Finder_Example.bf.psprj'));`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'block','MyModel\Sum', 'status','to fix'`

block — block to be annotated
`gcb` (default) | block name

The block you want to annotate specified by the block name. If you do not use this option, the block returned by the function `gcb` is annotated.

Example: `'block','MyModel\Sum'`

class — severity of the check
`'high'` | `'medium'` | `'low'` | `'unset'`

Severity of the check specified as `high`, `medium`, `low`, or `unset`.

Example: `'class','high'`

status — action status
`'unreviewed'` | `'to investigate'` | `'to fix'` | `'justified'` | `'no action planned'` | `'not a defect'` | `'other'`

Action status of the check specified as `unreviewed`, `to investigate`, `to fix`, `justified`, `no action planned`, `not a defect`, or `other`.

The statuses, `justified`, `not a defect`, and `no action planned` also mark the result as justified.

Example: `'status','no action planned'`

comment — additional comments
character vector

Additional comments specified as a character vector. The comments provide more information about why the results are justified.

Example: `'comment','defensive code'`

See Also
pslinkrun | pslinkoptions | gcb

Introduced in R2014a

pslinkoptions

Create an options object to customize configuration of a Simulink model, generated code or a S-Function block. Use the object to specify configuration options for these Simulink objects in a Polyspace run from the MATLAB command line

Syntax

```
opts = pslinkoptions(codegen)
opts = pslinkoptions(model)
opts = pslinkoptions(sfunc)
```

Description

`opts = pslinkoptions(codegen)` returns an options object with the configuration options for code generated by `codegen`.

`opts = pslinkoptions(model)` returns an options object with the configuration options for the Simulink model.

`opts = pslinkoptions(sfunc)` returns an options object with the configuration options for the S-Function.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Examples

Use a Simulink model to create and edit an options objects

Load `closed_loop_control` and create a Polyspace® options object from the model:

```
load_system('closed_loop_control');
model_opt = pslinkoptions('closed_loop_control')

model_opt =

                    ResultDir: 'results_$ModelName$'
           VerificationSettings: 'PrjConfig'
             OpenProjectManager: 0
           AddSuffixToResultDir: 0
        EnableAdditionalFileList: 0
             AdditionalFileList: {}
                VerificationMode: 'CodeProver'
            EnablePrjConfigFile: 0
                   PrjConfigFile: ''
            AddToSimulinkProject: 0
                  InputRangeMode: 'DesignMinMax'
                  ParamRangeMode: 'None'
                 OutputRangeMode: 'None'
```

```
               ModelRefVerifDepth: 'Current model only'
        ModelRefByModelRefVerif: 0
                     AutoStubLUT: 1
         CxxVerificationSettings: 'PrjConfig'
        CheckConfigBeforeAnalysis: 'OnWarn'
            VerifAllSFcnInstances: 0
```

The model is already configured for Embedded Coder®, so only the Embedded Coder configuration options appear. Change the results folder name option and set `OpenProjectManager` to true.

```
model_opt.ResultDir = 'results_v1_$ModelName$';
model_opt.OpenProjectManager = true

model_opt =

                       ResultDir: 'results_v1_$ModelName$'
            VerificationSettings: 'PrjConfig'
               OpenProjectManager: 1
            AddSuffixToResultDir: 0
         EnableAdditionalFileList: 0
               AdditionalFileList: {}
                 VerificationMode: 'CodeProver'
             EnablePrjConfigFile: 0
                    PrjConfigFile: ''
             AddToSimulinkProject: 0
                    InputRangeMode: 'DesignMinMax'
                    ParamRangeMode: 'None'
                   OutputRangeMode: 'None'
               ModelRefVerifDepth: 'Current model only'
        ModelRefByModelRefVerif: 0
                     AutoStubLUT: 1
         CxxVerificationSettings: 'PrjConfig'
        CheckConfigBeforeAnalysis: 'OnWarn'
            VerifAllSFcnInstances: 0
```

Create and edit an options object for Embedded Coder at the command line

Create a Polyspace® options object called `new_opt` with Embedded Coder parameters:

```
new_opt = pslinkoptions('ec')

new_opt =

                       ResultDir: 'results_$ModelName$'
            VerificationSettings: 'PrjConfig'
               OpenProjectManager: 0
            AddSuffixToResultDir: 0
         EnableAdditionalFileList: 0
               AdditionalFileList: {}
                 VerificationMode: 'CodeProver'
             EnablePrjConfigFile: 0
                    PrjConfigFile: ''
             AddToSimulinkProject: 0
                    InputRangeMode: 'DesignMinMax'
                    ParamRangeMode: 'None'
                   OutputRangeMode: 'None'
```

```
         ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
                 AutoStubLUT: 1
        CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
```

To follow the progress in the Polyspace interface, set the `OpenProjectManager` option to true. Change the configuration to check for both checks and MISRA C® 2012 coding rule violations:

```
new_opt.OpenProjectManager = true;
new_opt.VerificationSettings = 'PrjConfigAndMisraC2012'

new_opt =

                      ResultDir: 'results_$ModelName$'
           VerificationSettings: 'PrjConfigAndMisraC2012'
              OpenProjectManager: 1
              AddSuffixToResultDir: 0
        EnableAdditionalFileList: 0
              AdditionalFileList: {}
                VerificationMode: 'CodeProver'
             EnablePrjConfigFile: 0
                   PrjConfigFile: ''
             AddToSimulinkProject: 0
                  InputRangeMode: 'DesignMinMax'
                  ParamRangeMode: 'None'
                 OutputRangeMode: 'None'
              ModelRefVerifDepth: 'Current model only'
         ModelRefByModelRefVerif: 0
                     AutoStubLUT: 1
           CxxVerificationSettings: 'PrjConfig'
       CheckConfigBeforeAnalysis: 'OnWarn'
```

Create and edit an options object for TargetLink at the command line

Create a Polyspace® options object called `new_opt` with TargetLink® parameters:

```
new_opt = pslinkoptions('tl')

new_opt =

                      ResultDir: 'results_$ModelName$'
           VerificationSettings: 'PrjConfig'
              OpenProjectManager: 0
              AddSuffixToResultDir: 0
        EnableAdditionalFileList: 0
              AdditionalFileList: {}
                VerificationMode: 'CodeProver'
             EnablePrjConfigFile: 0
                   PrjConfigFile: ''
             AddToSimulinkProject: 0
                  InputRangeMode: 'DesignMinMax'
                  ParamRangeMode: 'None'
                 OutputRangeMode: 'None'
                     AutoStubLUT: 1
```

Set the `OpenProjectManager` option to true to follow the progress in the Polyspace interface. Also change the configuration to check for both run-time errors and MISRA C® coding rule violations:

```
new_opt.OpenProjectManager = true;
new_opt.VerificationSettings = 'PrjConfigAndMisra'

new_opt =

                  ResultDir: 'results_$ModelName$'
          VerificationSettings: 'PrjConfigAndMisra'
            OpenProjectManager: 1
          AddSuffixToResultDir: 0
      EnableAdditionalFileList: 0
            AdditionalFileList: {}
              VerificationMode: 'CodeProver'
            EnablePrjConfigFile: 0
                  PrjConfigFile: ''
            AddToSimulinkProject: 0
                  InputRangeMode: 'DesignMinMax'
                  ParamRangeMode: 'None'
                OutputRangeMode: 'None'
                    AutoStubLUT: 1
```

Input Arguments

codegen — Code generator
`'ec'` | `'tl'`

Code generator, specified as either `'ec'` for Embedded Coder or `'tl'` for TargetLink®. Each argument creates a Polyspace options object with properties specific to that code generator.

For a description of all configuration options and their values, see pslinkoptions.

Example: `ec_opt = pslinkoptions('ec')`

Example: `tl_opt = pslinkoptions('tl')`

Data Types: `char`

model — Simulink model name
model name

Simulink model, specified by the model name. Creates a Polyspace options object with the configuration options of that model. If you have not set any options, the object has the default configuration options. If you have set a code generator, the object has the default options for that code generator.

For a description of all configuration options and their values, see pslinkoptions.

Example: `model_opt = pslinkoptions('my_model')`

Data Types: `char`

sfunc — path to S-Function
character vector

Path to S-Function, specified as a character vector. Creates a Polyspace options object with the configuration options for the S-function. If you have not set any options, the object has the default configuration options.

For a description of all configuration options and their values, see pslinkoptions.

Example: `sfunc_opt = pslinkoptions('path/to/sfunction')`

Data Types: `char`

Output Arguments

`opts` — Polyspace configuration options
options object

Polyspace configuration options, returned as an options object. The object is used with `pslinkrun` to run Polyspace from the MATLAB command line.

For the list of object properties, see pslinkoptions.

Example: `opts= pslinkoptions('ec')`
`opts.VerificationSettings = 'Misra'`

See Also
`pslinkfun` | pslinkoptions | `pslinkrun`

Introduced in R2012a

polyspacesetup

Integrate Polyspace installation with Simulink

Syntax

```
polyspacesetup('install')
polyspacesetup('install', 'polyspacefolder', folder)
polyspacesetup('install', 'polyspacefolder', folder, 'silent', isSilent)
polyspacesetup('uninstall')
polyspacesetup('showpolyspacefolders')
```

Description

`polyspacesetup('install')` integrates Polyspace from the default installation folder with MATLAB or Simulink. If you installed Polyspace in a nondefault folder, the function prompts you for the installation folder. See "Installation Folder".

To run MATLAB scripts for Polyspace analysis, install MATLAB and Polyspace in separate folders, and then integrate them by using this function. See also "Integrate Polyspace with MATLAB and Simulink".

`polyspacesetup('install', 'polyspacefolder', folder)` integrates Polyspace installed in the folder `folder` with MATLAB or Simulink.

`polyspacesetup('install', 'polyspacefolder', folder, 'silent', isSilent)` integrates Polyspace installed in the folder `folder` with MATLAB or Simulink silently when `isSilent` is `true` or with a prompt if `isSilent` is `false`. When you start MATLAB with the option `-batch`, `isSilent` is set to `true` by default. If you use a nondefault folder to install Polyspace and then do not specify the folder in `folder`, you are prompted to specify the install location even if you use `-batch` to start MATLAB.

`polyspacesetup('uninstall')` unlinks the currently linked installation of Polyspace from MATLAB.

`polyspacesetup('showpolyspacefolders')` lists all Polyspace folders that are linked to your current installation of MATLAB.

Examples

Integrate Polyspace with MATLAB or Simulink

To integrate Polyspace with MATLAB or Simulink, use the function `polyspacesetup`.

Open MATLAB with administrator or root privilege.

At the MATLAB command prompt, enter:

```
polyspacesetup('install');
```

If you install Polyspace in the default folder `C:\Program Files\Polyspace\R2021a`, the command integrates Polyspace with MATLAB. You might be prompted that the workspace will be cleared and all open models closed. Click **Yes**. The process might take a few minutes to complete. When you start MATLAB with the `-batch` option, the installation completes without any prompts .

If a Polyspace installation is not detected at the default location, you are prompted for the installation location. Use this command:

```
polyspacesetup('install', 'polyspaceFolder', Folder)
```

where *Folder* is the Polyspace installation folder.

Restart MATLAB.

Integrate Polyspace Noninteractively with MATLAB at Command Line by Using `-batch`

To integrate Polyspace with MATLAB in the command line noninteractively, start MATLAB with the startup option `-batch`. See "Commonly Used Startup Options".

When you start MATLAB with the startup option `-batch`, `polyspacesetup` is silent by default. That is, the function does not emit any messages unless there is any input error.

1 Open a Windows command-line prompt with administrator or root privilege.
2 To ensure that the integration takes place noninteractively, specify the install folder for Polyspace. At the command-line prompt, enter:

```
matlab -batch "polyspacesetup('install','polyspaceFolder',folder)"
```

where *folder* is the installation location of Polyspace.
3 If the integration is successful, this message is displayed:

```
Polyspace plug-in: installation complete.
Restart MATLAB before using Polyspace plug-in.
```

You can also enter the command in a script. For instance, you might have a script that performs the installations of MATLAB and Polyspace. Append the preceding command to your script to integrate MATLAB and Polyspace noninteractively.

Silently Integrate Polyspace with MATLAB or Simulink

To integrate Polyspace with MATLAB or Simulink silently, use the function `polyspacesetup`. By default, Polyspace is installed in the folder `C:\Program Files\Polyspace\R2021a`.

Open MATLAB with administrator or root privilege.

At the MATLAB command prompt, enter:

```
polyspacesetup('install', 'polyspaceFolder', Folder, 'silent', true);
```

where *Folder* is the Polyspace installation folder. The process might take a few minutes to complete.

Restart MATLAB.

Input Arguments

`folder` — Polyspace installation folder
C:\Program Files\Polyspace\R2021a | path to Polyspace installation

A character array containing the path to the Polyspace installation folder.

Example: `polyspacesetup('install','polyspacefolder','C:\Program Files\Polyspace\R2020b')`

Data Types: `char`

`isSilent` — Indicator for silent integration
`false` (default) | `true`

Specifies whether to integrate Polyspace with MATLAB or Simulink silently (true) or not (false). When you start MATLAB with the `-batch` option, this argument is set to `true` by default .

Example: `polyspacesetup('install','polyspacefolder','C:\Program Files\Polyspace\R2020b','silent',true)`

Data Types: `logical`

See Also
`polyspace.Project`

Topics
"Run Polyspace Analysis by Using MATLAB Scripts"

Introduced in R2019a

polyspacePackNGo

Generate and package options files to run Polyspace analysis on code generated from Simulink model

Syntax

```
archivePath = polyspacePackNGo(mdlName)
archivePath = polyspacePackNGo(mdlName,psOpt)
archivePath = polyspacePackNGo(mdlName,psOpt,asModelRef)
```

Description

`archivePath = polyspacePackNGo(mdlName)` examines the Simulink model `mdlName`, extracts Polyspace options files from it, and packages the options files in the zip file located at `archivePath`. Before using `polyspacePackNGo`, generate code from your Simulink model. Then archive the generated code, for instance, by using `packNGo`. Generate the Polyspace options files from the Simulink model and include them in the code archive by using `polyspacePackNGo`. In a different development environment, when running a Polyspace analysis of the generated code, use the options files included in the code archive to preserve model-specific information, such as design range specifications. You must have Embedded Coder to use `slbuild`.

`archivePath = polyspacePackNGo(mdlName,psOpt)` generates and packages the Polyspace options files that are generated according to the specification in `psOpt`. The object `psOpt` must be a Polyspace options object that is generated by using `pslinkoptions`. Using `psOpt`, modify the options for the Polyspace analysis.

`archivePath = polyspacePackNGo(mdlName,psOpt,asModelRef)` generates and packages the Polyspace options files by using `asModelRef` to specify whether to generate option files for model reference code or standalone code.

Examples

Generate and Package Polyspace Options Files

To generate and package Polyspace options files for a Simulink model, use `polyspacePacknGo`.

Open the Simulink model `rtwdemo_counter` and specify a folder for storing the generated code.

```
% Open the model
mdlName = 'rtwdemo_counter';
open_system(mdlName);
% Specify a folder for generated code
codegenFolder = 'rtwdemo_counter_ert_rtw';
```

To enable packing the generated code in an archive, set the option `PackageGeneratedCodeAndArtifacts` to `true`. Specify the name of the generated code archive as `genCodeArchive.zip`.

```
configSet = getActiveConfigSet(mdlName);
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true);
set_param(configSet, 'PackageName', 'genCodeArchive.zip');
```

To make the model compatible with Polyspace, set `SystemTargetFile` to `etr.tlc`.

```
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

After configuring the model, generate code.

```
if exist(fullfile(pwd,codegenFolder), 'dir') == 0
    slbuild(mdlName)
end
```

Because `PackageGeneratedCodeAndArtifacts` is set to `true`, the generated code is packed into the archive `genCodeArchive.zip`.

Generate and package Polyspace options files.

```
zipFile = polyspacePackNGo(mdlName);
```

In the code archive `genCodeArchive.zip`, the Polyspace options files are packaged in the `polyspace` folder.

Package Polyspace Options Files That Have Specific Polyspace Analysis Options

To specify the Polyspace analysis options when packaging and generating options files, use `pslinktoptions`.

Open the Simulink model `rtwdemo_counter` and configure the model for generating a code archive that is compatible with Polyspace.

```
% Open the model
mdlName = 'rtwdemo_counter';
open_system(mdlName);
% Specify a folder for generated code
codegenFolder = 'rtwdemo_counter_ert_rtw';
configSet = getActiveConfigSet(mdlName);
% Enable packing the generated code into an archive
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true);
% Specify a name for the code archive
set_param(configSet, 'PackageName', 'genCodeArchive.zip');
% Configure the model to be Polyspace Compatible
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

After configuring the model, generate code.

```
if exist(fullfile(pwd,codegenFolder), 'dir') == 0
    slbuild(mdlName)
end
```

Because `PackageGeneratedCodeAndArtifacts` is set to `true`, the generated code is packed into the archive `genCodeArchive.zip`.

To specify the model configuration for the Polyspace analysis, use a `pslinkoptions` object. Create this object by using the function `pslinkoptions`.

```
psOpt = pslinkoptions(mdlName);
```

The object `psopt` is a structure where the fields are model configurations that you can specify.

Specify the model configuration by using `psOpt` object. For instance, set `InputRangeMode` to full range. For a full options list, see the input argument `psOpt`.

```
psOpt.InputRangeMode = 'FullRange';
```

Generate and package Polyspace options files. Use the `psOpt` object as the second argument in `polyspacePacknGo`.

```
zipFile = polyspacePackNGo(mdlName,psOpt);
```

In the code archive `genCodeArchive.zip`, the Polyspace options files are packaged in the `polyspace` folder. The file `optionsFile.txt` contains the specified Polyspace analysis options.

Package Polyspace Options Files for Code Generated as a Model Reference

To accelerate model simulations, invoke referenced Simulink models as simulation targets. To generate model reference simulation targets from a Simulink model, generate code from the model by using `slbuild` with the build process specified as `ModelReferenceCoderTargetOnly`. Then, package the generated code by using `packNGo`. To generate and package Polyspace options files for analyzing such code, use the function `polyspacePacknGo` with the optional argument `asModelRef` set to `true`.

Open the Simulink model `rtwdemo_counter` and configure the model for generating a code archive that is compatible with Polyspace.

```
% Load model
mdlName = 'rtwdemo_counter';
load_system(mdlName);
configSet = getActiveConfigSet(mdlName);
% Enable packing the generated code into an archive
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true);
set_param(configSet, 'PackageName', '');
% Configure the model to be Polyspace Compatible
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

After configuring the model, generate a model reference simulation target from it by using the function `slbuild`. Specify the option `ModelReferenceCoderTargetOnly`. See `slbuild`.

```
slbuild(mdlName,'ModelReferenceCoderTargetOnly');
```

The code that is generated is stored in the folder `slprj`.

To package the code that is generated as a model reference, use the function `packNGo`. Locate the file `buildinfo.mat` in *<working folder>*/slprj/ert/rtwdemo_counter and use the full path to it as the input to `packNGo`. This command generates an archive containing the generated code and the object `buildinfo.mat`. See `packNGo`.

```
% Locate buildinfo and generate code archive
buildinfo = fullfile(pwd,'slprj','ert',mdlName,'buildinfo.mat');
packNGo(buildinfo)
```

Generate and package Polyspace options files. Omit the optional second argument. Set the third argument `asModelRef` to `true`.

```
zipFile = polyspacePackNGo(mdlName,[],true);
```

In the code archive `rtwdemo_counter.zip`, the Polyspace options files are packaged in the `polyspace` folder.

Input Arguments

`mdlName` — Name of Simulink model for which to generate Polyspace options files
model name

A character array containing the name of the model for which you want to generate and package the Polyspace options files.

Example: `polyspacePackNGo('rtwdemo_roll')`

Data Types: `char`

`psOpt` — Polyspace options object
options associated with `model` (default) | object created by using `pslinkoptions`

Specifies the model configuration for the Polyspace analysis by using a `pslinkoptions` object. You can modify certain analysis options by modifying `psOpt`, which is a structure where individual fields represent analysis options. For a fill list of options that you can modify, see the table **Polyspace Analysis Options Supported by `polyspacePacknGo`**.

Polyspace Analysis Options Supported by `polyspacePacknGo`

Property	Value	Description
EnableAdditionalFileList: Enable an additional file list to be analyzed, specified as `true` or `false`. Use with the `AdditionalFileList` option.	`true`	Polyspace verifies additional files specified in the `AdditionalFileList` option.
	`false` (default)	Polyspace does not verify additional files.
AdditionalFileList: List of additional files to be analyzed specified as a cell array of files. To add these files to the analysis, use the `EnableAdditionalFileList` option.	cell array	Polyspace considers the listed files for verification.
InputRangeMode: Specifies the range of the input variables.	`'DesignMinMax'` (default)	Polyspace uses the input range defined in the workspace or a block.
	`'Fullrange'`	Polyspace uses full range inputs.
ParamRangeMode: Specifies the range of the constant parameters.	`'DesignMinmax'`	Polyspace uses the constant parameter range defined in the workspace or in a block.
	`'None'` (default)	Polyspace uses the value of parameters specified in the code.
OutputRangeMode: Specifies the output assertions.	`'DesignMinMax'`	Polyspace applies assertions to outputs by using a range defined in a block or the workspace.
	`'None'` (default)	Polyspace does not apply assertions to the output variables.
ModelRefVerifDepth: Specify the depth for analyzing the models that are referenced by the current model.	`'Current model Only'` (default)	Polyspace analyzes only the top model without analyzing the referenced models. Use this option when you refer to models that do not need to be analyzed, such as library models.

Property	Value	Description
	`'1'`|`'2'`|`'3'`	Polyspace analyzes referenced models up to the specified depth in the reference hierarchy. To analyze the models that are referenced by the top model, specify the property `ModelRefVerifDepth` as `'1'`. To analyze models that are referenced by the first level of references, specify this property as `'2'`.
	`'All'`	Polyspace verifies all referenced models.
ModelRefByModelRefVerif: Specify whether you want to analyze all referenced models together or individually.	true	Polyspace analyzes the top model and the referenced models together. Use this option to check for integration or scaling issues.
	`false` (default)	Polyspace analyzes the top model and the referenced models individually.
AutoStubLUT: Specifies how lookup tables are used.	`true` (default)	Polyspace stubs the lookup tables and verifies the model without analyzing the lookup table code.
	`false`	Polyspace includes the lookup table code in the analysis.
CheckConfigBeforeAnalysis: Specifies the level of configuration checking done before the Polyspace analysis starts.	`'Off'`	Polyspace checks only for errors. The analysis stops if errors are found.
	`'OnWarn'` (default)	Polyspace stops the analysis when errors are found and displays a message when warnings are found.
	`'OnHalt'`	Polyspace stops the analysis when either errors or warnings are found.

Example: `polyspacePackNGo('rtwdemo_roll', psOpt)`, where `ps_opt` is an options object created by calling `pslinkoptions`

asModelRef — Indicator for model reference analysis
`false` (default) | `true`

Indicator for model reference analysis, specified as `true` or `false`.

- If `asModelRef` is `false` (default), the function generates options files so that Polyspace analyzes the generated code as standalone code.

- If `asModelRef` is `true`, the function generates options files so that Polyspace analyzes the generated code as model reference code.

> **Note** If you set `asModelRef` to `true`, use `slbuild` to generate code.

Example: `polyspacePackNGo('rtwdemo_roll', psOpt,true)`

Data Types: `logical`

Output Arguments

archivePath — The full path to the archive containing the generated options files
path to archive

A character array containing the path to the generated archive. The options files are located in the `polyspace` folder in the archive. The `polyspace` folder contains these options files:

- `optionsFile.txt`: a text file containing the Polyspace options required to run a Polyspace analysis on the generated code without losing model-specific information, such as design range specification.
- *model*`_drs.xml`: A file containing the design range specification of the model.
- `linkdata.xml`: A file that links the generated code to the components of the model.

To run a Polyspace analysis on the generated code in an environment that is different than the environment where the code was generated from the Simulink model, use these files.

Data Types: `char`

See Also
`pslinkoptions` | `slbuild`

Topics
"Run Polyspace Analysis on Generated Code by Using Packaged Options Files"
"Run Polyspace Analysis by Using MATLAB Scripts"
pslinkoptions Properties

Introduced in R2020b

pslinkrun

Run Polyspace analysis on model, system, or S-Function

Syntax

```
[polyspaceFolder, resultsFolder] = pslinkrun
[polyspaceFolder, resultsFolder]= pslinkrun(target)
[polyspaceFolder, resultsFolder] = pslinkrun('-slcc',target)
[polyspaceFolder, resultsFolder] = pslinkrun(target, opts)
[polyspaceFolder, resultsFolder] = pslinkrun('-slcc', target, opts)
[polyspaceFolder, resultsFolder] = pslinkrun(target, opts, asModelRef)
[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder,
opts)
```

Description

`[polyspaceFolder, resultsFolder] = pslinkrun` analyzes code generated from the current system using the configuration options associated with the current system. It returns the location of the results folder. The current system is the system returned by the command `bdroot`.

`[polyspaceFolder, resultsFolder]= pslinkrun(target)` analyzes `target` with the configuration options associated with the model containing `target`. Before you run an analysis, you must:

- Generate code for models and subsystems.
- Compile S-Functions.

`[polyspaceFolder, resultsFolder] = pslinkrun('-slcc',target)` runs Polyspace on C/C++ custom code included in C Caller blocks and Stateflow charts in the model.

`[polyspaceFolder, resultsFolder] = pslinkrun(target, opts)` analyzes `target` using the configuration options specified in the object `opts`. It returns the location of the results folder.

`[polyspaceFolder, resultsFolder] = pslinkrun('-slcc', target, opts)` runs Polyspace on C/C++ custom code included in C Caller blocks and Stateflow charts in the model. The analysis uses the configuration options specified in the object `opts`.

`[polyspaceFolder, resultsFolder] = pslinkrun(target, opts, asModelRef)` uses `asModelRef` to specify which type of generated code to analyze—standalone code or model reference code. This option is useful when you want to analyze code that is generated as model reference. Code that is generated as model reference is intended to be called or used in other models or code.

`[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder, opts)` runs Polyspace on C/C++ code generated from MATLAB code and stored in `codegenFolder`.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Examples

Analyze Generated Code

Use a Simulink model to generate code, set configuration options, and then run an analysis from the command line.

```matlab
% Generate code from the model WhereAreTheErrors.
model = 'WhereAreTheErrors';
load_system(model);
slbuild(model);

% Create a Polyspace options object from the model.
opts = pslinkoptions(model);

% Set properties that define the Polyspace analysis.
opts.VerificationMode = 'CodeProver';
opts.VerificationSettings = 'PrjConfigAndMisraC2012';

% Run Polyspace using the options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,opts);
bdclose(model);
```

The results and the corresponding Polyspace project are saved to the `results_WhereAreTheErrors` folder, listed in the `polyspaceFolder` variable. The full path to the results folder is in the `resultsFolder` variable.

Analyze Referenced Model Code

Use a Simulink model to generate model reference code, set configuration options, and then run an analysis from the command line.

```matlab
% Generate code from the model WhereAreTheErrors.
% Treat WhereAreTheErrors as if referenced by another model.
model = 'WhereAreTheErrors';
load_system(model);
slbuild(model,'ModelReferenceCoderTargetOnly');

% Create a Polyspace options object from the model.
opts = pslinkoptions(model);

% Set properties that define the Polyspace analysis.
opts.VerificationMode = 'CodeProver';
opts.VerificationSettings = 'PrjConfigAndMisraC2012';

% Run Polyspace with the options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,opts,true);
bdclose(model);
```

The results and corresponding Polyspace project are saved to the
`results_mr_WhereAreTheErrors` folder, listed in the `polyspaceFolder` variable. The full path to
the results folder is in the `resultsFolder` variable.

Reuse Analysis Options for Multiple Models

This example shows how to reuse a subset of options for Polyspace analysis of multiple models.
Create a generic options object and specify properties that describe the common options. Associate
the generic options object with a model-specific options object. Optionally, set some model-specific
options and run the Polyspace analysis.

```matlab
% Generate code from the model WhereAreTheErrors.
model = 'psdemo_model_link_sl';
load_system(model);
slbuild(model);

% Create a generic options object to use for multiple model analyses.
opts = polyspace.ModelLinkOptions();
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';
opts.MergedReporting.ReportOutputFormat = 'PDF';
opts.MergedReporting.EnableReportGeneration = true;

% Create a model-specific options object.
mlopts = pslinkoptions(model);

% Create a project from the generic options object.
% Associate the project with the model-specific options object.
prjfile = opts.generateProject('model_link_opts');
mlopts.EnablePrjConfigFile = true;
mlopts.PrjConfigFile = prjfile;
mlopts.VerificationMode = 'BugFinder';

% Run Polyspace with the model-specific options object.
[polyspaceFolder, resultsFolder] = pslinkrun(model,mlopts);
bdclose(model);
```

After the analysis completes, results open automatically in the Polyspace interface.

Analyze C/C++ Code Generated from MATLAB Code

This example shows how to analyze C/C++ code generated from MATLAB code.

```matlab
% Generate code
codeName = 'average_filter';
matlabFileName = fullfile(polyspaceroot, 'help',...
    'toolbox','codeprover','examples','matlab_coder','averaging_filter.m');
codegenFolder = fullfile(pwd, 'codegenFolder');
codegen(matlabFileName, '-config:lib', '-c', '-args', ...
   {zeros(1,100,'double')}, '-d', codegenFolder);

% Configure Polyspace analysis
opts = pslinkoptions('ec');
```

```
opts.ResultDir = ['results_',codeName];
opts.OpenProjectManager = 1;

% Run Polyspace
[polyspaceFolder, resultsFolder] = pslinkrun('-codegenfolder', codegenFolder, opts);
```

After the analysis completes, results open automatically in the Polyspace interface.

Input Arguments

target — Target of the analysis
bdroot (default) | model or system name | path to S-Function block

Target of the analysis specified as a character vector, with the model, system, or S-function in single quotes. The default value is the system returned by bdroot.

If you analyze custom code in C Caller blocks and Stateflow charts using pslinkrun('-slcc',...), the argument target cannot be an S-Function block.

Example: [polyspaceFolder, resultsFolder] = pslinkrun('demo') where demo is the name of a model.

Example: [polyspaceFolder, resultsFolder] = pslinkrun('path/to/sfunction')

Data Types: char

opts — Configuration options
configuration options associated with target (default) | object created by pslinkoptions

Specify configuration options of target, specified as a Polyspace options object. The function pslinkoptions creates such an options object. You can customize the options object by changing the pslinkoption properties.

Example: pslinkrun('demo', opts_demo) where demo is the name of a model and opts_demo is an options object.

asModelRef — Indicator for model reference analysis
false (default) | true

Indicator for model reference analysis, specified as true or false.

- If asModelRef is false (default), Polyspace analyzes the generated code as stand-alone code. This option is equivalent to choosing **Verify Code Generated For > Model** in the Simulink Polyspace options.
- If asModelRef is true, Polyspace analyzes the generated code as a model reference code. This option is equivalent to choosing **Verify Code Generated For > Referenced Model** in the Simulink Polyspace options. Specifying model reference code indicates that Polyspace must look for the generated code in a different location from the location for standalone code.

Data Types: logical

codegenFolder — Folder containing generated C/C++ code
character vector

Folder containing C/C++ code generated from MATLAB code, specified as a character vector. You specify this folder with the codegen command using the flag -d.

Output Arguments

polyspaceFolder — Folder containing Polyspace project and results
character vector

Name of the folder containing Polyspace project and results, specified as a character vector. The default value of this variable is `results_$ModelName$`.

To change this value, see "Output folder" on page 12-16.

resultsFolder — Full path to subfolder containing Polyspace results
character vector

Full path to subfolder containing Polyspace results, specified as a character vector.

The folder `results_$ModelName$` contains your Polyspace project and a subfolder `$ModelName$` with the analysis results. This variable gives you the full path to the subfolder. You can use this path with a `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object.

To change the parent folder `results_$ModelName$`, see "Output folder" on page 12-16.

See Also
`pslinkfun` | `pslinkoptions` | pslinkoptions

Topics
"Run Polyspace Analysis on Code Generated from Simulink Model"
"Run Polyspace Analysis on S-Function Code"
"Run Polyspace Analysis on Custom Code in C Caller Blocks and Stateflow Charts"
"Recommended Model Configuration Parameters for Polyspace Analysis"

Introduced in R2012a

pslinkrunCrossRelease

Analyze C/C++ code generated by R2020b or newer Embedded Coder versions by using a different version of Polyspace that is more recent than the Simulink version

Syntax

```
[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem)
[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem,
psOpt)
[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem,
psOpt,asModelRef)
[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem,
psOpt,asModelRef,OptionsFile)
```

Description

[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem) runs a Polyspace analysis of the code generated from `ModelOrSubsystem` by using Embedded Coder from an earlier release of Simulink.

[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem, psOpt) runs a Polyspace analysis of the code generated from `ModelOrSubsystem` through an earlier release of Simulink. The analysis uses the model configuration options that are specified in the `pslinkoptions` object `psOpt`.

[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem, psOpt,asModelRef) runs a Polyspace analysis of the code generated as a model reference from `ModelOrSubsystem` through an earlier release of Simulink.The analysis uses `asModelRef` to specify which type of generated code to analyze—standalone code or model reference code.

[polyspaceFolder, resultsFolder] = pslinkrunCrossRelease(ModelOrSubsystem, psOpt,asModelRef,OptionsFile) runs a Polyspace analysis of the code generated from `ModelOrSubsystem` through an earlier release of Simulink. The analysis uses the Polyspace analysis options specified in the options file `OptionsFile`.

Examples

Analyze Code Generated by Using Earlier Simulink Release

To run a Polyspace analysis on code generated by using an earlier release of Simulink, use the function `pslinkrunCrossRelease`. The analysis uses the configuration options associated with `ModelOrSubsystem`. The Simulink release must be R2020b or later. Before you run an analysis, you must integrate Polyspace with Simulink. See "Integrate Polyspace with MATLAB or Simulink Installation from Earlier Release".

1 Open the Simulink model `rtwdemo_roll` and configure the model for code generation.

```
% Load the model
model = 'rtwdemo_roll';
```

```
load_system(model);
% Configure the Solver
configSet = getActiveConfigSet(model);
set_param(configSet,'Solver','FixedStepDiscrete');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

2 To enable packing the generated code in an archive, set the option `PackageGeneratedCodeAndArtifacts` to `true`.

```
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true)
```

3 Create temporary folders for code generation and generate code.

```
[TEMPDIR, CGDIR] = rtwdemodir();
slbuild(model);
```

4 Start a Polyspace analysis.

```
% Run Polyspace analysis
[~,resultsFolder] = pslinkrunCrossRelease(model);
bdclose(model);
```

The character vector `resultsFolder` contains the full path to the results folder.

Run Polyspace Analysis with Modified Configuration While Analyzing Code Generated by Using Earlier Simulink Release

To run a Polyspace analysis with modified model configurations, use a `pslinkoptions` object. For a list of model configurations related to Polyspace analysis that you can modify, see the table **Polyspace Configuration Parameters Supported by pslinkrunCrossRelease** on this page. Before you run an analysis, you must integrate Polyspace with Simulink. See "Integrate Polyspace with MATLAB or Simulink Installation from Earlier Release".

1 Open the Simulink model `rtwdemo_roll` and configure the model for code generation.

```
% Load the model
model = 'rtwdemo_roll';
load_system(model);
% Configure the Solver
configSet = getActiveConfigSet(model);
set_param(configSet,'Solver','FixedStepDiscrete');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

2 To enable packing the generated code in an archive, set the option `PackageGeneratedCodeAndArtifacts` to `true`.

```
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true)
```

3 Create temporary folders for code generation and generate code.

```
[TEMPDIR, CGDIR] = rtwdemodir();
slbuild(model);
```

4 To specify the model configurations for the Polyspace analysis, use a `pslinkoptions` object. Create this object by using the function `pslinkoptions`. To run a Bug Finder analysis, set `psOpt.VerificationMode` to `'BugFinder'`. To assert the range defined in a block on its input variables, specify `psOpts.InputRangeMode` as `'DesignMinMax'`.

```
% Create a Polyspace options object from the model.
psOpts = pslinkoptions(model);
```

```
% Set model configurtion for the Polyspace analysis.
psOpts.VerificationMode = 'BugFinder';
psOpts.InputRangeMode = 'DesignMinMax';
```

5 Start a Polyspace analysis. To specify model configuration for the Polyspace analysis, set the object `psOpt` as the optional second argument in `pslinkrunCrossRelease`.

```
% Run Polyspace analysis
[~,resultsFolder] = pslinkrunCrossRelease(model,psOpt);
bdclose(model);
```

The character vector `resultsFolder` contains the full path to the results folder.

Analyze Code Generated as Model Reference by Using an Earlier Simulink Release

To accelerate model simulations, invoke referenced Simulink models as simulation targets. To generate model reference simulation targets from a Simulink model, generate code from the `ModelOrSubsystem` by using `slbuild` with the build process specified as `ModelReferenceCoderTargetOnly`. Package the generated code by using `packNGo`. Then, analyze the generated code by running a cross-release Polyspace analysis. Before you run an analysis, you must integrate Polyspace with Simulink. See "Integrate Polyspace with MATLAB or Simulink Installation from Earlier Release".

1 Open the Simulink model `rtwdemo_roll` and configure the model for code generation.

```
% Load the model
model = 'rtwdemo_roll';
load_system(model);
% Configure the Solver
configSet = getActiveConfigSet(model);
set_param(configSet,'Solver','FixedStepDiscrete');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true)
```

2 Create temporary folders for code generation and generate code. Specify the option `ModelReferenceCoderTargetOnly`. See `slbuild`.

```
[TEMPDIR, CGDIR] = rtwdemodir();
slbuild(model,'ModelReferenceCoderTargetOnly');
```

The generated code is stored in the folder `slprj`

3 To package the code that is generated as a model reference, use the function `packNGo`. Locate the file `buildinfo.mat` in `<working folder>/slprj/ert/rtwdemo_counter` and use the full path to it as the input to `packNGo`. This command generates an archive containing the generated code and the object `buildinfo.mat`. See `packNGo`.

```
% Locate buildinfo and generate code archive
buildinfo = fullfile(pwd,'slprj','ert',model,'buildinfo.mat');
packNGo(buildinfo)
```

4 To specify the Polyspace analysis options, use a `pslinkoptions` object. Create this object by using the function `pslinkoptions`. To run a Bug Finder analysis, set `psOpt.VerificationMode` to `'BugFinder'`.

```
% Create a Polyspace options object from the model.
psOpts = pslinkoptions(model);
```

```
% Set properties that define the Polyspace analysis.
psOpts.VerificationMode = 'BugFinder';
psOpts.InputRangeMode = 'DesignMinMax';
```

5   Start a Polyspace analysis. To specify Polyspace analysis options, set the object `psOpt` as the optional second argument in `pslinkrunCrossRelease`. To analyze the code as a model reference, set the optional third argument `asModelRef` to true.

```
% Run Polyspace analysis
[~,resultsFolder] = pslinkrunCrossRelease(model,psOpt,true);
bdclose(model);
```

The character vector `resultsFolder` contains the full path to the results folder.

Specify Polyspace Analysis Options While Analyzing Code Generated by Using an Earlier Simulink Release

1   Open the Simulink model `rtwdemo_roll` and configure the model for code generation.

```
% Load the model
model = 'rtwdemo_roll';
load_system(model);
% Configure the Solver
configSet = getActiveConfigSet(model);
set_param(configSet,'Solver','FixedStepDiscrete');
set_param(configSet, 'SystemTargetFile', 'ert.tlc');
```

2   To enable packing the generated code in an archive, set the option `PackageGeneratedCodeAndArtifacts` to `true`.

```
set_param(configSet, 'PackageGeneratedCodeAndArtifacts', true)
```

3   Create temporary folders for code generation and generate code.

```
[TEMPDIR, CGDIR] = rtwdemodir();
slbuild(model);
```

4   To specify the model configuration for the Polyspace analysis, use a `pslinkoptions` object. Create this object by using the function `pslinkoptions`. To run a Bug Finder analysis, set `psOpt.VerificationMode` to `'BugFinder'`.

```
% Create a Polyspace options object from the model.
psOpts = pslinkoptions(model);

% Set properties that define the Polyspace analysis.
psOpts.VerificationMode = 'BugFinder';
psOpts.InputRangeMode = 'DesignMinMax';
```

5   To specify Polyspace analysis options, create an options file. An options file is a text file that contains Polyspace options in a flat list, one line for each option. For instance, to enable all Bug Finder checkers and AUTOSAR C++14 coding rules, create a text file named *OptionFile.txt*. In the text file, enter:

```
-checkers all
-autosarcpp14 all
```

Save the options file. You can save the preceding options in an options file named `Options.txt` in the default work folder.

See "Analysis Options in Polyspace Code Prover".

6 Start a Polyspace analysis.

- To specify the model configurations for the Polyspace analysis run, set the object `psOpt` as the optional second argument in `pslinkrunCrossRelease`.
- Because the code is generated as standalone code, set the third argument `asModelRef` to `false`.
- To specify the Polyspace analysis options, specify the relative path to the created options file as the fourth argument.

```
%  Locate options file
optionsPath = fullfile(userpath,'Options.txt');
% Run Polyspace analysis
[~,resultsFolder] = pslinkrunCrossRelease(model,psOpts,false,optionsPath);
bdclose(model);
```

The character vector `resultsFolder` contains the full path to the results folder.

Input Arguments

`ModelOrSubsystem` — Target of the analysis
bdroot (default) | model or system name

Target of the analysis specified as a character vector with the model or system in single quotes. The default value is the system returned by `bdroot`.

Example: `resultsDir = pslinkrunCrossRelease('rtwdemo_roll')` where `rtwdemo_roll` is the name of a model.

Data Types: `char`

`psOpt` — Options object
configuration options associated with `ModelOrSubsystem` (default) | configuration object created by using `pslinkoptions`

Specifies the model configuration for the Polyspace analysis by using a `pslinkoptions` object. You can modify certain analysis options by modifying `psOpt`, which is an object where individual fields represent model configuration options. For a full list of options that you can modify, see this table.

Polyspace Configuration Parameters Supported by `pslinkrunCrossRelease`

Property	Description	Value	Description
EnableAdditionalFileList	Enable an additional file list to be analyzed, specified as `true` or `false`. Use with the `AdditionalFileList` option.	`true`	Polyspace verifies additional files specified in the `AdditionalFileList` option.
		`false` (default)	Polyspace does not verify additional files.
AdditionalFileList	List of additional files to be analyzed, specified as a cell array of files. To add these files to the analysis, use the `EnableAdditionalFileList` option.	cell array	Polyspace considers the listed files for verification.
VerificationMode	Polyspace analysis mode specified as `'BugFinder'`, for a Bug Finder analysis, or `'CodeProver'`, for a Code Prover verification.	'BugFinder'	Polyspace runs a Bug Finder analysis.
		'CodeProver' (default)	Polyspace runs a Code Prover analysis.
InputRangeMode	Specifies the range of the input variables.	`'DesignMinMax'` (default)	Polyspace uses the input range defined in the workspace or a block.
		`'Fullrange'`	Polyspace uses full range inputs.
ParamRangeMode	Specifies the range of the constant parameters.	`'DesignMinmax'`	Polyspace uses the constant parameter range defined in the workspace or in a block.
		`'None'` (default)	Polyspace uses the value of parameters specified in the code.
OutputRangeMode	Specifies the output assertions.	`'DesignMinMax'`	Polyspace applies assertions to outputs by using a range defined in a block or the workspace.
		`'None'` (default)	Polyspace does not apply assertions to the output variables.

Property	Description	Value	Description
ModelRefVerifDepth	Specify the depth for analyzing the models that are referenced by the current model.	`'Current model Only'` (default)	Polyspace analyzes only the top model without analyzing the referenced models. Use this option when you refer to models that do not need to be analyzed, such as library models.
		`'1'|'2'|'3'`	Polyspace analyzes referenced models up to the specified depth in the reference hierarchy. To analyze the models that are referenced by the top model, specify the property `ModelRefVerifDepth` as `'1'`. To analyze models that are referenced by the first level of references, specify this property as `'2'`.
		`'All'`	Polyspace verifies all referenced models.
ModelRefByModelRefVerif	Specify whether you want to analyze all referenced models together or individually.	`true`	Polyspace analyzes the top model and the referenced models together. Use this option to check for integration or scaling issues.
		`false` (default)	Polyspace analyzes the top model and the referenced models individually.
AutoStubLUT	Specifies how lookup tables are used.	`true` (default)	Polyspace stubs the lookup tables and verifies the model without analyzing the lookup table code.
		`false`	Polyspace includes the lookup table code in the analysis.

Property	Description	Value	Description
CheckConfigBeforeAnalysis	Specifies the level of configuration checking done before the Polyspace analysis starts.	`'Off'`	Polyspace checks only for errors. The analysis stops if errors are found.
		`'OnWarn'` (default)	Polyspace stops the analysis when errors are found and displays a message when warnings are found.
		`'OnHalt'`	Polyspace stops the analysis when either errors or warnings are found.

Example: `pslinkrunCrossRelease('rtwdemo_roll', psOpt)`, where `psOpt` is an options object created by calling `pslinkoptions`

asModelRef — Indicator for model reference analysis
`false` (default) | `true`

Indicator for model reference analysis, specified as `true` or `false`.

- If `asModelRef` is `false` (default), the function generates options files so that Polyspace analyzes the generated code as standalone code.
- If `asModelRef` is `true`, the function generates options files so that Polyspace analyzes the generated code as model reference code.

Example: `pslinkrunCrossRelease('rtwdemo_roll', psOpt,true)`

Data Types: `logical`

OptionsFile — Relative path to a Polyspace options file
default Polyspace analysis options (default) | relative path to a custom options file

Relative path to a text file that contains a list of Polyspace analysis options. The options file must have each option in a separate line.

Example: `pslinkrunCrossRelease('rtwdemo_roll', psOpt,true,'OptionsFile.txt')`

Data Types: `char`

Output Arguments

polyspaceFolder — Folder containing Polyspace project and results
character vector

Name of the folder containing Polyspace projects and results, specified as a character vector. The default value of this variable is `results_$ModelName$`.

To change this value, see "Output folder" on page 12-16.

resultsFolder — Full path to subfolder containing Polyspace results
character vector

Full path to subfolder containing Polyspace results, specified as a character vector.

The folder `results_$ModelName$` contains your Polyspace project and a subfolder `$ModelName$` containing the analysis results. This variable provides the full path to the subfolder.

To change the parent folder `results_$ModelName$`, see "Output folder" on page 12-16.

See Also

`polyspacePackNGo` | `pslinkoptions` | `pslinkrunCrossRelease` | `slbuild`

Topics
"Run Polyspace on Code Generated by Using Previous Releases of Simulink"
"Run Polyspace Analysis on Generated Code by Using Packaged Options Files"
"Run Polyspace Analysis by Using MATLAB Scripts"
pslinkoptions Properties

Introduced in R2021a

polyspaceAutosar

Run Polyspace Code Prover on code implementation of AUTOSAR software components using MATLAB scripts

Syntax

```
[status, msg] = polyspaceAutosar('-create-project',projectFolder,'-arxml-
dir',arxmlFolder,'-sources-dir',codeFolder,options)
[status, msg] = polyspaceAutosar('-update-project',prevProjectFile,options)
[status, msg] = polyspaceAutosar('-update-and-clean-project',prevProjectFile,
options)

[status, msg, out] = polyspaceAutosar( ___ )
```

Description

`[status, msg] = polyspaceAutosar('-create-project',projectFolder,'-arxml-dir',arxmlFolder,'-sources-dir',codeFolder,options)` checks the code implementation of AUTOSAR software components for run-time errors and violation of data constraints in the corresponding AUTOSAR XML specifications. The analysis parses the AUTOSAR XML specifications (`.arxml` files) in `arxmlFolder`, modularizes the code implementation (`.c` files) in `codeFolder` based on the specifications, and runs Code Prover on each module for the checks. The Code Prover results are stored in `projectFolder`. After analysis, you can open the project `psar_project.psprj` from `projectFolder` in the Polyspace user interface or the file `psar_project.xhtml` in a web browser. You can view the results for each software component individually.

You can use additional options for troubleshooting, for instance, to perform only certain parts of the update and track down an issue or to provide extra header files or define macros.

`[status, msg] = polyspaceAutosar('-update-project',prevProjectFile,options)` updates the Code Prover analysis results based on changes in ARXML files or C source code since the last analysis. The update uses the XHTML file `prevProjectFile` from the previous analysis and reanalyzes only the code implementation of software components that changed since that analysis.

You can use additional options for troubleshooting.

`[status, msg] = polyspaceAutosar('-update-and-clean-project',prevProjectFile,options)` updates the Code Prover analysis results based on changes in ARXML files or C source code since the last analysis. The update reanalyzes only the code implementation of software components that changed since the previous analysis. A clean update also removes information about software components that are out of date. For instance, if you use an additional option to force the update for specific software components and other SWC-s have also changed, a clean update removes those other SWC-s from the Polyspace project.

You can use additional options for troubleshooting.

`[status, msg, out] = polyspaceAutosar( ___ )` runs a Code Prover analysis using the same options as before. The output, instead of appearing in the MATLAB Command Window, is redirected to a character vector `out`.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Examples

Run Code Prover on All Software Components

Suppose your ARXML files are in a folder `arxml` and your C source files in a folder `code` in the current folder.

Run Code Prover on all software components defined in your ARXML files. Store the results in a folder `polyspace-project` in a temporary folder.

The folder must not already exist. If previous results exist in that folder, you can update those results. An update only reanalyzes source files that changed since the previous run.

```
exampleDir = fullfile(polyspaceroot,'help',...
    'toolbox','codeprover','examples','polyspace_autosar');
arxmlDir = fullfile(exampleDir, 'arxml');
sourceDir = fullfile(exampleDir, 'code');

tempDir = tempdir;
projectDir = fullfile(tempDir, 'polyspace-project');
prevProjectFile = fullfile(projectDir, 'psar_project.xhtml');

% Update project file if it already exists, else create new project
projectDirAlreadyExists = isfolder(projectDir);

if projectDirAlreadyExists
    [status, msg] = polyspaceAutosar('-update-project', ...
        prevProjectFile);
else
    [status, msg] = polyspaceAutosar('-create-project', projectDir, ...
    '-arxml-dir', arxmlDir, ...
    '-sources-dir', sourceDir);
end
```

Input Arguments

`projectFolder` — Folder to store Polyspace results
character vector

Folder name, specified as a character vector. If the folder exists, it must be empty.

After analysis, the folder contains two project files `psar_project.psprj` and `psar_project.xhtml`.

- To see the results, open the project file `psar_project.psprj` in the Polyspace user interface or the file `psar_project.xhtml` in a web browser.
- For subsequent updates using MATLAB scripts, use the project file `psar_project.xhtml`.

See also "Review Polyspace Results on AUTOSAR Code".

Example: `'C:\Polyspace_Projects\proj_swc1'`

arxmlFolder — Folder containing ARXML files
character vector

Folder name, specified as a character vector.

UNC paths are not supported for the folder name.

Example: `'C:\arxml_swc1'`

codeFolder — Folder containing C files
character vector

Folder name, specified as a character vector.

UNC paths are not supported for the folder name.

Example: `'C:\code_swc1'`

prevProjectFile — Path to `psar_project.xhtml`
character vector

Path to the previously created project file `psar_project.xhtml`, specified as a character vector.

Example: `'C:\Polyspace_Projects\proj1\psar_project.xhtml'`

options — Options to control project creation
character vector

Options to control creation of a Polyspace project and subsequent analysis. You primarily use the options for troubleshooting, for instance, to perform only certain parts of the update and narrow down an issue or to provide extra header files or define macros.

Specify each option as a character vector, followed by the option value as a separate character vector. For instance, you can specify an options file `opts.txt` by using the syntax `polyspaceAutosar(...,'-options-file','opts.txt')`.

General options

Option	Value	Description
`'-verbose'`		Save additional information about the various phases of command execution (verbose mode). The file `psar_project.log` and other auxiliary files store this additional information.
		If an error occurs in command execution, the error message is stored in a separate file, irrespective of whether you enable verbose mode. Running in verbose mode only stores the various phases of execution. Use this information to see when an error was introduced.
`'-options-file'`	Options file name, for instance, `'opts.txt'`.	Use an options file to supplement or replace the command-line options. In the options file, specify each option on a separate line. Begin a line with # to indicate comments.
		An options file `opts.txt` can look like this:

		`# Store Polyspace results`
		`-create-project polyspace`
		`# ARXML Folder`
		`-arxml-dir arxml`
		`# SOURCE Folder`
		`-sources-dir code`

		If an option that is directly specified with the `polyspaceAutosar` function conflicts with an option in the options file, the directly specified option is used.
		You typically use an options file to store and reuse options that are common to multiple projects.

Options to control update of project

If you update a project, by default, the analysis results are updated for all AUTOSAR SWC behaviors with respect to any change in the ARXML files or C source code since the last analysis. Control the update by using these options.

Option	Value	Description

`'-autosar-behavior'`	Full qualified name of SWC behavior, for instance, `'pkg.component.bhv'`.	Check the implementation of software components whose internal behavior-s are specified. The default analysis considers all software components present in the ARXML specifications.
		
		To specify multiple software components, repeat the option. Alternatively, use regular expressions to specify a group of software components under the same package.
		
		For instance:
		
		• To specify the software component whose internal behavior has the fully qualified name `pkg.component.bhv`, use:
		
		`polyspaceAutosar(...,`
		`'-autosar-behavior',...`
		`'pkg.component.bhv')`
		
		• To specify the software components whose internal behavior-s have fully qualified names beginning with `pkg.component`, use:
		
		`polyspaceAutosar(...,`
		`'-autosar-behavior',...`
		`'pkg.component\..*')`
		
		The `\.` represents the package name separator `.` (dot) and the `.*` represents any number of characters.

`'-do-not-update-autosar-prove-environment'`		Do not read the ARXML specifications. Use ARXML specifications stored from the previous analysis. Use this option during project updates to compare the code against previous specifications. If you do not use this option, project updates read the entire ARXML specifications again.

Option	Value	Description
`'-do-not-update-extract-code'`		Do not read the C source code. Use source code stored from the previous analysis. Use this option during project updates to compare the previous source code against ARXML specifications. If you do not use this option, project updates consider all changes to the source code since the previous analysis.
`'-do-not-update-verification'`		Read the ARXML specifications and C code implementation only but do not run the Code Prover analysis. Use this option during project updates to investigate errors introduced in the ARXML specifications or compilation errors introduced in the source code. You can first fix these issues, and then run the Code Prover analysis.

Options to control parsing of ARXML specifications

Option	Value	Description
`'-autosar-datatype'`	Full qualified name of data type, for instance, `'pkg.datatypes.type'`	Import definition of AUTOSAR data types specified. The default analysis imports only data types specified in the internal behavior of software components that you verify.
		To specify multiple data types, repeat the option. Alternatively, use regular expressions to specify all data types under the same package.
		For instance:

		• To specify a data type that has the fully qualified name `pkg.datatypes.type`, use:
		
		`polyspaceAutosar(...,`
		`'-autosar-datatype',...`
		`'pkg.datatypes.type')`

		• To specify data types that have fully qualified names beginning with `pkg.datatypes`, use:
		
		`polyspaceAutosar(...,`
		`'-autosar-datatype',...`
		`'pkg.datatypes\..*')`
		
		The `\.` represents the package name separator `.` (dot) and the `.*` represents any number of characters.

		• To force import of all data types, use:
		
		`polyspaceAutosar(...,`
		`'-autosar-datatype',...`
		`'.*\..*')`

Option	Value	Description

`'-Eautosar-xmlReaderSameUuidForDifferentElements'`
 	
`'-Eno-autosar-xmlReaderSameUuidForDifferentElements'`
		If multiple elements in the ARXML specifications have the same universal-unique-identifier (UUID), use these options to toggle between a warning and an error.
		
		The default analysis stops with an error if this issue happens. To convert to a warning, use `'-Eno-autosar-xmlReaderSameUuidForDifferentElements'`. For conflicting UUIDs, the analysis stores the last element read and continues with a warning.
		
		The subsequent executions continue to use the warning mode. To revert back to an error, use `'-Eautosar-xmlReaderSameUuidForDifferentElements'`.

`'-Eautosar-xmlReaderTooManyUuids'`
 	
`'-Eno-autosar-xmlReaderTooManyUuids'`
		If the same element in the ARXML specifications has different universal-unique-identifiers (UUID), use these options to toggle between a warning and an error.
		
		The default analysis stops with an error if this issue happens. To convert to a warning, use `'-Eno-autosar-xmlReaderTooManyUuids'`. For conflicting UUIDs, the analysis stores the last element read and continues with a warning.
		
		The subsequent executions continue to use the warning mode. To revert back to an error, use `'-Eautosar-xmlReaderTooManyUuids'`.

Options to control reading of C source code

Option	Value	Description
`'-include'`	File with data type and macro definitions.	Define additional data types and macros that are not part of your ARXML specifications, but needed for analysis of the code implementation.
		Add the data type and macro definitions to a file. These definitions are appended to a header file `Rte_Type.h` that is used in the analysis. The file that you provide must itself not be named `Rte_Type.h`.
		You can provide the file with data type and macro definitions only during project creation. For subsequent updates, you can change the contents of this file but not provide a new file. Also, this file must not be in the same folder as the Polyspace project and results.
		If you additionally define macros or undefine them using the options `'-D'` or `'-U'`, for definitions that conflict with the ones in *USER_RTE_TYPE_H*, the `-D` or `-U` specifications prevail.
`'-I'`	Folder containing header files.	Specify folders containing header files. The analysis looks for `#include`-d files in this folder. The folder must be a subfolder of your source code folder.
		Repeat the option for multiple folders. The analysis looks for header files in these folders in the order in which you specify them.
		If you want to specify folders that are not in the source code folder, use the option:
		``` polyspaceAutosar(..., '-extra-project-options',... '-I INCLUDE_FOLDER') ```

Option	Value	Description
`'-D'`	Name of macro, for instance, `'_WIN32`.	Specify macros that the analysis must consider as defined. For instance, if you specify:the preprocessor conditional `#ifdef _WIN32` succeeds and the corresponding branch is executed.
`'-U'`	Name of macro, for instance, `'_WIN32`.	Specify macros that the analysis must consider as undefined. For instance, if you specify:the preprocessor conditional `#ifndef _WIN32` succeeds and the corresponding branch is executed.

Options to control Code Prover checks

Option	Value	Description
`'-extra-project-options'`	Space-separated list of options.	Specify additional options for the Code Prover analysis. The options that you specify do not apply to the ARXML parsing or code extraction, but only to the subsequent Code Prover analysis. Use this method to specify analysis options that are used in a non-AUTOSAR Code Prover analysis. See "Analysis Options in Polyspace Code Prover". For instance, you might want to specify a compiler and target architecture. By default, compilation of projects created from AUTOSAR specifications use the gnu4.7 compiler and i386 architecture. To specify a visual11.0 compiler with x86_64 architecture, enter this option:See also `Compiler (-compiler)` and `Target processor type (-target)`.

Option	Value	Description
`'-extra-options-file'`	File with Polyspace options.	Specify additional options for the Code Prover analysis in an options file. The options that you specify do not apply to the ARXML parsing or code extraction, but only to the subsequent Code Prover analysis. For instance, you can trace your build command to gather compiler options, macro definitions and paths to include folders, and provide this information in an options file for analysis of code implementation of AUTOSAR software components. 1 Trace your build command (for instance, `make`) with the `polyspaceConfigure` function and generate an options file for subsequent Code Prover analysis. Suppress inclusion of sources in the options file with the `-no-sources` option. `polyspaceConfigure ... -output-options-file ... options.txt ... -no-sources make` 2 Run Code Prover on AUTOSAR code with `polyspace-autosar`. Provide your ARXML folder, source folders and other options. In addition, provide the earlier generated options file with the `-extra-options-file` option.
`'-show-prove'`	Full qualified name of SWC behavior, for instance, `'pkg.component.bhv'`.	After analysis, open results for a specific software component whose internal behavior is specified.

Output Arguments

status — Value indicating completion
0 | 1-10 (error values)

Boolean flag indicating whether the analysis ran to completion. If the analysis is completed, the return value is 0, otherwise it is a nonzero value.

If you see a nonzero value, check the second output argument of `polyspaceAutosar` for error messages.

You can also look for error messages in the file `psar_project.xhtml` in your project folder. You can use this XHTML file to determine which software components were analyzed.

See "Troubleshoot Polyspace Analysis of AUTOSAR Code".

msg — Analysis log
structure

Analysis log, specified as a structure with these fields:

Criticity — Type of message
`'info'` | `'warning'` | `'error'`

Type of message, returned as one of three character vectors:

- `'info'`: Information such as current stage of analysis.
- `'warning'`: Warnings that do not stop analysis but can cause errors later.
- `'error'`: Errors that can stop the entire analysis or analysis of specific software components.

To check for errors, use this type information. For instance, to check for errors in the structure `msg`, use this code:

```
% Convert to table for logical indexing
msgTable = struct2table(msg);

% Check which messages have the type 'error'
errorMatches = (strcmp(msgTable.Criticity, 'error'));

% Read the error messages to another table
errorMessage = msgTable(errorMatches, :);
```

Message — Content of message
character vector

Content of message, returned as a character vector.

Example: `'Start Extract user-implementation for Behavior ''pkg.tst002.swc001.bhv001''...'`

out — Raw data in analysis log
character vector

Analysis log, returned as a character vector.

See Also

Topics
"Run Polyspace on AUTOSAR Code"
"Review Polyspace Results on AUTOSAR Code"
"Benefits of Polyspace for AUTOSAR"
"Using Polyspace in AUTOSAR Software Development"

Introduced in R2018b

polyspaceCodeProver

Run Polyspace Code Prover verification from MATLAB

Note For easier scripting, run Polyspace® analysis using a `polyspace.Project` object.

Syntax

```
polyspaceCodeProver
status = polyspaceCodeProver(projectFile)

status = polyspaceCodeProver(optsObject)
status = polyspaceCodeProver(projectFile, '-nodesktop')

status = polyspaceCodeProver(resultsFile)
status = polyspaceCodeProver('-results-dir',resultsFolder)

status = polyspaceCodeProver('-help')

status = polyspaceCodeProver('-sources',sourceFiles)
polyspaceCodeProver('-sources',sourceFiles,Name,Value)
```

Description

`polyspaceCodeProver` opens Polyspace Code Prover.

`status = polyspaceCodeProver(projectFile)` opens a Polyspace project file in Polyspace Code Prover.

`status = polyspaceCodeProver(optsObject)` runs a verification on the Polyspace options object in MATLAB.

`status = polyspaceCodeProver(projectFile, '-nodesktop')` runs a verification on the Polyspace project file in MATLAB. If you have multiple modules or configurations, Polyspace runs the active configuration and active module. To see which module and configuration are active, open the project in the Polyspace interface and look for the bold, selected module and configuration. To change which module or configuration is active, before closing the Polyspace interface, select the module and configuration you want to verify.

`status = polyspaceCodeProver(resultsFile)` opens a Polyspace results file in Polyspace Code Prover.

`status = polyspaceCodeProver('-results-dir',resultsFolder)` opens a Polyspace results file from `resultsFolder` in Polyspace Code Prover.

`status = polyspaceCodeProver('-help')` displays all options that can be supplied to the `polyspaceCodeProver` command to run a Polyspace Code Prover verification.

`status = polyspaceCodeProver('-sources',sourceFiles)` runs a Polyspace Code Prover verification on the source files specified in `sourceFiles`.

polyspaceCodeProver('-sources',sourceFiles,Name,Value) runs a Polyspace Code Prover verification on the source files with additional options specified by one or more Name,Value pair arguments.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Examples

Open Polyspace Projects from MATLAB

This example shows how to open a Polyspace project file with extension .psprj from MATLAB. In this example, open the project file Code_Prover_Example.psprj.

Assign the full project file path to a MATLAB variable prjFile.

```
prjFile = fullfile(polyspaceroot, 'polyspace', 'examples', 'cxx', ...
         'Code_Prover_Example', 'Code_Prover_Example.psprj');
```

Open the project.

```
polyspaceCodeProver(prjFile)
```

Open Polyspace Results from MATLAB

This example shows how to open a Polyspace results file from MATLAB. In this example, you open the results file from the folder *polyspaceroot*\polyspace\examples\cxx\Code_Prover_Example\Module_1\CP_Result.

Assign the full folder path to a MATLAB variable resFolder.

```
resFolder = fullfile(polyspaceroot, 'polyspace', 'examples', ...
       'cxx', 'Code_Prover_Example', 'Module_1', 'CP_Result');
```

Open the results.

```
polyspaceCodeProver('-results-dir',resFolder)
```

Run Polyspace Verification with Options Object

This example shows how to run a Polyspace verification in MATLAB using objects.

Create an options object and add the source file and include folder to the properties.

```
opts = polyspace.CodeProverOptions;
opts.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
opts.EnvironmentSettings.IncludeFolders = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources')};
opts.ResultsDir = fullfile(pwd,'results');
```

Run the verification and view the results.

```
polyspaceCodeProver(opts);
polyspaceCodeProver('-results-dir',opts.ResultsDir)
```

Run Polyspace Verification from MATLAB with DOS/UNIX Options

This example shows how to run a Polyspace verification in MATLAB using DOS/UNIX-style options.

Run the analysis and open the results.

```
sourceFiles = fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c');
includeFolders = fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources');
resultsDir = fullfile(pwd,'results');
polyspaceCodeProver('-sources',sourceFiles, ...
            '-I',includeFolders, ...
            '-results-dir',resultsDir,...
            '-main-generator');
polyspaceCodeProver('-results-dir',resultsDir);
```

Run Polyspace Verification with Coding Rules Checking

This example shows two different ways to customize a verification in MATLAB. You can customize as many additional options as you want by changing properties in an options object or by using Name-Value pairs. You specify checking of MISRA C 2012 coding rules, exclude headers from coding rule checking, and generate a main.

To create variables for source file path, include folder path, and results folder path that you can use for either analysis method.

```
sourceFileName = fullfile(polyspaceroot, 'polyspace','examples', 'cxx', ...
'Code_Prover_Example','sources','example.c');
includeFileName = fullfile(polyspaceroot, 'polyspace','examples', 'cxx', ...
 'Code_Prover_Example','sources','include.h');
resFolder1 = fullfile('Polyspace_Results_1');
resFolder2 = fullfile('Polyspace_Results_2');
```

Verify coding rules with an options object.

```
opts = polyspace.CodeProverOptions('C');
opts.Sources = {sourceFileName};
opts.EnvironmentSettings.IncludeFolders = {includeFileName};
opts.ResultsDir = resFolder1;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory';
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodeProverVerification.EnableMain = true;
opts.InputsStubbing.DoNotGenerateResultsFor = 'all-headers';
polyspaceCodeProver(opts);
polyspaceCodeProver('-results-dir',resFolder1);
```

Verify coding rules with DOS/UNIX options.

```
polyspaceCodeProver('-sources',sourceFileName,...
    '-I',includeFileName, ...
    '-results-dir',resFolder2,...
    '-misra3','mandatory',...
    '-do-not-generate-results-for','all-headers',...
```

```
        '-main-generator');
polyspaceCodeProver('-results-dir',resFolder2);
```

Input Arguments

optsObject — Polyspace options object name
object handle

Polyspace options object name, specified as the object handle.

To create an options object, use one of the Polyspace options classes: `polyspace.Options` or `polyspace.ModelLinkOptions`.

Example: `opts`

projectFile — Name of .psprj file
character vector

Name of project file with extension `.psprj`, specified as a character vector.

If the file is not in the current folder, `projectFile` must include a full or relative path. To identify the current folder, use `pwd`. To change the current folder, use `cd`.

Example: `'C:\Polyspace_Projects\myProject.psprj'`

resultsFile — Name of .pscp file
character vector

Name of results file with extension `.pscp`, specified as a character vector.

If the file is not in the current folder, `resultsFile` must include a full or relative path.

Example: `'myResults.pscp'`

resultsFolder — Name of result folder
character vector

Name of result folder, specified as a character vector. The folder must contain the results file with extension `.pscp`. If the results file resides in a subfolder of the specified folder, this command does not open the results file.

If the folder is not in the current folder, `resultsFolder` must include a full or relative path.

Example: `'C:\Polyspace\Results\'`

sourceFiles — Comma-separated names of .c or .cpp files
character vector

Comma-separated source file names with extension `.c` or `.cpp`, specified as a single character vector.

If the files are not in the current folder, `sourceFiles` must include a full or relative path.

Example: `'myFile.c'`, `'C:\mySources\myFile1.c,C:\mySources\myFile2.c'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'-target','i386','-compiler','gnu4.6'` specifies that the source code is intended for i386 processors and contains non-ANSI C syntax for the GCC 4.6 compiler.

For the full list of analysis options, see "Analysis Options in Polyspace Code Prover".

Output Arguments

`status` — Status indicating whether the Polyspace Code Prover verification completed successfully or not
`true` | `false`

If the Code Prover verification completes without error, `status` is `false`. Otherwise, it is `true`.

The verification might fail for multiple reasons, including:

*   You provided a source file, project file, or results file that does not exist.
*   You specified an invalid path.
*   One of your files did not compile.

See Also
`polyspace.CodeProverOptions` | `polyspace.ModelLinkCodeProverOptions`

Introduced in R2013b

polyspaceConfigure

Create Polyspace project from your build system at the MATLAB command line

Syntax

```
polyspaceConfigure buildCommand
```

```
polyspaceConfigure -option value buildCommand
```

Description

`polyspaceConfigure buildCommand` traces your build system and creates a Polyspace project with information gathered from your build system. You can run an analysis on a Polyspace project only in the user interface of the Polyspace desktop products.

`polyspaceConfigure -option value buildCommand` traces your build system and uses `-option value` to modify the default operation of `polyspaceConfigure`. Specify the modifiers before `buildCommand`, otherwise they are considered as options in the build command itself.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Examples

Create Polyspace Project from Makefile

This example shows how to create a Polyspace project if you use the command `make` *targetName buildOptions* to build your source code. The example creates a Polyspace project that can be opened only in the user interface of the Polyspace desktop products.

Create a Polyspace project specifying a unique project name. Use the `-B` or `-W` *makefileName* option with `make` so that the all prerequisite targets in the makefile are remade.

```
polyspaceConfigure  -prog myProject ...
         make -B targetName buildOptions
```

Open the Polyspace project in the **Project Browser**.

```
polyspaceCodeProver('myProject.psprj')
```

Create Projects That Have Different Source Files from Same Build Trace

This example shows how to create different Polyspace projects from the same trace of your build system. You can specify which source files to include for each project. The example creates a Polyspace project that can be opened only in the user interface of the Polyspace desktop products.

Trace your build system without creating a Polyspace project by specifying the option `-no-project`. To ensure that all the prerequisite targets in your makefile are remade, use the appropriate `make` build command option, for instance `-B`.

```
polyspaceConfigure -no-project make -B;
```

`polyspace-configure` stores the cache information and the build trace in default locations inside the current folder. To store the cache information and build trace in a different location, specify the options `-cache-path` and `-build-trace`.

Generate Polyspace projects by using the build trace information from the previous step. Specify a project name and use the `-include-sources` or `-exclude-sources` option to select which files to include for each project.

```
polyspaceConfigure -no-build -prog myProject ...
-include-sources "glob_pattern";
```

glob_pattern is a glob pattern that corresponds to folders or files you filter in or out of your project. To ensure the shell does not expand the glob patterns you pass to `polysapce-configure`, enclose them in double quotes.For more information on the supported syntax for glob patterns, see "polyspace-configure Source Files Selection Syntax".

If you specified the options `-build-trace` and `-cache-path` in the previous step, specify them again.

Delete the trace file and cache folder.

```
rmdir('polyspace_configure_cache', 's');
delete polyspace_configure_built_trace;
```

If you used the options `-build-trace` and `-cache-path`, use the paths and file names from those options.

Run Command-Line Polyspace Analysis from Makefile

This example shows how to run Polyspace analysis if you use a build command such as `make targetName buildOptions` to build your source code. In this example, you use `polyspaceConfigure` to trace your build system but do not create a Polyspace project. Instead you create an options file that you can use to run Polyspace analysis from the command-line.

Create a Polyspace options file specifying the `-output-options-file` command. Use the `-B` or `-W makefileName` option with `make` so that all prerequisite targets in the makefile are remade.

```
polyspaceConfigure -output-options-file ...
        myOptions make -B targetName buildOptions
```

Use the options file that you created to run a Polyspace analysis at the command line:

```
polyspaceCodeProver -options-file myOptions
```

Input Arguments

buildCommand — Command for building source code
build command

Build command specified exactly as you use to build your source code.

Example: `make -B`, `make -W makefileName`

`-option value` — Options for changing default operation of `polyspaceConfigure`

single option starting with `-`, followed by argument | multiple space-separated option-argument pairs

Basic Options

Option	Argument	Description
`-prog`	Project name	Project name that appears in the Polyspace user interface. The default is `polyspace`. If you do not use the option `-output-project`, the `-prog` argument also sets the project name. **Example:** `-prog myProject` creates a project that has the name `myProject` in the user interface. If you do not use the option `-output-project`, the project name is also `myProject.psrprj`.
`-author`	Author name	Name of project author. **Example:** `-author jsmith`
`-output-project`	Path	Project file name and location for saving project. The default is the file `polyspace.psprj` in the current folder. **Example:** `-output-project ../myProjects/project1` creates a project `project1.psprj` in the folder with the relative path `../myProjects/`.
`-output-options-file`	File name	Option to create a Polyspace analysis options file. Use this file for command-line analysis using one of these commands: • `polyspace-bug-finder` • `polyspace-code-prover` • `polyspace-bug-finder-server` • `polyspace-code-prover-server` • `polyspace-bug-finder-access`

Option	Argument	Description

`-allow-build-error`	None	Option to create a Polyspace project even if an error occurs in the build process.
		
		If an error occurs, the build trace log shows the following message:
		
		`polyspace-configure (polyspaceConfigure)`
		   `ERROR: build command`
		   `command_name fail [status=status_value]`
		
		command_name is the build command name that you use and *status_value* is the non-zero exit status or error level that indicates which error occurred in your build process.
		
		This option is ignored when you use `-compilation-database`.

`-allow-overwrite`	None	Option to overwrite a project with the same name, if it exists. By default, `polyspace-configure (polyspaceConfigure)` throws an error if a project with the same name already exists in the output folder. Use this option to overwrite the project.

`-no-console-output`
 	
`-silent` (default)
 	
`-verbose`
	None	Option to suppress or display additional messages from running `polyspace-configure (polyspaceConfigure)`.
		
		• `-no-console-output` – Suppress all outputs including errors and warnings.
		• `-silent` (default) – Show only errors and warnings.
		• `-verbose` – Show all messages.
		
		If you specify more than one of these options, the most verbose option is applied.
		
		These options are ignored if they are used in combination with `-easy-debug`.

`-help`	None	Option to display the full list of `polyspace-configure (polyspaceConfigure)` commands
`-debug`	None	Option to store debug information for use by MathWorks technical support. This option has been superseded by the option `-easy-debug`.

Option	Argument	Description
`-easy-debug`	Path	Option to store debug information for use by MathWorks technical support. After a `polyspace-configure` (`polyspaceConfigure`) run, the path provided contains a zipped file ending with `pscfg-output.zip`. If the run fails to create a complete Polyspace project or options file, send this zipped file to MathWorks Technical Support for further debugging. The zipped file does not contain source files traced in the build. See also "Errors in Project Creation from Build Systems".

Options to Create Multiple Modules

These options are not compatible with `-compilation-database`.

Option	Argument	Description
`-module`	None	Option to create a separate options file for each binary created in build system. You can only create separate options files for different binaries. You cannot create multiple modules in a Polyspace project (for running in the Polyspace user interface). Use this option only for build systems that use GNU and Visual C++ compilers. See also "Modularize Polyspace Analysis by Using Build Command".
`-output-options-path`	Path name	Location where generated options files are saved. Use this option together with the option `-module`. The options files are named after the binaries created in the build system.

Advanced Options

Option	Argument	Description
`-compilation-database`	Path and file name	Location and name of JSON compilation database (JSON CDB) file. You generate this file from your build system, for instance by using the flag `-DCMAKE_EXPORT_COMPILE_COMMANDS=1` with `cmake`. The file contains compiler calls for all the translation units in you projects. For more information, see JSON Compilation Database. `polyspace-configure` uses the content of this file to get information about your build system. The extracted compiler paths in the JSON CDB must be accessible from the path where you run `polyspace-configure`.

You do not specify a build command when you use this option.

The build systems and compilers support the generation of a JSON CDB:

- CMake
- Bazel
- Clang
- Ninja
- Qbs
- waf

This option is not compatible with `-no-project` and with the options to create multiple modules.

The cache control options, `-allow-build-error`, and `-no-build` are ignored when you use this option. |
| `-compiler-config` | Path and file name | Location and name of compiler configuration file.

The file must be in a specific format. For guidance, see the existing configuration files in *polyspaceroot*\polyspace\configure\compiler_configuration\. For information on the contents of the file, see "Compiler Not Supported for Project Creation from Build Systems".

Example: `-compiler-configuration myCompiler.xml` |

Option	Argument	Description
`-no-project`	None	Option to trace your build system without creating a Polyspace project and save the build trace information. Use this option to save your build trace information for a later run of `polyspace-configure` `(polyspaceConfigure)` with the `-no-build` option. This option is not compatible with `-compilation-database`.
`-no-build`	None	Option to create a Polyspace project using previously saved build trace information. To use this option, you must have the build trace information saved from an earlier run of `polyspace-configure` `(polyspaceConfigure)` with the `-no-project` option. If you use this option, you do not need to specify the `buildCommand` argument. This option is ignored when you use `-compilation-database`.

Option	Argument	Description

`-no-sources`	None	Option to create a Polyspace options file that does not contain the source file specifications.
		
		Use this option when you intend to specify the source files by other means. For instance, you can use this option when:
		
		• Running Polyspace on AUTOSAR-specific code.
		
		  You want to create an options file that traces your build command for the compiler options:
		
		  `-output-options-file options.txt -no-sources`
		
		  You later append this options file when extracting source file names from ARXML specifications and running the subsequent Code Prover analysis with `polyspace-autosar`
		
		  `-extra-options-file options.txt`
		
		  See also "Run Polyspace on AUTOSAR Code Using Build Command".
		• Running Polyspace in Eclipse.
		
		  Your source files are already specified in your Eclipse project. When running a Polyspace analysis, you want to specify an options file that has the compilation options only.

Option	Argument	Description

`-extra-project-options`	Options to use for subsequent Polyspace analysis. For instance, `"-stubbed-pointers-are-unsafe"`.	Options that are used for subsequent Polyspace analysis.
		
		Once a Polyspace project is created, you can change some of the default options in the project. Alternatively, you can pass these options when tracing your build command. The flag `-extra-project-options` allows you to pass additional options.
		
		Specify multiple options in a space separated list, for instance `"-allow-negative-operand-in-shift -stubbed-pointers-are-unsafe"`.
		
		Suppose you have to set the option `-stubbed-pointers-are-unsafe` for every Polyspace project created. Instead of opening each project and setting the option, you can use this flag when creating the Polyspace project:
		
		`-extra-project-options`
		`   "-stubbed-pointers-are-unsafe"`
		
		For the list of options available, see:
		
		• "Analysis Options in Polyspace Code Prover"
		
		If you are creating an options file instead of a Polyspace project from your build command, do not use this flag.

`-tmp-path`	Path	Location of folder where temporary files are stored.
`-build-trace`	Path and file name	Location and name of file where build information is stored. The default is `./polyspace_configure_build_trace.log`. **Example:** `-build-trace ../build_info/trace.log`

`-include-sources`
 	
`-exclude-sources`
	Glob pattern	Option to specify which source files `polyspace-configure` (`polyspaceConfigure`) includes in, or excludes from, the generated project. You can combine both options together.
		
		A source file is included if the file path matches the glob pattern that you pass to `-include-sources`.
		
		A source file is excluded if the file path matches the glob pattern that you pass to `-exclude-sources`.

Option	Argument	Description

`-print-included-sources`
 	
`-print-excluded-sources`
	None	Option to print the list of source files that `polyspace-configure (polyspaceConfigure)` includes in, or excludes from, the generated project. You can combine both options together. The output displays the full path of each file on a separate line.
		
		Use this option to troubleshoot the glob patterns that you pass to `-include-sources` or `-exclude-sources`. You can see which files match the pattern that you pass to `-include-sources` or `-exclude-sources`.

`-compiler-cache-path`	Folder path	Specify a folder path where `polyspace-configure` looks for or stores the compiler cache files. If the folder does not exist, `polyspace-configure` creates it.
		
		By default, Polyspace looks for and stores compiler caches under these folder paths:
		
		• **Windows**
		
		  `%appdata%\Mathworks\R20xxY\Polyspace`
		• **Linux**
		
		  `~/.matlab/R20xxY/Polyspace`
		• **Mac**
		
		  `~/Library/Application Support/ MathWorks/MATLAB/R20xxY/Polyspace`
		
		R20xxY is the release version of your Polyspace product, for instance R2020b.

`-no-compiler-cache`	None	Use this option if you do not want Polyspace to cache your compiler configuration information or to use an existing cache for your compiler configuration. By default, the first time you run `polyspace-configure` with a particular compiler configuration, Polyspace queries your compiler for the size of fundamental types, compiler macro definitions, and other compiler configuration information then caches this information. Polyspace reuses the cached information in subsequent runs of `polyspace-configure` for builds that use the same compiler configuration.
`-reset-compiler-cache-entry`	None	Use this option to query the compiler for the current configuration and to refresh the entry in the cache file that corresponds to this configuration. Other compiler configuration entries in the cache are not updated.

Option	Argument	Description
`-clear-compiler-cache`	None	Use this option to delete all compiler configurations stored in the cache file. If you also specify a build command or `-compilation-database`, `polyspace-configure` computes and caches the compiler configuration information of the current run, except if you specify `-no-project` or `-no-compiler-cache`.
`-import-macro-definitions`	none from-whitelist from-source-tokens	Use this option to specify how `polyspace-configure` queries the compiler for macro definitions. You can specify: • `none` — Polyspace does not query the compiler for macro definitions. You must provide the macro definitions manually. • `from-whitelist` — Polyspace uses an internal white list to query the compiler for macro definitions. Polyspace uses the white list by default when you use the option `-compilation-database`. • `from-source-tokens` (default, except if you use `-compilation-database`) — Polyspace uses every non-keyword token in your source code to query your compiler for macro definitions.
`-options-for-sources-delimiter`	A single character	Specify an option separator to use when multiple analysis options are associated with one source file using the `-options-for-sources` option. Typically, the `-options-for-sources` option uses a semicolon as separator. See also `-options-for-sources`.

Cache Control Options

These options are primarily useful for debugging. Use the options if `polyspace-configure (polyspaceConfigure)` fails and MathWorks Technical Support asks you to use the option and provide the cached files. Starting R2020a, the option `-easy-debug` provides an easier way to provide debug information. See "Contact Technical Support About Issues with Running Polyspace".

These options are ignored when you use `-compilation-database`.

Option	Argument	Description

`-no-cache`
 	
`-cache-sources` (default)
 	
`-cache-all-text`
 	
`-cache-all-files`
	None	Option to perform one of the following:
		
		• `-no-cache`: Not create a cache
		• `-cache-sources`: Cache text files temporarily created during build for later use by `polyspace-configure` (polyspaceConfigure).
		• `-cache-all-text`: Cache all text files including sources and headers.
		• `-cache-all-files`: Cache all files including binaries.
		
		Typically, you cache temporary files created by your build command to debug issues in tracing the command.

`-cache-path`	Path	Location of folder where cache information is stored. When tracing a Visual Studio build (`devenv.exe`), if you see the error: `path is too long` try using a shorter path for this option to work around the error. **Example:** `-cache-path ../cache`
`-keep-cache` `-no-keep-cache` (default)	None	Option to preserve or clean up cache information after `polyspace-configure` (polyspaceConfigure) completes execution. If `polyspace-configure` (polyspaceConfigure) fails, you can provide this cache information to technical support for debugging purposes.

See Also

Topics
"Modularize Polyspace Analysis by Using Build Command"
"Requirements for Project Creation from Build Systems"
"Compiler Not Supported for Project Creation from Build Systems"

Introduced in R2013b

polyspaceJobsManager

Manage Polyspace jobs on a MATLAB Parallel Server cluster

Syntax

```
polyspaceJobsManager('listjobs')
polyspaceJobsManager('cancel','-job',jobNumber)
polyspaceJobsManager('remove','-job',jobNumber)
polyspaceJobsManager('getlog','-job',jobNumber)
polyspaceJobsManager('wait','-job',jobNumber)
polyspaceJobsManager('promote','-job',jobNumber)
polyspaceJobsManager('demote','-job',jobNumber)

polyspaceJobsManager('download','-job',jobNumber)
polyspaceJobsManager('download','-job',jobNumber,'-results-folder',
resultsFolder)

polyspaceJobsManager( ___ ,'-scheduler',scheduler)
```

Description

`polyspaceJobsManager('listjobs')` lists all Polyspace jobs in your cluster.

`polyspaceJobsManager('cancel','-job',jobNumber)` cancels the specified job. The job appears in your queue as cancelled.

`polyspaceJobsManager('remove','-job',jobNumber)` removes the specified job from your cluster.

`polyspaceJobsManager('getlog','-job',jobNumber)` displays the log for the specified job.

`polyspaceJobsManager('wait','-job',jobNumber)` pauses until the specified job is done.

`polyspaceJobsManager('promote','-job',jobNumber)` moves the specified job up in the MATLAB job scheduler queue.

`polyspaceJobsManager('demote','-job',jobNumber)` moves the specified job down in the MATLAB job scheduler queue.

`polyspaceJobsManager('download','-job',jobNumber)` downloads the results from the specified job. The results are downloaded to the folder you specified when starting analysis, using the `-results-dir` on page 2-44 option.

`polyspaceJobsManager('download','-job',jobNumber,'-results-folder',
resultsFolder)` downloads the results from the specified job to `resultsFolder`.

`polyspaceJobsManager( ___ ,'-scheduler',scheduler)` performs the specified action on the job scheduler specified. If you do not specify a server with any of the previous syntaxes, Polyspace uses the server stored in your Polyspace preferences.

Examples

Manipulate Two Jobs in the Cluster

In this example, use a MATLAB Job Scheduler scheduler to run Polyspace remotely and monitor your jobs through the queue.

Before performing this example, set up a MATLAB Job Scheduler and Polyspace Metrics. This example uses the *myMJS@myCompany.com* scheduler. When you perform this example, replace this scheduler with your own cluster name.

Set up your source files.

```
tempDir = fullfile(tempdir, 'psdemo', 'src');
mkdir(tempDir);
demo = fullfile(polyspaceroot,'polyspace','examples','cxx',...
'Code_Prover_Example','sources');
copyfile(demo,tempDir);
```

Submit two jobs to your scheduler.

If your jobs have not started running, promote the second job to run before the first job.

```
polyspaceJobsManager('promote','-job','20','-scheduler',...
    'myMJS@myCompany.com')
```

Job 20 starts running before job 19.

Cancel job 19.

```
polyspaceJobsManager('cancel','-job','19','-scheduler',...
    'myMJS@myCompany.com')
polyspaceJobsManager('listjobs','-scheduler','myMJS@myCompany.com')
```

Remove job 19.

```
polyspaceJobsManager('remove','-job','19','-scheduler',...
    'myMJS@myCompany.com')
polyspaceJobsManager('listjobs','-scheduler','myMJS@myCompany.com')
```

Get the log for job 20.

```
polyspaceJobsManager('getlog','-job','20','-scheduler',...
    'myMJS@myCompany.com')
```

Download the information from job 20.

```
resFolder3 = fullfile(tempDir, 'res3');
polyspaceJobsManager('download','-job','20','-results-folder', ...
    resFolder3,'-scheduler','myCluster')
```

Input Arguments

jobNumber — Queued job number
character vector of job number

Number of the queued job that you want to manage, specified as a character vector in single quotes.

Example: `'-job','10'`

resultsFolder — Path to results folder
character vector

Path to results folder specified as a character vector in single quotes. This folder stores the downloaded results files.

Example: `'-results-folder','C:\psdemo\myresults'`

scheduler — job scheduler
head node of your cluster | job scheduler name | cluster profile

Job scheduler for remote verifications specified as one of the following:

* Name of the computer that hosts the head node of your MATLAB Parallel Server cluster (*NodeHost*).
* Name of the MATLAB Job Scheduler on the head node host (*MJSName*@*NodeHost*).
* Name of a MATLAB cluster profile (*ClusterProfile*).

Example: `'-scheduler','myscheduler@mycompany.com'`

See Also
polyspaceCodeProver

Topics
"Discover Clusters and Use Cluster Profiles" (Parallel Computing Toolbox)
"Send Polyspace Analysis from Desktop to Remote Servers Using Scripts"

Introduced in R2013b

polyspaceroot

Get Polyspace installation folder

Syntax

```
polyspaceroot
```

Description

`polyspaceroot` returns the Polyspace installation folder.

Starting in R2019a, to run MATLAB scripts for Polyspace analysis, you install MATLAB and Polyspace in separate folders and link between them. After installation and linking, to access files in the Polyspace installation folder from MATLAB, use this function. See also "Integrate Polyspace with MATLAB and Simulink".

Examples

Get Polyspace Installation Folder

To determine the Polyspace installation folder, use the `polyspaceroot` function.

```
polyspaceroot
```

```
C:\Program Files\Polyspace\R2019a
```

With the products, Polyspace Bug Finder Server or Polyspace Code Prover Server, the default installation folder in Windows is:

```
C:\Program Files\Polyspace Server\R2019a
```

Run Polyspace on Sample Files in Polyspace Installation Folder

To access sample files in the Polyspace installation folder, use the `polyspaceroot` function to get the root of the installation folder. Append subfolders to the root folder path with the `fullfile` function.

Run Bug Finder on the file `numerical.c` in the subfolder `polyspace\examples\cxx\Bug_Finder_Example\sources` of the Polyspace installation folder.

```
proj = polyspace.Project

% Specify sources and includes
sourceFile = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c');
includeFolder = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources');

% Configure analysis
proj.Configuration.Sources = {sourceFile};
```

```
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.EnvironmentSettings.IncludeFolders = {includeFolder};
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = proj.run('bugFinder');
```

See Also

polyspace.Project

Topics

"Run Polyspace Analysis by Using MATLAB Scripts"

Introduced in R2019a

polyspace_report

Generate reports from Polyspace analysis results

Syntax

```
polyspace_report('-template', template, '-results-dir', resultsFolder,
options)
polyspace_report('-generate-results-list-file', '-results-dir',
resultsFolder, options)
polyspace_report('-generate-variable-access-file', '-results-dir',
resultsFolder, options)
```

Description

`polyspace_report('-template', template, '-results-dir', resultsFolder, options)` generates a report using a predefined template specified by `template`. By default, the report is named after the results file in the folder `resultsFolder` and saved in the `Polyspace-Doc` subfolder. You can change the default behavior using additional options.

`polyspace_report('-generate-results-list-file', '-results-dir', resultsFolder, options)` exports the list of Polyspace results to a tab-delimited text file.

`polyspace_report('-generate-variable-access-file', '-results-dir', resultsFolder, options)` exports the list of global variables to a tab-delimited text file.

Note

- Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".
- You need MATLAB Report Generator™ to use this function.

Examples

Generate PDF Report from Results

Generate a PDF report from sample Polyspace Code Prover results.

```
template = fullfile(polyspaceroot,'toolbox','polyspace','psrptgen','templates',...
    'Developer.rpt');
resPath = fullfile(polyspaceroot,'polyspace','examples','cxx','Code_Prover_Example',...
    'Module_1','CP_Result');
polyspace_report('-template', template, '-results-dir', resPath, '-format', 'PDF');
```

Input Arguments

`template` — Path to report template file
character vector

Path to report template file, specified as a character vector. To generate multiple reports, specify a comma-separated list of report template paths in the character vector (do not put a space after the commas). The templates are available in *polyspaceroot*\toolbox\polyspace\psrptgen \templates\ as .rpt files. Here, *polyspaceroot* is the Polyspace installation folder. For more information on the available templates, see Bug Finder and Code Prover report (-report-template).

Example: fullfile(polyspaceroot,'toolbox','polyspace','psrptgen','templates', 'Developer.rpt');

resultsFolder — Folder containing analysis results
character vector

Folder containing analysis results, specified as a character vector. The folder must contain a .psbf file containing Polyspace Bug Finder results or a .pscp file containing Polyspace Code Prover results.

To generate reports for multiple analyses, specify a comma-separated list of folder paths (do not put a space after the commas).

Example: 'C:\Polyspace_Workspace\My_project\Module_1\results'

options — Options for generating report
character vector

Options to control report generation, for instance, output format and output name.

Specify each option as a character vector, followed by the option value as a separate character vector. For instance, you can specify the PDF format by using the syntax polyspace_report(..., '-format','PDF').

Option	Value	Description
'-format'	'PDF', 'HTML' or 'WORD'	File format of the report that you generate. By default, the command generates a Word document. To generate reports in multiple formats, specify a comma-separated list of formats. (Do not put a space after the commas). For instance, polyspace_report(..., '-format', 'PDF,HTML'). This option is not compatible with -generate-variable-access-file and -generate-results-list-file.
'-set-language-english'		Generate the report in English. Use this option if your display option is set to another language.

Option	Value	Description
`'-output-name'`	Report name, for instance, `PolyspaceReport`.	Name of the generated report or folder name if you generate multiple reports. The full path to the report is created by appending the name to the current working folder. To store the reports on a different path, specify the full path as value for this option.

See Also

Introduced in R2013b

polyspace.Project

Run Polyspace analysis on C and C++ code and read results

Description

Run a Polyspace analysis on C and C++ source files by using this MATLAB object. To specify source files and customize analysis options, use the `Configuration` property. To run the analysis, use the `run` method. To read results after analysis, use the `Results` property.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Creation

`proj = polyspace.Project` creates an object that you can use to configure and run a Polyspace analysis, and then read the analysis results.

Properties

Configuration — Analysis options
`polyspace.Options` object

Options for running Polyspace analysis, implemented as a `polyspace.Options` object. The object has properties corresponding to the analysis options. For more information on those properties, see polyspace.Project.Configuration properties.

You can retain the default options or change them in one of these ways:

- Set the source code language to `'C'`, `'CPP'`, or `'C-CPP'` (default). Some analysis options might not be available depending on the language setting of the object.

  ```
  proj=polyspace.Project;
  proj.Configuration=polyspace.Options('C');
  ```

- Modify the properties directly.

  ```
  proj = polyspace.Project;
  proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
  ```

- Obtain the options from another `polyspace.Project` object.

  ```
  proj1 = polyspace.Project;
  proj1.Configuration.TargetCompiler.Compiler = 'gnu4.9';

  proj2 = proj1;
  ```

  To use common analysis options across multiple projects, follow this approach. For instance, you want to reuse all options and change only the source files.

- Obtain the options from a project created in the user interface of the Polyspace desktop products (`.psprj` file).

```
proj = polyspace.Project;
projectLocation = fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'Bug_Finder_Example.psprj')
proj.Configuration = polyspace.loadProject(projectLocation);
```

To determine the optimal set of options, set your options in the user interface and then import them to a `polyspace.Project` object. In the user interface, you can access help from features such as the Compilation Assistant and get tooltip help on options.

- Obtain the options from a Simulink model (applies only to Polyspace desktop products). Before obtaining the options, generate code from the model.

```
modelName = 'rtwdemo_roll';
load_system(modelName);

% Set parameters for Embedded Coder target
set_param(modelName, 'SystemTargetFile', 'ert.tlc');
set_param(modelName,'Solver','FixedStepDiscrete');
set_param(modelName,'SupportContinuousTime','on');
set_param(modelName,'LaunchReport','off');
set_param(modelName,'InitFltsAndDblsToZero','on');

if exist(fullfile(pwd,'rtwdemo_roll_ert_rtw'), 'dir') == 0
    slbuild(modelName);
end

% Obtain configuration from model
proj = polyspace.Project;
proj.Configuration = polyspace.ModelLinkOptions(modelName);
```

Use the options to analyze the code generated from the model.

Results — Analysis results
polyspace.BugFinderResults or polyspace.CodeProverResults object

Results of Polyspace analysis. When you create a `polyspace.Project` object, this property is initially empty. The property is populated only after you execute the `run` method of the object. Depending on the argument to the `run` method, `'bugFinder'` or `'codeProver'`, the property is implemented as a `polyspace.BugFinderResults` object or`polyspace.CodeProverResults` object.

To read the results, use these methods of the `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object:

- `getSummary`: Obtain a summarized format of the results into a MATLAB table.

```
proj = polyspace.Project;
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.ResultsDir = fullfile(pwd,'results');

run(proj, 'bugFinder');

resObj = proj.Results;
resTable = getSummary(resObj, 'defects');
```

For more information, see `getSummary`.

- **getResults**: Obtain the full results or a more readable format into a MATLAB table.

```
proj = polyspace.Project;
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.ResultsDir = fullfile(pwd,'results');

run(proj, 'bugFinder');

resObj = proj.Results;
resTable = getResults(resObj, 'readable');
```

For more information, see `getResults`.

Object Functions

run   Run a Polyspace analysis

Examples

Check for Bugs

Run a Polyspace Bug Finder analysis on the example file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
resObj = proj.Results;
bfSummary = getSummary(resObj, 'defects');
```

Prove Absence of Run-Time Errors

Run a Polyspace Code Prover analysis on the example file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a `main` function must be generated, if the function does not exist in the source code.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
```

5-131

```
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;


% Run analysis
cpStatus = run(proj, 'codeProver');

% Read results
resObj = proj.Results;
cpSummary = getSummary(resObj, 'runtime');
```

Check for Bugs and MISRA C:2012 Violations

Run a Polyspace Bug Finder analysis on the example file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Enable checking of MISRA C:2012 rules. Check for the mandatory rules only.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory';

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
resObj = proj.Results;
defectsSummary = getSummary(resObj, 'defects');
misraSummary = getSummary(resObj, 'misraC2012');
```

See Also

Topics
"Run Polyspace Analysis by Using MATLAB Scripts"
"Generate MATLAB Scripts from Polyspace User Interface"
"Troubleshoot Polyspace Analysis from MATLAB"

Introduced in R2017b

polyspace.Options class

Package: `polyspace`

Create object for running Polyspace analysis on handwritten code

---

Note For easier scripting, specify the Polyspace® analysis options using the `Configuration` property of a `polyspace.Project` object. Do not create a `polyspace.Options` object directly.

Description

Run a Polyspace analysis from MATLAB by using an options object. To specify source files and customize analysis options, change the object properties.

To analyze model-generated code (using the Polyspace desktop products), use `polyspace.ModelLinkOptions` instead.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Construction

`opts = polyspace.Options` creates an object whose properties correspond to options for running a Polyspace analysis.

`proj = polyspace.Project` creates a `polyspace.Project` object. The object has a property `Configuration`, which is a `polyspace.Options` object.

`opts = polyspace.Options(lang)` creates a Polyspace options object with options that are applicable to the language `lang`.

`opts = polyspace.loadProject(projectFile)` creates a Polyspace options object from an existing Polyspace project `projectFile`. You set the options in your project in the Polyspace user interface and create the options object from that project for programmatically running the analysis.

Input Arguments

lang — Language of analysis
`'C-CPP'` (default) | `'C'` | `'CPP'`

The language of the analysis specified as `'C-CPP'`, `'C'`, or `'CPP'`. This argument determines the object properties.

Data Types: `char`

projectFile — Name of .psprj file
character vector

Name of Polyspace project file with extension `.psprj`, specified as a character vector.

If the file is not in the current folder, `projectFile` must include a full or relative path. To identify the current folder, use `pwd`. To change the current folder, use `cd`.

Note You cannot use the `loadProject` method on a project file that is created from a build command by using `polyspace-configure`.

Example: `'C:\projects\myProject.psprj'`

Properties

The object properties correspond to the analysis options for Polyspace projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS/UNIX command-line name. For syntax details, see polyspace.Project.Configuration properties.

Methods

copyTo	Copy common settings between Polyspace options objects
generateProject	Generate psprj project from options object
toScript	Add Polyspace options object definition to a script

Examples

Customize and Run Analysis

Create a Polyspace analysis options object and customize the properties. Then, run an analysis.

Create object and customize properties. In case you do not have write access to your current folder, a temporary folder is being used for storing analysis results.

```
sources = fullfile(polyspaceroot, 'polyspace','examples','cxx','Bug_Finder_Example',...
    'sources','numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
opts.ResultsDir = tempname;
```

Run a Bug Finder analysis. To run a Code Prover analysis, use `polyspaceCodeProver` instead of `polyspaceBugFinder`.

```
results = polyspaceBugFinder(opts);
```

With the Polyspace Server products, you can use the functions `polyspaceBugFinderServer` or `polyspaceCodeProverServer`.

Open the results in the Polyspace user interface of the desktop products.

```
polyspaceBugFinder('-results-dir',opts.ResultsDir);
```

Run Polyspace by Generating a Project File

Create a Polyspace analysis options object and customize the properties. Then, run a Bug Finder analysis.

Create object and customize properties.

```
sources=fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example',...
    'sources','numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
opts.ResultsDir = tempname;
```

Generate a Polyspace project, name it using the `Prog` property, and open the project in the Polyspace interface.

```
psprj = opts.generateProject(opts.Prog);
polyspaceBugFinder(psprj);
```

You can also analyze the project from the command line. Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceBugFinder(psprj, '-nodesktop');
polyspaceBugFinder('-results-dir',opts.ResultsDir);
```

Alternatives

If you are analyzing code generated from a model, use `polyspace.ModelLinkOptions` instead.

See Also
`polyspace.ModelLinkOptions` | `polyspace.Project` | `polyspaceCodeProver`

Topics
"Run Polyspace Analysis by Using MATLAB Scripts"
"Generate MATLAB Scripts from Polyspace User Interface"

Introduced in R2017a

polyspace.ModelLinkOptions class

Package: `polyspace`

Create a project configuration object for running Polyspace analysis on generated code

Description

Run a Polyspace analysis from MATLAB by using a project configuration object. To specify source files and customize analysis options, change the object properties.

This class is intended for model-generated code. If you are analyzing handwritten code, use `polyspace.Options` instead.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Construction

`psprjConfig = polyspace.ModelLinkOptions` creates a project configuration object that is configured for running a Polyspace analysis on generated code.

`psprjConfig = polyspace.ModelLinkOptions(lang)` creates a project configuration object that is configured to run analysis on code generated in the language `lang`.

`psprjConfig = polyspace.ModelLinkOptions(model)` creates a project configuration object that is configured by using model specific information from the Simulink model `model`. Prior to extracting options from `model`, you must load the model and generate code from it.

`psprjConfig = polyspace.ModelLinkOptions(model, psOpt)` creates a model-specific project configuration object that is configured by using the Polyspace analysis options specified in `psOpt`.

`psprjConfig = polyspace.ModelLinkOptions(model, psOpt, asModelRef)` creates a project configuration object that uses `asModelRef` to specify which type of generated code to analyze—standalone code or model reference code.

Input Arguments

`lang` — Language of analysis
`'C-CPP'` (default) | `'C'` | `'CPP'`

The language of the analysis specified as `'C-CPP'`, `'C'`, or `'CPP'`. This argument determines the object properties.

`model` — Model or subsystem name
character vector

Name or path to model or subsystem, specified as a character vector.

Prior to extracting options from the model, you must:

1   Load the model. Use `load_system` or `open_system`.

2   Generate code from the model. Use `slbuild` or `slbuild`.

Example: `'psdemo_model_link_sl'`

psOpt — Polyspace analysis options object
`pslinkoptions` object

An object containing the options that you use for the Polyspace analysis. You create this by calling the function `pslinkoptions`. You can customize the options object by changing the properties of the `psOpt` object.

Example: `psOpt = pslinkoptions(model)` where `model` is the name of a Simulink model.

asModelRef — Indicator for model reference analysis
`false` (default) | `true`

Indicator for model reference analysis, specified as true or false.

*   To analyze generated code used or called elsewhere, set the flag `asModelRef` to `true`. This option is equivalent to choosing **Analyze Code from > Code Generated as Model Reference** on the Polyspace tab in the Simulink toolstrip.

*   To analyze code that is generated to be used as stand-alone code, set the flag `asModelRef` to `false`. This option is equivalent to choosing **Analyze Code from > Code Generated as Top model** on the Polyspace tab in the Simulink toolstrip.

Data Types: `logical`

Properties

The object properties correspond to the configuration options for Polyspace projects. The properties are organized in the same categories as the Polyspace interface. The property names are a shortened version of the DOS command-line name. For syntax details, see polyspace.ModelLinkOptions.

Methods

copyTo	Copy common settings between Polyspace options objects
generateProject	Generate psprj project from options object
toScript	Add Polyspace options object definition to a script

Examples

Script Analysis of Model Generated Code

This example shows how to customize and run an analysis on code generated from a model.

Generate code from the model `sldemo_bounce`. Before code generation, set a system target file appropriate for code analysis. See also "Recommended Model Configuration Parameters for Polyspace Analysis".

```
modelName = 'rtwdemo_roll';
load_system(modelName);
```

```matlab
% Set parameters for Embedded Coder target
set_param(modelName, 'SystemTargetFile', 'ert.tlc');
set_param(modelName,'Solver','FixedStepDiscrete');
set_param(modelName,'SupportContinuousTime','on');
set_param(modelName,'LaunchReport','off');
set_param(modelName,'InitFltsAndDblsToZero','on');

if exist(fullfile(pwd,'rtwdemo_roll_ert_rtw'), 'dir') == 0
    slbuild(modelName);
end
```

Associate a `polyspace.ModelLinkOptions` object with the model. A subset of the object properties are set from the configuration parameters associated with the model. The other properties take their default values. For details on the configuration parameters, see "Polyspace Analysis in Simulink".

```matlab
psprjCfg = polyspace.ModelLinkOptions(modelName);
```

Change the property values if needed. For instance, you can specify that the analysis must check for all MISRA C: 2012 violations and generate a PDF report of the results. You can also specify a folder for the analysis results.

```matlab
psprjCfg.CodingRulesCodeMetrics.EnableMisraC3 = true;
psprjCfg.CodingRulesCodeMetrics.MisraC3Subset = 'all';
psprjCfg.MergedReporting.EnableReportGeneration = true;
psprjCfg.MergedReporting.ReportOutputFormat = 'PDF';
psprjCfg.ResultsDir = 'newResfolder';
```

Create a `polyspace.Project` object. Associate the `Configuration` property of this object to the options that you previously specified.

```matlab
proj = polyspace.Project;
proj.Configuration = psprjCfg;
```

Run analysis and open results.

```matlab
cpStatus = proj.run('codeProver');
proj.Results.getResults('readable');
```

Analyze Code Generated as Model Reference

This example shows how to analyze generated code used as a callable entity in another model or code.

Load the Simulink model `rtwdemo_roll` and configure it for a Polyspace analysis. For details, see "Recommended Model Configuration Parameters for Polyspace Analysis" for details.

```matlab
%  Specify model name
model = 'rtwdemo_roll';
% Load the model
load_system(model);
% Configure the model for generating code
set_param(model, 'SystemTargetFile', 'ert.tlc');
set_param(model,'MatFileLogging','off');
set_param(model,'GenerateComments','on');
```

```
set_param(model,'Solver','FixedStepDiscrete');
set_param(model,'LaunchReport','off');
```

To generate code as a model reference from the Simulink model, use `slbuild`. Set the `buildspec` parameter to `'ModelReferenceCoderTargetOnly'`.

```
if exist(fullfile(pwd,'slprj','ert'), 'dir') == 0
    slbuild(model,'ModelReferenceCoderTargetOnly');
end
```

To configure the Polyspace analysis of the generated code, create an options object `psOpt` by using the function `pslinkoptions`. Change the properties of the object as needed. For instance, to run a **Code Prover** analysis, set the `Verificationmode` to `'CodeProver'`.

```
psOpt = pslinkoptions(model);
psOpt.VerificationMode = 'CodeProver';
```

To run a Polyspace analysis, create and configure a Polyspace project configuration object.

- To create the Polyspace project configuration object, use the function `polyspace.ModelLinkOptions`.
- To associate the Polyspace analysis options with the project configuration, set the object `psOpt` as the second argument in `polyspace.ModelLinkOptions()`.
- To specify that the generated code must be analyzed as a model reference, specify the third argument as `'true'`.

For instance:

```
psprjCfg = polyspace.ModelLinkOptions(model, psOpt,true);
```

To configure the Polyspace project, change the properties of the `psprjCfg` object. For instance, to enable checkers for the mandatory MISRA C: 2012 rules and to generate a PDF report of the results, use:

```
psprjCfg = polyspace.ModelLinkOptions(model, psOpt,asModelRef);
psprjCfg.CodingRulesCodeMetrics.EnableMisraC3 = true;
psprjCfg.CodingRulesCodeMetrics.MisraC3Subset = 'mandatory-required';
psprjCfg.MergedReporting.EnableReportGeneration = true;
psprjCfg.MergedReporting.ReportOutputFormat = 'PDF';
```

For convenience, you can specify a separate result folder.

```
psprjCfg.ResultsDir = 'newResfolder';
```

Create a Polyspace project by using `polyspace.Project` and associate the project configuration with it.

```
proj = polyspace.Project;
proj.Configuration = psprjCfg;
```

Run the Polyspace analysis by using the `run` function of the object `proj`.

```
cpStatus = proj.run('codeProver');
```

Because you enabled PDF report generation, the result of the Polyspace analysis is reported in a PDF file, which can be found in *newResfolder/Polyspace-Doc*. To view the results in a MATLAB table, use:

```
result = proj.Results.getResults('readable');
```

Alternatives

If you are analyzing handwritten code, use a `polyspace.Project` object directly. Alternatively, use a `polyspace.Options` object.

See Also

`polyspace.Options` | `polyspace.Project` | `polyspaceCodeProver` | `pslinkrun`

Topics
"Run Polyspace Analysis by Using MATLAB Scripts"
polyspace.ModelLinkOptions Properties

Introduced in R2017a

polyspace.CodeProverOptions class

Package: `polyspace`

Create Polyspace Code Prover object for handwritten code

Note This class is deprecated and will be removed in a future release. Use `polyspace.Options` instead.

Description

Customize a Polyspace Code Prover verification from MATLAB by creating a Code Prover options object. To specify source files and customize analysis options, change the object properties.

If you are verifying model-generated code, use `polyspace.ModelLinkCodeProverOptions` instead.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Construction

`opts = polyspace.CodeProverOptions` creates a Code Prover options object with options for C code verification.

`opts = polyspace.CodeProverOptions(lang)` creates a Code Prover options object with options applicable for the language `lang`.

Input Arguments

`lang` — Language of analysis
`'C'` (default) | `'CPP'`

Language of verification specified as either `'C'` or `'CPP'`. This argument determines which properties the object has.

Properties

The object properties match the analysis options found in the Polyspace interface. For syntax details, see polyspace.Options.

Methods

copyTo	Copy common settings between Polyspace options objects
generateProject	Generate psprj project from options object
toScript	Add Polyspace options object definition to a script

Examples

Use Code Prover Object to Customize and Run Verification

Create a Code Prover options object and customize the properties. Then, run a verification.

Create object and customize properties.

```
sources = fullfile(polyspaceroot, 'polyspace','examples','cxx','Code_Prover_Example', ...
'sources','single_file_analysis.c');
includes = fullfile(polyspaceroot, 'polyspace','examples','cxx','Code_Prover_Example', ...
'sources');
optsCP = polyspace.CodeProverOptions();
optsCP.Prog = 'MyProject';
optsCP.Sources = {sources};
optsCP.EnvironmentSettings.IncludeFolders = {includes};
optsCP.TargetCompiler.Compiler = 'gnu4.7';
optsCP.ResultsDir = tempname;
```

Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceCodeProver(optsCP);
polyspaceCodeProver('-results-dir',optsCP.ResultsDir);
```

Run Polyspace by Generating a Project File

Create a Code Prover analysis options object and customize the properties. Then, run an analysis.

Create object and customize properties.

```
sources = fullfile(polyspaceroot, 'polyspace','examples','cxx','Code_Prover_Example', ...
'sources','single_file_analysis.c');
includes = fullfile(polyspaceroot, 'polyspace','examples','cxx','Code_Prover_Example', ...
'sources');
optsCP = polyspace.CodeProverOptions();
optsCP.Prog = 'MyProject';
optsCP.Sources = {sources};
optsCP.EnvironmentSettings.IncludeFolders = {includes};
optsCP.TargetCompiler.Compiler = 'gnu4.7';
optsCP.ResultsDir = tempname;
```

Generate a Polyspace project, name it using the `Prog` property, and open the project in the Polyspace interface.

```
psprj = generateProject(optsCP, optsCP.Prog);
polyspaceCodeProver(psprj);
```

You can also analyze the project from the command line. Run the analysis and open the results in the Polyspace interface.

```
results = polyspaceCodeProver(psprj, '-nodesktop');
polyspaceCodeProver('-results-dir',optsCP.ResultsDir);
```

Alternatives

If you are verifying model-generated code, use `polyspace.ModelLinkCodeProverOptions` instead.

See Also

`polyspace.ModelLinkCodeProverOptions` | `polyspace.Options` | `polyspaceCodeProver`

Topics

"Run Polyspace Analysis by Using MATLAB Scripts"

Introduced in R2016b

polyspace.ModelLinkCodeProverOptions class

Package: `polyspace`

Create Polyspace Code Prover object for generated code

Note This class is deprecated and will be removed in a future release. Use `polyspace.ModelLinkOptions` instead.

Description

Customize a Polyspace Code Prover verification from MATLAB by creating a Code Prover options object. To specify source files and customize analysis options, change the object properties.

If you are verifying handwritten code, use `polyspace.CodeProverOptions` instead.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Construction

`opts = polyspace.ModelLinkCodeProverOptions` creates a Code Prover options object with options for C code verification.

`opts = polyspace.ModelLinkCodeProverOptions(lang)` creates a Code Prover options object with options applicable for the language `lang`.

Input Arguments

lang — Language of analysis
C (default) | CPP

Language of verification specified as either `'C'` or `'CPP'`. This argument determines which properties the object has.

Example: `opts = polyspace.ModelLinkCodeProverOptions('CPP')`

Properties

The object properties are the analysis options for Polyspace Code Prover model link projects. The properties are organized in the same categories as in the Polyspace interface. The property names are a shortened version of the DOS command-line name. For syntax details, see polyspace.ModelLinkOptions.

Methods

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

To copy properties between Polyspace objects, use . You can copy corresponding properties between BugFinderOptions objects and CodeProverOptions objects.

Examples

Script Verification of Model Generated Code

This example shows how to customize and run a verification on model-generated code with MATLAB functions and objects.

Create a custom configuration that checks MISRA C 2012 rules and generates a PDF report.

```
opts = polyspace.ModelLinkCodeProverOptions('C');
opts.CodingRulesCodeMetrics.EnableMisraC3 = true;
opts.CodingRulesCodeMetrics.MisraC3Subset = 'all';
opts.MergedReporting.ReportOutputFormat = 'PDF';
opts.MergedReporting.EnableReportGeneration = true;
```

Generate code from psdemo_model_link_sl.

```
model = 'psdemo_model_link_sl';
load_system(model);
slbuild(model);
```

Add the configuration to pslinkoptions object.

```
prjfile = opts.generateProject('model_link_opts');
mlopts = pslinkoptions(model);
mlopts.EnablePrjConfigFile = true;
mlopts.PrjConfigFile = prjfile;
mlopts.VerificationMode = 'CodeProver';
```

Run the verification.

```
[polyspaceFolder, resultsFolder] = pslinkrun(model);
```

Alternatives

If you are verifying handwritten code, use `polyspace.CodeProverOptions` instead.

See Also

`polyspace.CodeProverOptions` | polyspace.ModelLinkOptions | `polyspaceCodeProver` |
`pslinkrun`

Topics

"Run Polyspace Analysis by Using MATLAB Scripts"

Introduced in R2016b

polyspace.GenericTargetOptions class

Package: `polyspace`

Create a generic target configuration

Description

Create a custom target for a Polyspace analysis if your target processor does not match one of the predefined targets,.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Construction

`genericTarget = polyspace.GenericTargetOptions` creates a generic target that you can customize. To specify the sizes and alignment of data types, change the properties of the object. For instance:

```
target = polyspace.GenericTargetOptions;
target.CharNumBits = 16;
```

Properties

For more details about any of the properties below, see `Generic target options`.

Alignment — Largest alignment of struct or array objects
32 (default) | 16 | 8

Largest alignment of struct or array objects, specified as 32, 16, or 8. Comparable with the DOS/UNIX command-line option `-align`.

Example: `target.Alignment = 8`

CharNumBits — Define the number of bits for a `char`
8 (default) | 16

Define the number of bits for a `char`, specified as 8 or 16. Comparable with the DOS/UNIX command-line option `-char-is-16bits`.

Example: `target.CharNumBits = 16`

DoubleNumBits — Define the number of bits for a double
32 (default) | 64

Define the number of bits for a `double`, specified as 32 or 64. Comparable with the DOS/UNIX command-line option `-double-is-64bits`.

Example: `target.DoubleNumBits = 64`

5-147

Endianness — Endianness of target architecture
little (default) | big

Endianness of target architecture, specified as little or big. Comparable with the DOS/UNIX command-line options -little-endian or -big-endian.

Example: target.Endianess = 'big'

IntNumBits — Define the number of bits for an int
16 (default) | 32

Define the number of bits for an int, specified as 16 or 32. Comparable with the DOS/UNIX command-line option -int-is-32bits.

Example: target.IntNumBits = 32

LongLongNumBits — Define the number of bits for a long long
32 (default) | 64

Define the number of bits for a long long, specified as 32 or 64. Comparable with the DOS/UNIX command-line option -long-long-is-64bits.

Example: target.LongNumBits = 64

LongNumBits — Define the number of bits for a long
32 (default)

Define the number of bits for a long, specified as 32. Comparable with the DOS/UNIX command-line option -long-is-32bits.

Example: target.LongNumBits = 32

PointerNumBits — Define the number of bits for a pointer
16 (default) | 24 | 32

Define the number of bits for a pointer, specified as 16, 24, or 32. Comparable with the DOS/UNIX command-line options -pointer-is-24bits and -pointer-is-32bits.

Example: target.PointerNumBits = 32

ShortNumBits — Define the number of bits for a short
16 (default) | 8

Define the number of bits for an int, specified as 16 or 8. Comparable with the DOS/UNIX command-line option -short-is-8bits.

Example: target.ShortNumBits = 8

SignOfChar — Default sign of plain char
signed (default) | unsigned

Default sign of plain char, specified as signed or unsigned. Comparable with the DOS/UNIX command-line option -default-sign-of-char.

Example: target.SignOfChar = 'unsigned'

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Customize Generic Target Settings

Use a custom target for the Polyspace analysis.

Create two objects: a `polyspace.GenericTargetOptions` object for creating a custom target and a `polyspace.Project` object for running the Polyspace analysis.

```
target = polyspace.GenericTargetOptions;
proj = polyspace.Project;
```

Customize the generic target.

```
target.Endianess = 'big';
target.LongLongNumBits = 64;
target.ShortNumBits = 8;
```

Add the custom target to the `Configuration` property of the `polyspace.Project` object.

```
proj.Configuration.TargetCompiler.Target = target;
```

You can now use the `polyspace.Project` object to run the analysis.

`Generic target options | polyspace.CodingRulesOptions | polyspace.ModelLinkOptions | polyspace.Options | polyspace.Project`

Introduced in R2016b

polyspace.CodingRulesOptions class

Package: `polyspace`

Create custom list of coding rules to check

Description

Create a custom list of coding rules to check in a Polyspace analysis.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See "Integrate Polyspace with MATLAB and Simulink".

Construction

`ruleList = polyspace.CodingRulesOptions(RuleSet)` creates the coding rules object `ruleList` for the `RuleSet` coding rule set. Set the active rules in the coding rules object.

Input Arguments

RuleSet — Standard coding rule set
`misraC` (default) | `misraC2012` | `misraAcAgc` | `misraCpp` | `jsf` | `certC` | `certCpp` | `iso17961` | `autosarCpp14`

Standard coding rule set specified as one of the coding rule acronyms.

Example: `'misraCpp'`

Data Types: `char`

Properties

For each coding rule set, an object is created with all supported rules divided into sections. By default, all rules are on. To turn off a rule, set the rule to false. For example:

```
misraRules = polyspace.CodingRulesOptions('misraC');
misraRules.Section_20_Standard_libraries.rule_20_1 = false;
```

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Customize List of Coding Rules to Check

Customize the coding rules that are checked in a Polyspace analysis. Since all rules are enabled by default, you can create a custom subset by disabling some rules.

Create two objects: a `polyspace.CodingRulesOptions` object for setting coding rules and a `polyspace.Project` object for running the Polyspace analysis.

```
misraRules = polyspace.CodingRulesOptions('misraC2012');
proj = polyspace.Project;
```

Customize the coding rule list by turning off rules 2.1-2.7.

```
misraRules.Section_2_Unused_code.rule_2_1 = false;
misraRules.Section_2_Unused_code.rule_2_2 = false;
misraRules.Section_2_Unused_code.rule_2_3 = false;
misraRules.Section_2_Unused_code.rule_2_4 = false;
misraRules.Section_2_Unused_code.rule_2_5 = false;
misraRules.Section_2_Unused_code.rule_2_6 = false;
misraRules.Section_2_Unused_code.rule_2_7 = false;
```

Add the customized list of coding rules to the `Configuration` property of the `polyspace.Project` object.

```
proj.Configuration.CodingRulesCodeMetrics.MisraC3Subset = misraRules;
proj.Configuration.CodingRulesCodeMetrics.EnableMisraC3 = true;
proj.Configuration.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;
```

You have to enable checkers selection by file because the Polyspace run uses an XML file underneath to enable the coding rule checkers. The XML file is saved in a `.settings` subfolder of the results folder.

You can now use the `polyspace.Project` object to run the analysis. For instance, you can enter:

```
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
run(proj, 'bugfinder');
```

Create Coding Rules Object Using Rule Numbers to Enable

Suppose that you want to specify a subset of MISRA C: 2012 rules for the analysis. Instead of enumerating rules that you want disabled, you can specify the rules that you want to keep enabled. You can also specify the rule numbers only without the MISRA C: 2012 sections containing the rules.

Specify the rule numbers in a cell array to the `createRulesObject` function defined as follows.

```
function rulesObject = createRulesObject(rulesToEnable)

%% This function takes a cell array of MISRA C:2012 rules and returns
%% a polyspace.CodingRulesOptions object with the rules enabled.
%% Example input argument: {'2.7', '3.1'}

    rulesObject = polyspace.CodingRulesOptions('misraC2012');

    % Coding Standards documents have many sections. Loop over all
    % sections.
    ruleSections = properties(rulesObject);
    for i=1:length(ruleSections)
        sectionName = ruleSections{i};
        rulesInSection = properties(rulesObject.(sectionName));
```

```
            % Loop over all rules in a section, enable or disable rule based
            % on input
            for j=1:length(rulesInSection)
                ruleNumberAsProperty = rulesInSection{j};
                ruleNumber = strrep(strrep(ruleNumberAsProperty,'rule_',''),'_','.');
                if(any(strcmp(rulesToEnable,ruleNumber)))
                    rulesObject.(sectionName).(ruleNumberAsProperty)=1;
                else
                    rulesObject.(sectionName).(ruleNumberAsProperty)=0;
                end
            end
        end
end
```

For instance, to enable rules 1.1 and 2.2, enter:

```
createRulesObject({'1.1','2.2'})
```

See Also
`polyspace.ModelLinkOptions` | `polyspace.Options` | `polyspace.Project`

Introduced in R2016b

polyspace.CodeProverResults

Read Polyspace Code Prover results from MATLAB

Description

Read Polyspace Code Prover analysis results to MATLAB tables by using this object. You can obtain a high-level overview of results or details such as each instance of a run-time check.

Note Before you run Polyspace from MATLAB, you must link your Polyspace and MATLAB installations. See .

Creation

Syntax

```
resObj = polyspace.CodeProverResults(resultsFolder)
proj = polyspace.Project; resObj = proj.Results;
```

Description

`resObj = polyspace.CodeProverResults(resultsFolder)` creates an object for reading a specific set of Code Prover results into MATLAB tables. Use the object methods to read the results.

`proj = polyspace.Project; resObj = proj.Results;` creates a `polyspace.Project` object with a `Results` property. If you run a Code Prover analysis, this property is a `polyspace.CodeProverResults` object.

Input Arguments

resultsFolder — Name of result folder
character vector

Name of result folder, specified as a character vector. The folder must directly contain the results file with extension `.psbf`. Even if the results file resides in a *subfolder* of the specified folder, it cannot be accessed.

If the folder is not in the current folder, `resultsFolder` must include a full or relative path.

Example: `'C:\Polyspace\Results\'`

Object Functions

getSummary	View number of Polyspace results organized by results type (Bug Finder) or color and file (Code Prover)
getResults	View all instances of Bug Finder or Code Prover results
variableAccess	View C/C++ global variables along with read/write operations

Examples

Read Existing Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(polyspaceroot,'polyspace','examples','cxx','Code_Prover_Example',...
'Module_1','CP_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary (resObj);
resTable = getResults (resObj);
```

Run Analysis and Read Results to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a `main` function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project;

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;


% Run analysis
cpStatus = run(proj, 'codeProver');

% Read results
resObj = proj.Results;
cpSummary = getResults(resObj, 'readable');
```

See Also

Introduced in R2017a

copyTo

Class: `polyspace.Options`
Package: `polyspace`

Copy common settings between Polyspace options objects

Syntax

```
optsFrom.copyTo(optsTo)
```

Description

`optsFrom.copyTo(optsTo)` copies the common options from `optsFrom` to `optsTo`. The options objects do not need to be the same type of options object. This method copies only properties that are common between the two objects.

Input Arguments

optsFrom — Options object you want to copy properties from
`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object that you want to copy properties from, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

optsTo — Options object you want to copy properties to
`polyspace.Options` object

Option object that you want to copy properties to, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

Examples

Copy Polyspace Options Object

This example shows how to set the properties of one options object and then copy that object to another one.

Create a Polyspace options object and set properties.

```
opts1 = polyspace.Options();
opts1.Prog = 'DataRaceProject';
opts1.Sources = {'datarace.c'};
opts1.TargetCompiler.Compiler = 'gnu4.9';
```

Create another object and use copyTo to copy over options from the previous object.

```
opts2 = polyspace.Options();
opts1.copyTo(opts2);
```

See Also
generateProject | polyspace.ModelLinkOptions | polyspace.Options

Introduced in R2016b

generateProject

Class: `polyspace.Options`
Package: `polyspace`

Generate psprj project from options object

Syntax

`opts.generateProject(projectName)`

Description

`opts.generateProject(projectName)` creates a `.psprj` project called `projectName` from the options specified in the `polyspace.Options` object `opts`. You can open a `.psprj` project in the user interface of the Polyspace desktop products.

Input Arguments

`opts` — Options object to convert into a `psprj` file
`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object convert into a `psprj` file, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

`projectName` — Project file name
character vector

Project file name specified as a character vector. This argument is used as the name of the `psprj` file.

Example: `'myProject'`

Examples

Generate Project from a Bug Finder Options Object

This example shows how to create and use a Polyspace project that was generated from an options object.

Create a Bug Finder object and set properties.

```
sources = fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example',...
    'sources','numerical.c');
opts = polyspace.Options();
opts.Prog = 'MyProject';
opts.Sources = {sources};
opts.TargetCompiler.Compiler = 'gnu4.7';
```

Generate a Polyspace project. Name the project using the `Prog` property.

```
psprj = opts.generateProject(opts.Prog);
```

Run a Bug Finder analysis using one of these commands. Both commands produce identical analysis results. The only difference is that the `psprj` project can be rerun in the Polyspace interface.

```
polyspaceBugFinder(psprj, '-nodesktop');
polyspaceBugFinder(opts);
```

To run a Code Prover analysis, use `polyspaceCodeProver` instead of `polyspaceBugFinder`.

Tips

If you want to include an options object in a `pslinkoptions` object:

1   Use this method to convert your object to a project.
2   Add the project to the `pslinkoptions` property `PrjConfig`.
3   Turn on the property `EnablePrjConfig`.

See Also
copyTo | polyspace.ModelLinkOptions | polyspace.Options

Introduced in R2016b

toScript

Class: `polyspace.Options`
Package: `polyspace`

Add Polyspace options object definition to a script

Syntax

```
filePath = opts.toScript(fileName,positionInScript)
```

Description

`filePath = opts.toScript(fileName,positionInScript)` adds the properties of a `polyspace.Options` object to a MATLAB script. The script shows the values assigned to all the properties of the object. You can run the script later to define the object in the MATLAB workspace and use it.

Input Arguments

opts — Options object with Polyspace analysis options
`polyspace.Options` or `polyspace.ModelLinkOptions` object

Option object to store in MATLAB script, specified as a `polyspace.Options` or `polyspace.ModelLinkOptions` object.

Example: `opts = polyspace.Options;`

fileName — Script name
character vector

Name or path to script, specified as a character vector. If you specify a relative path, the script is created in subfolder of the current working folder.

Example: `'runPolyspace.m'`

positionInScript — Where to add object definition
`'create'` (default) | `'append'`

Position in script where the object properties are added, specified as `'create'` or `'append'`. If you specify `'append'`, the object properties are added to the end of an existing script. Otherwise, a new script is created.

Output Arguments

filePath — Full path to script
character vector

Full path to script, specified as a character vector.

Example: `'C:\myScripts\runPolyspace.m'`

See Also

copyTo | generateProject | polyspace.ModelLinkOptions | polyspace.Options

Introduced in R2017b

run

Run a Polyspace analysis

Syntax

```
run(proj, product)
```

Description

`status = run(proj, product)` runs a Polyspace Bug Finder or Polyspace Code Prover analysis using the configuration specified in the `polyspace.Project` object `proj`. The analysis results are also stored in `proj`.

Input Arguments

`proj` — Polyspace project
`polyspace.Project` object

Polyspace project with configuration and results, specified as a `polyspace.Project` object.

`product` — Type of analysis
`'bugFinder'` | `'codeProver'`

Type of analysis to run.

Output Arguments

`status` — Results of a Code Prover analysis
`true` | `false`

Status of analysis. If the analysis succeeds, the status is `false`. Otherwise, it is `true`.

The analysis can fail for multiple reasons:

- You provide source files that do not exist.
- None of your files compile. Even if one file compiles, unless you set the property `StopWithCompileError` to `true`, the analysis succeeds and returns a `false` status.

There can be many other reasons why the analysis fails. If the analysis fails, in your results folder, check the log file. You can see the results folder using the `Configuration` property of the `polyspace.Project` object:

```
proj = polyspace.Project;
proj.Configuration.ResultsDir
```

The log file is named `Polyspace_R20##n_ProjectName_date-time.log`.

Examples

Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
bfSummary = proj.Results.getSummary('defects');
```

Introduced in R2017b

getSummary

View number of Polyspace results organized by results type (Bug Finder) or color and file (Code Prover)

Syntax

```
resSummary = getSummary(resObj, resultsType)
```

Description

resSummary = getSummary(resObj, resultsType) returns the distribution of results of type resultsType in a Polyspace results set, resObj. The results set resObj can be a Bug Finder results set denoted by a polyspace.BugFinderResults object or a Code Prover results set denoted by a polyspace.CodeProverResults object.

For instance:

- If you choose to see Bug Finder defects, you can see how many defects of each type are present in the result set, for instance, how many non-initialized variables or declaration mismatches.
- If you choose to see Code Prover run-time checks, you see how many red, orange, gray and green checks are present in each file.

Examples

Read Existing Bug Finder Results to MATLAB Tables

This example shows how to read Bug Finder analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath=fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example', ...
'Module_1','BF_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.BugFinderResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary(resObj, 'defects');
resTable = getResults(resObj);
```

Run Bug Finder Analysis and Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file numerical.c. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = run(proj, 'bugFinder');

% Read results
resObj = proj.Results;
bfSummary = getSummary(resObj, 'defects');
```

Read Existing Code Prover Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(polyspaceroot,'polyspace','examples','cxx','Code_Prover_Example', ...
'Module_1','CP_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary(resObj, 'runtime');
resTable = getResults(resObj);
```

Run Code Prover Analysis and Read Results to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a `main` function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
```

```
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;


% Run analysis
cpStatus = run(proj, 'codeProver');

% Read results
resObj = proj.Results;
cpSummary = getResults(resObj, 'readable');
```

Input Arguments

resObj — Bug Finder or Code Prover results
polyspace.BugFinderResults or polyspace.CodeProverResults object

Bug Finder or Code Prover results set, specified as a polyspace.BugFinderResults or
polyspace.CodeProverResults object respectively.

resultsType — Type of Bug Finder or Code Prover analysis result
'defects' | 'runtime' | 'misraC' | 'misraCAGC' | 'misraCPP' | 'misraC2012' | 'jsf' |
'certC' | 'certCpp' | 'iso17961' | 'autosarCPP14' | 'metrics' | 'customRules'

Type of result, specified as a character vector. The default for a Bug Finder results set is 'defects'
and the default for a Code Prover results set is 'runtime'.

Entry	Meaning
'defects'	Bug Finder defects.
'runtime'	Code Prover checks for run-time errors.
'misraC'	MISRA C:2004 rules.
'misraCAGC'	MISRA C:2004 rules for generated code.
'misraCPP'	MISRA C++ rules.
'misraC2012'	MISRA C:2012 rules.
'jsf'	JSF C++ rules.
'certC'	CERT C rules.
'certCpp'	CERT C++ rules.
'iso17961'	ISO/IEC TS 17961 rules.
'autosarCPP14'	AUTOSAR C++ 14 rules.
'metrics'	Code complexity metrics.
'customRules'	Custom rules enforcing naming conventions for identifiers.

Output Arguments

**resSummary — Distribution of Bug Finder results by result type or Code Prover run-time
checks by check color and file**
table

Distribution of results, specified as a table. For instance:

- If you choose to see a summary of Bug Finder defects, an extract of the table looks like this:

Category	Defect	Impact	Total
Concurrency	Data race	High	2
Concurrency	Deadlock	High	1
Data flow	Non-initialized variable	High	2

The table above shows that the result set contains two data races, one deadlock and two non-initialized variables.

- If you choose to see a summary of Code Prover run-time checks, an extract of the table looks like this:

File	Proven	Green	Red	Gray	Orange
file1.c	92.0%	87	3	2	8
file2.c	97.7%	41	0	1	1

The table above shows that `file1.c` has:

- 3 red, 2 gray and 8 orange checks.
- 92% of operations proven.

  In other words, of every 100 operations that the verification checked, 92 operations were proven green, red or gray. See "Code Prover Result and Source Code Colors".

For more information on MATLAB tables, see "Tables".

See Also

polyspace.BugFinderResults | polyspace.CodeProverResults

Introduced in R2017a

getResults

View all instances of Bug Finder or Code Prover results

Syntax

```
resTable = getResults(resObj, content)
```

Description

`resTable = getResults(resObj, content)` returns a table showing all results in a Polyspace result set, `resObj`. The results set `resObj` can be a Bug Finder results set denoted by a `polyspace.BugFinderResults` object or a Code Prover results set denoted by a `polyspace.CodeProverResults` object. You can manipulate the table to produce graphs and statistics about your results that you cannot obtain readily from the user interface.

Examples

Read Existing Bug Finder Results to MATLAB Tables

This example shows how to read Bug Finder analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example', ...
'Module_1','BF_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.BugFinderResults(userResPath);
```

Read results to MATLAB tables using the object.

```
resSummary = getSummary(resObj, 'defects');
resTable = getResults(resObj);
```

Run Bug Finder Analysis and Read Results to MATLAB Tables

Run a Polyspace Bug Finder analysis on the demo file `numerical.c`. Configure these options:

- Specify GCC 4.9 as your compiler.

- Save the results in a `results` subfolder of the current working folder.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
```

```matlab
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');

% Run analysis
bfStatus = proj.run('bugFinder');

% Read results
resObj = proj.Results;
bfSummary = getResults(resObj, 'readable');
```

Read Existing Code Prover Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```matlab
resPath=fullfile(polyspaceroot,'polyspace','examples','cxx','Code_Prover_Example', ...
'Module_1','CP_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```matlab
resObj = polyspace.CodeProverResults(userResPath);
```

Read results to MATLAB tables using the object.

```matlab
resSummary = getSummary (resObj);
resTable = getResults (resObj);
```

Run Code Prover Analysis and Read Results to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a `main` function must be generated, if it does not exist in the source code.

```matlab
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;


% Run analysis
cpStatus = proj.run('codeProver');

% Read results
```

```
resObj = proj.Results;
cpSummary = getResults(resObj, 'readable');
```

Input Arguments

resObj — Bug Finder or Code Prover results
polyspace.BugFinderResults or polyspace.CodeProverResults object

Bug Finder or Code Prover results set, specified as a `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object respectively.

content — Result information to include
'' (default) | 'readable'

Amount of information to be included for each result. If you specify `''`, all information is included. If you specify `'readable'`, the following information is not included:

- ID: Unique number for a result for the current analysis.
- Group: , , Check groupsMISRA C:2012 groups, etc.
- Status, Severity, Comment: Information that *you* enter about a result.

If you do not specify this argument, the full table is included.

See "Export Polyspace Analysis Results".

Output Arguments

resTable — Results of a Bug Finder or Code Prover analysis
table

Table showing all results from a single Bug Finder or Code Prover analysis. For each result, the table has information such as file, family, and so on. If a particular information is not available for a result, the entry in the table states `<undefined>`.

For more information on:

- The columns of the table, see "Export Polyspace Analysis Results".
- MATLAB tables, see "Tables".

See Also
polyspace.BugFinderResults | polyspace.CodeProverResults

Introduced in R2017a

variableAccess

View C/C++ global variables along with read/write operations

Syntax

```
varList = variableAccess(resObj)
```

Description

`varList = variableAccess(resObj)` returns the distribution of global variables in a Code Prover result set denoted by the `polyspace.CodeProverResults` object `resObj`. The list also contains all read and write operations on the global variables.

Examples

Read Global Variables from Existing Results to MATLAB Tables

This example shows how to read Code Prover analysis results from MATLAB.

Copy a demo result set to a temporary folder.

```
resPath = fullfile(polyspaceroot,'polyspace','examples','cxx','Code_Prover_Example', ...
'Module_1','CP_Result');
userResPath = tempname;
copyfile(resPath,userResPath);
```

Create the results object.

```
resObj = polyspace.CodeProverResults(userResPath);
```

Read list of global variables to MATLAB tables using the object.

```
varList = variableAccess(resObj);
```

Run Code Prover Analysis and Read Global Variables to MATLAB Tables

Run a Polyspace Code Prover analysis on the demo file `single_file_analysis.c`. Configure these options:

- Specify GCC 4.9 as your compiler.
- Save the results in a `results` subfolder of the current working folder.
- Specify that a `main` function must be generated, if it does not exist in the source code.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples',...
    'cxx', 'Code_Prover_Example', 'sources', 'single_file_analysis.c')};
```

```
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd,'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;


% Run analysis
cpStatus = proj.run('codeProver');

% Read results
resObj = proj.Results;
cpSummary = variableAccess(resObj);
```

Input Arguments

resObj — Code Prover results
polyspace.CodeProverResults object

Code Prover results set, specified as a `polyspace.CodeProverResults` object.

Output Arguments

varList — Distribution of global variables
table

Table showing all global variables from a single Code Prover analysis along with read and write operations on them.

- For each global variable, the table has information such as data type, number of times accessed, and so on.
- For each read or write operation, the table has information such as file and function name, line number, and so on.

If a particular information is not available for a result, the entry in the table states `<undefined>`.

For more information on:

- The columns of the table, see "Export Global Variable List".
- MATLAB tables, see "Tables".

See Also
`polyspace.CodeProverResults`

Topics
"Export Global Variable List"

Introduced in R2017a

pslinkoptions Properties

Properties for the `pslinkoptions` object

Description

You can create a `pslinkoptions` object to customize your analysis at the command-line. Use these properties to specify configuration options, where and how to store results, additional files to include, and data range modes.

Properties

Configuration Options

VerificationSettings — Coding rule and configuration settings for C code
`'PrjConfig'` (default) | `'PrjConfigAndMisraAGC'` | `'PrjConfigAndMisra'` |
`'PrjConfigAndMisraC2012'` | `'MisraAGC'` | `'Misra'` | `'MisraC2012'`

Coding rule and configuration settings for C code specified as:

- `'PrjConfig'` – Inherit options from the project configuration.
- `'PrjConfigAndMisraAGC'` – Inherit options from the project configuration and enable MISRA AC AGC rule checking.
- `'PrjConfigAndMisra'` – Inherit options from the project configuration and enable MISRA C:2004 rule checking.
- `'PrjConfigAndMisraC2012'` – Inherit options from the project configuration and enable MISRA C:2012 guideline checking.
- `'MisraAGC'` – Enable MISRA AC AGC rule checking. This option runs only compilation and rule checking.
- `'Misra'` – Enable MISRA C:2004 rule checking. This option runs only compilation and rule checking.
- `'MisraC2012'` – Enable MISRA C:2012 rule checking. This option runs only compilation and guideline checking.

Example: `opt.VerificationSettings = 'PrjConfigAndMisraC2012'`

VerificationMode — Polyspace mode
`'CodeProver'` (default) | `'BugFinder'`

Polyspace mode specified as `'BugFinder'`, for a Bug Finder analysis, or `'CodeProver'`, for a Code Prover verification.

Example: `opt.VerificationMode = 'BugFinder';`

EnablePrjConfigFile — Allow a custom configuration file
`false` (default) | `true`

Allows a custom configuration file instead of the default configuration specified as true or false. Use the `PrjConfigFile` option to specify the configuration file.

Example: `opt.EnablePrjConfigFile = true;`

PrjConfigFile — Custom configuration file
`''` (default) | full path to a `.psprj` file

Custom configuration file to use instead of the default configuration specified by the full path to a `.psprj` file. Use the `EnablePrjConfigFile` option to use this configuration file during your analysis.

Example: `opt.PrjConfigFile = 'C:\Polyspace\config.psprj';`

CheckConfigBeforeAnalysis — Configuration check before analysis
`'OnWarn'` (default) | `'OnHalt'` | `'Off'`

This property sets the level of configuration checking done before the analysis starts. The configuration check before analysis is specified as:

- `'Off'` — Checks only for errors. Stops if errors are found.
- `'OnWarn'` — Stops for errors. Displays a message for warnings.
- `'OnHalt'` — Stops for errors and warnings.

Example: `opt.CheckConfigBeforeAnalysis = 'OnHalt';`

Results

ResultDir — Results folder name and location
`'C:\Polyspace_Results\results_$ModelName$'` (default) | folder name | folder path

Results folder name and location specified as the local folder name or the folder path. This folder is where Polyspace writes the analysis results. This folder name can be either an absolute path or a path relative to the current folder. The text `$ModelName$` is replaced with the name of the original model.

Example: `opt.ResultDir = '\results_v1_$ModelName$';`

AddSuffixToResultDir — Add unique number to the results folder name
`false` (default) | `true`

Add unique number to the results folder name specified as true or false. If true, a unique number is added to the end of every new result. Using this option helps you avoid overwriting the previous results folders.

Example: `opt.AddSuffixToResultDir = true;`

OpenProjectManager — Open the Polyspace environment
`false` (default) | `true`

Open the Polyspace environment to monitor the progress of the analysis, specified as true or false. Afterward, you can review the results.

Example: `opt.OpenProjectManager = true;`

AddToSimulinkProject — Add results to the open Simulink project
`false` (default) | `true`

Add your results to the currently open Simulink project, if any, specified as true or false. This option allows you to keep your Polyspace results organized with the rest of your project files. If a Simulink project is not open, the results are not added to a Simulink project.

Example: `opt.AddToSimulinkProject = true;`

Additional Files

EnableAdditionalFileList — Allow an additional file list
`false` (default) | `true`

Allow an additional file list to be analyzed, specified as true or false. Use with the `AdditionalFileList` option.

Example: `opt.EnableAdditionalFileList = true;`

AdditionalFileList — List of additional files to be analyzed
`{0x1 cell}` (default) | cell array of files

List of additional files to be analyzed specified as a cell array of files. Use with the `EnableAdditionalFileList` option to add these files to the analysis.

Example: `opt.AdditionalFileList = {'sources\file1.c', 'sources\file2.c'};`

Data Types: `cell`

Data Ranges

InputRangeMode — Enable design range information
`'DesignMinMax'` (default) | `'FullRange'`

Enable design range information specified as `'DesignMinMax'`, to use data ranges defined in blocks and workspaces, or `'FullRange'`, to treat inputs as full-range values.

Example: `opt.InputRangeMode = 'FullRange';`

ParamRangeMode — Enable constant parameter values
`'None'` (default) | `'DesignMinMax'`

Enable constant parameter values, specified as `'None'`, to use constant parameters values specified in the code, or `'DesignMinMax'` to use a range defined in blocks and workspaces.

Example: `opt.ParamRangeMode = 'DesignMinMax';`

OutputRangeMode — Enable output assertions
`'None'` (default) | `'DesignMinMax'`

Enable output assertions specified by `'None'`, to not apply assertions, or `'DesignMinMax'` to apply assertions to outputs using a range defined in blocks and workspace.

Example: `opt.ParamRangeMode = 'DesignMinMax';`

Embedded Coder Only

ModelRefVerifDepth — Depth of verification
`'Current model only'` (default) | `'1'` | `'2'` | `'3'` | `'All'`

Specify the depth for analyzing the models that are referenced by the current model.

- `'Current Model Only'`: Analyze only the top model without analyzing the referenced models. For instance, you might use this option when the referenced models are library models.
- `'1'`, `'2'`, or `'3'`: Analyze referenced models up to the specified depth in the reference hierarchy. For instance, to analyze the models that are referenced by the top model, specify the property

ModelRefVerifDepth as '1'. To analyze models that are referenced by the first level of references, specify this property as '2'.

- 'All': Analyze all referenced models with the current model.

For Embedded Coder only

Example: opt.ModelRefVerifDepth = '3';

ModelRefByModelRefVerif — Model reference analysis mode
false (default) | true

Specify whether you want to analyze all referenced models together, or analyze the models individually.

- false: Analyze the top model and the referenced models together. For instance, you might want to use this option to check for integration or scaling issues.
- true: Analyze the top model and the referenced models individually.

For Embedded Coder only

Example: opt.ModelRefByModelRefVerif = true;

CxxVerificationSettings — Coding rule and configuration settings for C++ code
'PrjConfig' (default) | 'PrjConfigAndMisraCxx' | 'PrjConfigAndJSF' | 'MisraCxx' | 'JSF'

Coding rule and configuration settings for C++ code specified as:

- 'PrjConfig' – Inherit options from project configuration and run complete analysis.
- 'PrjConfigAndMisraCxx' – Inherit options from project configuration, enable MISRA C++ rule checking, and run complete analysis.
- 'PrjConfigAndJSF' – Inherit options from project configuration, enable JSF rule checking, and run complete analysis.
- 'MisraCxx' – Enable MISRA C++ rule checking, and run compilation phase only.
- 'JSF' – Enable JSF rule checking, and run compilation phase only.

Only for Embedded Coder

Example: opt.CxxVerificationSettings = 'MisraCxx';

TargetLink Only

AutoStubLUT — Lookup Table code usage
false (default) | true

Lookup Table code usage, specified as true or false.

- true — use Lookup Table code during the analysis.
- false — stub Lookup Table code.

Only for TargetLink

Example: opts.AutoStubLUT = true;

See Also

pslinkoptions | pslinkrun

polyspace.Project.Configuration Properties

Customize Polyspace analysis of handwritten code with options object properties

---

Note The Automatic Orange Tester properties will be removed in a future release. See "Compatibility Considerations".

Description

To customize your Polyspace analysis, use these `polyspace.Options` or `polyspace.Project.Configuration` properties. Each property corresponds to an analysis option on the **Configuration** pane in the Polyspace user interface.

The properties are grouped using the same categories as the **Configuration** pane. This page only shows what values each property can take. For details about:

- The different options, see the analysis option reference pages.
- How to create and use the object, see `polyspace.Options` or `polyspace.Project`.

  The same properties are also available with the deprecated classes `polyspace.BugFinderOptions` and `polyspace.CodeProverOptions`.

Each property description below also highlights if the option affects only one of Bug Finder or Code Prover.

Note Some options might not be available depending on the language setting of the object. You can set the source code language (`Language`) to `'C'`, `'CPP'` or `'C-CPP'` during object creation, but cannot change it later.

Properties

Advanced

`Additional` — Additional flags for analysis
character vector

Additional flags for analysis specified as a character vector.

For more information, see `Other`.

Example: `opts.Advanced.Additional = '-extra-flags -option -extra-flags value'`

`PostAnalysisCommand` — Command or script software should execute after analysis finishes
character vector

Command or script software should execute after analysis finishes, specified as a character vector.

For more information, see `Command/script to apply after the end of the code verification (-post-analysis-command)`.

5-177

Example: `opts.Advanced.PostAnalysisCommand = '"C:\Program Files\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"'`

AutomaticOrangeTester — (To be removed) Run the Automatic Orange Tester
false (default) | true

This property affects Code Prover analysis only.

This property will be removed in a future release.

Run the Automatic Orange Tester after verification, specified as true or false.

For more information, see `Automatic Orange Tester (-automatic-orange-tester)`.

Example: `opts.Advanced.AutomaticOrangeTester = true`

AutomaticOrangeTesterLoopMaxIteration — (To be removed) Number of loop iterations after which Automatic Orange Tester considers infinite loop
1000 (default) | positive integer

This property affects Code Prover analysis only.

This property will be removed in a future release.

Number of loop iterations after which Automatic Orange Tester considers the test an infinite loop, specified as a positive integer, maximum of 1000.

For more information, see `Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)`.

Example: `opts.Advanced.AutomaticOrangeTesterLoopMaxIteration = 500`

AutomaticOrangeTesterTestsNumber — (To be removed) Number of tests that Automatic Orange Tester must run
500 (default) | positive integer

This property affects Code Prover analysis only.

This property will be removed in a future release.

Number of tests that Automatic Orange Tester must run, specified as a positive integer, maximum of 100,000.

For more information, see `Number of automatic tests (-automatic-orange-tester-tests-number)`.

Example: `opts.Advanced.AutomaticOrangeTesterTestsNumber = 1000`

AutomaticOrangeTesterTimeout — (To be removed) Time in seconds allowed for a single test in Automatic Orange Tester
5 (default) | positive integer

This property affects Code Prover analysis only.

This property will be removed in a future release.

Time in seconds allowed for a single test in Automatic Orange Tester, specified as a positive integer, maximum of 60.

For more information, see `Maximum test time (-automatic-orange-tester-timeout)`.

Example: `opts.Advanced.AutomaticOrangeTesterTimeout = 10`

BugFinderAnalysis (Affects Bug Finder Only)

CheckersList — List of custom checkers to activate
`polyspace.DefectsOptions` object | cell array of defect acronyms

This property affects Bug Finder analysis only.

List of custom checkers to activate specified by using the name of a `polyspace.DefectsOptions` object or a cell array of defect acronyms. To use this custom list in your analysis, set `CheckersPreset` to `custom`.

For more information, see `polyspace.DefectsOptions`.

Example: `defects = polyspace.DefectsOptions;`
`opts.BugFinderAnalysis.CheckersList = defects`

Example: `opts.BugFinderAnalysis.CheckersList = {'INT_ZERO_DIV','FLOAT_ZERO_DIV'}`

CheckersPreset — Subset of Bug Finder defects
`'default'` (default) | `'all'` | `'CWE'` | `'custom'`

This property affects Bug Finder analysis only.

Preset checker list, specified as a character vector of one of the preset options: `'default'`, `'all'`, `'CWE'`, or `'custom'`. To use `'custom'`, specify a value for the property `BugFinderAnalysis.CheckersList`.

For more information, see `Find defects (-checkers)`.

Example: `opts.BugFinderAnalysis.CheckersPreset = 'all'`

ChecksUsingSystemInputValues — Activate stricter checks for system inputs
false (default) | true

This property affects Bug Finder analysis only.

Activate stricter checks that consider all possible value for:

- Global variables.
- Reads of volatile variables.
- Returns of stubbed functions.
- Inputs to functions specified with **SystemInputsFrom**.

The analysis considers all possible values for a subset of **Numerical** and **Static memory** defects.

This property is equivalent to the **Run stricter checks considering all values of system inputs** check box in the Polyspace interface.

For more information, see `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`

Example: `opts.BugFinderAnalysis.ChecksUsingSystemInputValues = true`

EnableCheckers — Activate defect checking
true (default) | false

This property affects Bug Finder analysis only.

Activate defect checking, specified as true or false. Setting this property to false disables all defects. If you want to disable defect checking but still get results, turn on coding rules checking or code metric checking.

This property is equivalent to the **Find defects** check box in the Polyspace interface.

Example: `opts.BugFinderAnalysis.EnableCheckers = false`

SystemInputsFrom — List of functions for which you run stricter checks
'auto' (default) | 'uncalled' | 'all' | 'custom'

This property affects Bug Finder analysis only.

Functions for which you want to run stricter checks that consider all possible values of the function inputs. Specify the list of functions as `'auto'`, `'uncalled'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names.

To enable this option, set `BugFinderAnalysis.ChecksUsingSystemInputValues = true`.

For more information, see `Consider inputs to these functions (-system-inputs-from)`

Example: `opts.BugFinderAnalysis.SystemInputsFrom = 'custom=foo,bar'`

ChecksAssumption (Affects Code Prover Only)

AllowNegativeOperandInShift — Allow left shift operations on a negative number
false (default) | true

This property affects Code Prover analysis only.

Allow left shift operations on a negative number, specified as true or false.

For more information, see `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`.

Example: `opts.ChecksAssumption.AllowNegativeOperandInShift = true`

AllowNonFiniteFloats — Incorporate infinities and/or NaNs
false (default) | true

This property affects Code Prover analysis only.

Incorporate infinities and/or NaNs, specified as true or false.

For more information, see `Consider non finite floats (-allow-non-finite-floats)`.

Example: `opts.ChecksAssumption.AllowNonFiniteFloats = true`

AllowPtrArithOnStruct — Allow arithmetic on pointer to a structure field so that it points to another field
false (default) | true

This property affects Code Prover analysis only.

Allow arithmetic on pointer to a structure field so that it points to another field, specified as true or false.

For more information, see `Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)`.

Example: `opts.ChecksAssumption.AllowPtrArithOnStruct = true`

CheckInfinite — Detect floating-point operations that result in infinities
`'allow'` (default) | `'warn-first'` | `'forbid'`

This property affects Code Prover analysis only.

Detect floating-point operations that result in infinities.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `Infinities (-check-infinite)`.

Example: `opts.ChecksAssumption.CheckInfinite = 'forbid'`

CheckNan — Detect floating-point operations that result in NaN-s
`'allow'` (default) | `'warn-first'` | `'forbid'`

This property affects Code Prover analysis only.

Detect floating-point operations that result in NaN-s.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `NaNs (-check-nan)`.

Example: `opts.ChecksAssumption.CheckNan = 'forbid'`

CheckSubnormal — Detect operations that result in subnormal floating point values
`'allow'` (default) | `'warn-first'` | `'warn-all'` | `'forbid'`

This property affects Code Prover analysis only.

Detect operations that result in subnormal floating point values.

For more information, see `Subnormal detection mode (-check-subnormal)`.

Example: `opts.ChecksAssumption.CheckSubnormal = 'forbid'`

DetectPointerEscape — Find cases where a function returns a pointer to one of its local variables
false (default) | true

This property affects Code Prover analysis only.

Find cases where a function returns a pointer to one of its local variables, specified as true or false.

For more information, see `Detect stack pointer dereference outside scope (-detect-pointer-escape)`.

Example: `opts.ChecksAssumption.DetectPointerEscape = true`

DisableInitializationChecks — Disable checks for noninitialized variables and pointers
false (default) | true

This property affects Code Prover analysis only.

Disable checks for noninitialized variables and pointers, specified as true or false.

For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Example: `opts.ChecksAssumption.DisableInitializationChecks = true`

PermissiveFunctionPointer — Allow type mismatch between function pointers and the functions they point to
false (default) | true

This property affects Code Prover analysis only.

Allow type mismatch between function pointers and the functions they point to, specified as true or false.

For more information, see `Permissive function pointer calls (-permissive-function-pointer)`.

Example: `opts.ChecksAssumption.PermissiveFunctionPointer = true`

SignedIntegerOverflows — Behavior of signed integer overflows
`'forbid'` (default) | `'allow'` | `'warn-with-wrap-around'`

This property affects Code Prover analysis only.

Enable the check for signed integer overflows and the assumptions to make following an overflow specified as `'forbid'`, `'allow'`, or `'warn-with-wrap-around'`.

For more information, see `Overflow mode for signed integer (-signed-integer-overflows)`.

Example: `opts.ChecksAssumption.SignedIntegerOverflows = 'warn-with-wrap-around'`

SizeInBytes — Allow a pointer with insufficient memory buffer to point to a structure
false (default) | true

This property affects Code Prover analysis only.

Allow a pointer with insufficient memory buffer to point to a structure, specified as true or false.

For more information, see `Allow incomplete or partial allocation of structures (-size-in-bytes)`.

Example: `opts.ChecksAssumption.SizeInBytes = true`

UncalledFunctionCheck — Detect functions that are not called directly or indirectly from main or another entry-point function
'none' (default) | 'never-called' | 'called-from-unreachable' | 'all'

This property affects Code Prover analysis only.

Detect functions that are not called directly or indirectly from main or another entry-point function, specified as none, never-called, called-from-unreachable, or all.

For more information, see Detect uncalled functions (-uncalled-function-checks).

Example: opts.ChecksAssumption.UncalledFunctionCheck = 'all'

UnsignedIntegerOverflows — Behavior of unsigned integer overflows
'allow' (default) | 'forbid' | 'warn-with-wrap-around'

This property affects Code Prover analysis only.

Enable the check for unsigned integer overflows and the assumptions to make following an overflow, specified as 'forbid', 'allow', or 'warn-with-wrap-around'.

For more information, see Overflow mode for unsigned integer (-unsigned-integer-overflows).

Example: opts.ChecksAssumption.UnsignedIntegerOverflows = 'allow'

CodeProverVerification (Affects Code Prover only)

ClassAnalyzer — Classes that you want to verify
'all' (default) | 'none' | 'custom=*class1[,class2,...]*'

This property affects Code Prover analysis only.

Classes that you want to verify, specified as 'all', 'none', or as a character array beginning with custom= followed by a comma-separated list of class names.

For more information, see Class (-class-analyzer).

Example: opts.CodeProverVerification.ClassAnalyzer = 'custom=myClass1,myClass2'

ClassAnalyzerCalls — Class methods that you want to verify
'unused' (default) | 'all' | 'all-public' | 'inherited-all' | 'inherited-all-public' | 'unused-public' | 'inherited-unused' | 'inherited-unused-public' | 'custom=*method1[,method2,...]*'

This property affects Code Prover analysis only.

Class methods that you want to verify, specified as one of the predefined sets or as a character array beginning with custom= followed by a comma-separated list of method names.

For more information, see Functions to call within the specified classes (-class-analyzer-calls).

Example: opts.CodeProverVerification.ClassAnalyzerCalls = 'unused-public'

ClassOnly — Analyze only class methods
false (default) | true

This property affects Code Prover analysis only.

Analyze only class methods, specified as true or false.

For more information, see `Analyze class contents only (-class-only)`.

Example: `opts.CodeProverVerification.ClassOnly = true`

EnableMain — Use `main` function provided in application
false (default) | true

This property affects Code Prover analysis only.

Use `main` function provided in application, specified as true or false. If you set this property to false, the analysis generates a `main` function, if it is not present in the source files.

For more information, see `Verify whole application`.

Example: `opts.CodeProverVerification.EnableMain = true`

FunctionsCalledBeforeMain — Functions that you want the generated main to call ahead of other functions
cell array of function names

This property affects Code Prover analysis only.

Functions that you want the generated main to call ahead of other functions, specified as a cell array of function names.

For more information, see `Initialization functions (-functions-called-before-main)`.

Example: `opts.CodeProverVerification.FunctionsCalledBeforeMain = {'func1','func2'}`

Main — Use a Microsoft Visual C++ extensions of main
'_tmain' (default) | 'wmain' | '_tWinMain' | 'wWinMain' | 'WinMain' | 'DllMain'

This property applies to a Code Prover analysis only .

Use a Microsoft Visual C++ extension of main, specified as one of the predefined main extensions.

For more information, see `Main entry point (-main)`.

Example: `opts.CodeProverVerification.Main = 'wmain'`

MainGenerator — Generate a main function if it is not present in source files
true (default) | false

This property applies to a Code Prover analysis only .

Generate a main function if it is not present in source files, specified as true or false.

For more information, see `Verify module or library (-main-generator)`.

Example: `opts.CodeProverVerification.MainGenerator = false`

MainGeneratorCalls — Functions that you want the generated main to call after the initialization functions
'unused' (default) | 'none' | 'all' | 'custom=*function1[,function2,...]*'

This property applies to a Code Prover analysis only .

Functions that you want the generated main to call after the initialization functions, specified as 'unused', 'all', 'none', or as a character array beginning with custom= followed by a comma-separated list of function names.

For more information, see Functions to call (-main-generator-calls).

Example: opts.CodeProverVerification.MainGeneratorCalls = 'all'

MainGeneratorWriteVariables — Global variables that you want the generated main to initialize
'uninit' (C++ default) | 'public' (C default) | 'none' | 'all' | 'custom=*variable1[,variable2,...]*'

This property applies to a Code Prover analysis only .

Global variables that you want the generated main to initialize, specified as one of the predefined sets, or as a character array beginning with custom= followed by a comma-separated list of variable names.

For more information, see Variables to initialize (-main-generator-writes-variables).

Example: opts.CodeProverVerification.MainGeneratorWriteVariables = 'all'

NoConstructorsInitCheck — Do not check if class constructor initializes class members
false (default) | true

This property applies to a Code Prover analysis only .

Do not check if class constructor initializes class members, specified as true or false.

For more information, see Skip member initialization check (-no-constructors-init-check).

Example: opts.CodeProverVerification.NoConstructorsInitCheck = true

UnitByUnit — Verify each source file independently of other source files
false (default) | true

This property affects Code Prover analysis only.

Verify each source file independently of other source files, specified as true or false.

For more information, see Verify files independently (-unit-by-unit).

Example: opts.CodeProverVerification.UnitByUnit = true

UnitByUnitCommonSource — Files that you want to include with each source file during a file-by-file verification
cell array of file paths

This property affects Code Prover analysis only.

Files that you want to include with each source file during a file-by-file verification, specified as a cell array of file paths.

For more information, see `Common source files (-unit-by-unit-common-source)`.

Example: `opts.CodeProverVerification.UnitByUnitCommonSource = {'/inc/file1.h','/inc/file2.h'}`

CodingRulesCodeMetrics

AcAgcSubset — Subset of MISRA AC AGC rules to check
`'OBL-rules'` (default) | `'OBL-REC-rules'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA AC AGC rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA AC AGC (-misra-ac-agc)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA AC AGC rules, also set `EnableAcAgc` to true.

Example: `opts.CodingRulesCodeMetrics.AcAgcSubset = 'all-rules'`

Data Types: `char`

AllowedPragmas — Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied
cell array of character vectors

Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied, specified as a cell array of character vectors. This property affects only MISRA C:2004 or MISRA AC AGC rule checking.

For more information, see `Allowed pragmas (-allowed-pragmas)`.

Example: `opts.CodingRulesCodeMetrics.AllowedPragmas = {'pragma_01','pragma_02'}`

Data Types: `cell`

AutosarCpp14 — Set of AUTOSAR C++ 14 rules to check
`'all'` (default) | `'required'` | `'automated'` | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of AUTOSAR C++ 14 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check AUTOSAR C++ 14 security checks (-autosar-cpp14)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check AUTOSAR C++ 14 rules, also set `EnableAutosarCpp14` to true.

Example: `opts.CodingRulesCodeMetrics.AutosarCpp14 = 'all'`

Data Types: `char`

BooleanTypes — Data types the coding rule checker must treat as effectively Boolean
cell array of character vectors

Data types that the coding rule checker must treat as effectively Boolean, specified as a cell array of character vectors.

For more information, see `Effective boolean types (-boolean-types)`.

Example: `opts.CodingRulesCodeMetrics.BooleanTypes = {'boolean1_t','boolean2_t'}`

Data Types: `cell`

CertC — Set of CERT C rules and recommendations to check
`'all'` (default) | `'publish-2016'` | `'all-rules'` | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of CERT C rules and recommendations to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C security checks (-cert-c)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `from-file` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C rules and recommendations, also set `EnableCertC` to true.

Example: `opts.CodingRulesCodeMetrics.CertC = 'all'`

Data Types: `char`

CertCpp — Set of CERT C++ rules to check
`'all'` (default) | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of CERT C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C++ security checks (-cert-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C++ rules, also set `EnableCertCpp` to true.

Example: `opts.CodingRulesCodeMetrics.CertCpp = 'all'`

Data Types: `char`

CheckersSelectionByFile — File that defines custom set of coding standard checkers
full file path of `.xml` file

File where you define a custom set of coding standards checkers to check, specified as a `.xml` file. You can, in the same file, define a custom set of checkers for each of the coding standards that Polyspace supports. To create a file that defines a custom selection of coding standard checkers, in the Polyspace interface, select a coding standard on the **Coding Standards & Code Metrics** node of the **Configuration** pane and click **Edit**.

For more information, see `Set checkers by file (-checkers-selection-file)`.

Example: `opts.CodingRulesCodeMetrics.CheckersSelectionByFile = 'C:\ps_settings\coding_rules\custom_rules.xml'`

Data Types: `char`

CodeMetrics — Activate code metric calculations
false (default) | true

Activate code metric calculations, specified as true or false. If this property is turned off, Polyspace does not calculate code metrics even if you upload your results to Polyspace Metrics.

For more information about the code metrics, see `Calculate code metrics (-code-metrics)`.

If you assign a coding rules options object to this property, an XML file gets created automatically with the rules specified.

Example: `opts.CodingRulesCodeMetrics.CodeMetrics = true`

EnableAcAgc — Check MISRA AC AGC rules
false (default) | true

Check MISRA AC AGC rules, specified as true or false. To customize which rules are checked, use `AcAgcSubset`.

For more information about the MISRA AC AGC checker, see `Check MISRA AC AGC (-misra-ac-agc)`.

Example: `opts.CodingRulesCodeMetrics.EnableAcAgc = true;`

EnableAutosarCpp14 — Check AUTOSAR C++ 14 rules
false (default) | true

This property affects Bug Finder only.

Check AUTOSAR C++ 14 rules, specified as true or false. To customize which rules are checked, use `AutosarCpp14`.

For more information about the AUTOSAR C++ 14 checker, see `Check AUTOSAR C++ 14 security checks (-autosar-cpp14)`.

Example: `opts.CodingRulesCodeMetrics.EnableAutosarCpp14 = true;`

EnableCertC — check CERT C rules and recommendations
false (default) | true

This property affects Bug Finder only.

Check CERT C rules and recommendations, specified as true or false. To customize which rules are checked, use `CertC`.

For more information about the CERT C checker, see `Check CERT-C security checks (-cert-c)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertC = true;`

EnableCertCpp — check CERT C++ rules
false (default) | true

This property affects Bug Finder only.

Check CERT C++ rules, specified as true or false. To customize which rules are checked, use `CertCpp`.

For more information about the CERT C++ checker, see `Check CERT-C++ security checks (-cert-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertCpp = true;`

EnableCheckersSelectionByFile — Check custom set of coding standard checkers
false (default) | true

Check custom set of coding standard checkers, specified as true or false. Use with `CheckersSelectionByFile` and these coding standards:

- `opts.CodingRulesCodeMetrics.AutosarCpp14='from-file'`
- `opts.CodingRulesCodeMetrics.CertC='from-file'`
- `opts.CodingRulesCodeMetrics.CertCpp='from-file'`
- `opts.CodingRulesCodeMetrics.Iso17961='from-file'`
- `opts.CodingRulesCodeMetrics.JsfSubset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraC3Subset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCSubset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCppSubset='from-file'`

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;`

EnableCustomRules — Check custom coding rules
false (default) | true

Check custom coding rules, specified as true or false. The file you specify with `CheckersSelectionByFile` defines the custom coding rules.

Use with `EnableCheckersSelectionByFile`.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCustomRules = true;`

EnableIso17961 — check ISO-17961 rules
false (default) | true

This property affects Bug Finder only.

Check ISO/IEC TS 17961 rules, specified as true or false. To customize which rules are checked, use `Iso17961`.

For more information about the ISO-17961 checker, see `Check ISO-17961 security checks (-iso-17961)`.

Example: `opts.CodingRulesCodeMetrics.EnableIso17961 = true;`

EnableJsf — Check JSF C++ rules
false (default) | true

Check JSF C++ rules, specified as true or false. To customize which rules are checked, use `JsfSubset`.

For more information, see `Check JSF C++ rules (-jsf-coding-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableJsf = true;`

EnableMisraC — Check MISRA C:2004 rules
false (default) | true

Check MISRA C:2004 rules, specified as true or false. To customize which rules are checked, use `MisraCSubset`.

For more information, see `Check MISRA C:2004 (-misra2)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC = true;`

EnableMisraC3 — Check MISRA C:2012 rules
false (default) | true

Check MISRA C:2012 rules, specified as true or false. To customize which rules are checked, use `MisraC3Subset`.

For more information about the MISRA C:2012 checker, see `Check MISRA C:2012 (-misra3)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC3 = true;`

EnableMisraCpp — Check MISRA C++:2008 rules
false (default) | true

Check MISRA C++:2008 rules, specified as true or false. To customize which rules are checked, use `MisraCppSubset`.

For more information about the MISRA C++:2008 checker, see `Check MISRA C++ rules (-misra-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraCpp = true;`

Iso17961 — Set of ISO-17961 rules to check
'all' (default) | 'decidable' | polyspace.CodingRulesOptions object | 'from-file'

This property affects Bug Finder only.

Set of ISO/IEC TS 17961 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check ISO-17961 security checks (-iso-17961)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check ISO/IEC TS 17961 rules, also set `EnableIso17961` to true.

Example: `opts.CodingRulesCodeMetrics.Iso17961 = 'all'`

Data Types: `char`

JsfSubset — Subset of JSF C++ rules to check
'shall-rules' (default) | 'shall-will-rules' | 'all-rules' | polyspace.CodingRulesOptions object | 'from-file'

Subset of JSF C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check JSF C++ rules (-jsf-coding-rules)`.

- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check JSF C++ rules, set `EnableJsf` to true.

Example: `opts.CodingRulesCodeMetrics.JsfSubset = 'all-rules'`

Data Types: `char`

Misra3AgcMode — Use the MISRA C:2012 categories for automatically generated code
false (default) | true

Use the MISRA C:2012 categories for automatically generated code, specified as true or false.

For more information, see `Use generated code requirements (-misra3-agc-mode)`.

Example: `opts.CodingRulesCodeMetrics.Misra3AgcMode = true;`

MisraC3Subset — Subset of MISRA C:2012 rules to check
`'mandatory-required'` (default) | `'mandatory'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2012 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2012 (-misra3)`.

- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2012 rules, also set `EnableMisraC3` to true.

Example: `opts.CodingRulesCodeMetrics.MisraC3Subset = 'all'`

Data Types: `char`

MisraCSubset — Subset of MISRA C:2004 rules to check
`'required-rules'` (default) | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2004 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2004 (-misra2)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

    You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2004 rules, also set `EnableMisraC` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCSubset = 'all-rules'`

Data Types: `char`

MisraCppSubset — Subset of MISRA C++ rules
`'required-rules'` (default) | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C++:2008 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C++ rules (-misra-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

    You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C++ rules, set `EnableMisraCpp` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCppSubset = 'all-rules'`

Data Types: `char`

EnvironmentSettings

Dos — Consider that file paths are in MS-DOS style
true (default) | false

Consider that file paths are in MS-DOS style, specified as true or false.

For more information, see `Code from DOS or Windows file system (-dos)`.

Example: `opts.EnvironmentSettings.Dos = true;`

IncludeFolders — Include folders needed for compilation
cell array of include folder paths

Include folders needed for compilation, specified as a cell array of the include folder paths.

To specify all subfolders of a folder, use folder path followed by `**`, for instance, `'C:\includes\**'`. The notation follows the syntax of the `dir` function. See also "Specify Multiple Source Files".

For more information, see `-I`.

Example: `opts.EnvironmentSettings.IncludeFolders = {'/includes','/com1/inc'};`

Example: `opts.EnvironmentSettings.IncludeFolders = {'C:\project1\common\includes'};`

Data Types: `cell`

Includes — Files to be #include-ed by each C file
cell array of files

Files to be `#include`-ed by each C source file in the analysis, specified by a cell array of files.

For more information, see `Include (-include)`.

Example: `opts.EnvironmentSettings.Includes = {'/inc/inc_file.h','/inc/inc_math.h'}`

NoExternC — Ignore linking errors inside extern blocks
false (default) | true

Ignore linking errors inside extern blocks, specified as true or false.

For more information, see `Ignore link errors (-no-extern-c)`.

Example: `opts.EnvironmentSettings.NoExternC = false;`

PostPreProcessingCommand — Command or script to run on source files after preprocessing
character vector

Command or script to run on source files after preprocessing, specified as a character vector of the command to run.

For more information, see `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Example: Linux — `opts.EnvironmentSettings.PostPreProcessingCommand = [pwd,'/replace_keyword.pl']`

Example: Windows — `opts.EnvironmentSettings.PostPreProcessingCommand = '"C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"'`

StopWithCompileError — Stop analysis if a file does not compile
false (default) | true

Stop analysis if a file does not compile, specified as true or false.

For more information, see `Stop analysis if a file does not compile (-stop-if-compile-error)`.

Example: `opts.EnvironmentSettings.StopWithCompileError = true;`

InputsStubbing

DataRangeSpecifications — Constrain global variables, function inputs, and return values of stubbed functions
file path

Constrain global variables, function inputs, and return values of stubbed functions specified by the path to an XML constraint file. For more information about the constraint file, see "Specify External Constraints".

For more information about this option, see `Constraint setup (-data-range-specifications)`.

Example: `opts.InputsStubbing.DataRangeSpecifications = 'C:\project\constraint_file.xml'`

DoNotGenerateResultsFor — Files on which you do not want analysis results
`'include-folders'` (default) | `'all-headers'` | `'custom=folder1[,folder2,...]'`

Files on which you do not want analysis results, specified by `'include-folders'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.GenerateResultsFor`. For more information, see `Do not generate results for (-do-not-generate-results-for)`.

Example: `opts.InputsStubbing.DoNotGenerateResultsFor = 'custom=C:\project\file1.c,C:\project\file2.c'`

GenerateResultsFor — Files on which you want analysis results
`'source-headers'` (default) | `'all-headers'` | `'custom=folder1[,folder2,...]'`

Files on which you want analysis results, specified by `'source-headers'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.DoNotGenerateResultsFor`. For more information, see `Generate results for sources and (-generate-results-for)`.

Example: `opts.InputsStubbing.GenerateResultsFor = 'custom=C:\project\includes_common_1,C:\project\includes_common_2'`

FunctionsToStub — Functions to stub during analysis
cell array of function names

This property affects Code Prover analysis only.

Functions to stub during analysis, specified as a cell array of function names.

For more information, see `Functions to stub (-functions-to-stub)`.

Example: `opts.InputsStubbing.FunctionsToStub = {'func1', 'func2'}`

NoDefInitGlob — Consider global variables as uninitialized
false (default) | true

This property affects Code Prover analysis only.

Consider global variables as uninitialized, specified as true or false.

For more information, see `Ignore default initialization of global variables (-no-def-init-glob)`.

Example: `opts.InputsStubbing.NoDefInitGlob = true`

NoStlStubs — Do not use Polyspace implementations of functions in the Standard Template Library
false (default) | true

This property applies only to a Code Prover analysis of C++ code.

Do not use Polyspace implementations of functions in the Standard Template Library, specified as true or false.

For more information, see `No STL stubs (-no-stl-stubs)`.

Example: `opts.InputsStubbing.NoStlStubs = true`

StubECoderLookupTables — Specify that the analysis must stub functions in the generated code that use lookup tables
true (default) | false

This property applies only to a Code Prover analysis of code generated from models.

Specify that the analysis must stub functions in the generated code that use lookup tables. By replacing the functions with stubs, the analysis assumes more precise return values for the functions.

For more information, see `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`.

Example: `opts.InputsStubbing.StubECoderLookupTables = true`

Macros

DefinedMacros — Macros to be replaced
cell array of macros

In preprocessed code, macros are replaced by the definition, specified in a cell array of macros and definitions. Specify the macro as `Macro=Value`. If you want Polyspace to ignore the macro, leave the `Value` blank. A macro with no equal sign replaces all instances of that macro by 1.

For more information, see `Preprocessor definitions (-D)`.

Example: `opts.Macros.DefinedMacros = {'uint32=int','name3=','var'}`

UndefinedMacros — Macros to undefine
cell array of macros

In preprocessed code, macros are undefined, specified by a cell array of macros to undefine.

For more information, see `Disabled preprocessor definitions (-U)`.

Example: `opts.Macros.DefinedMacros = {'name1','name2'}`

MergedComputingSettings

AddToResultsRepositoryBugFinder — Upload Bug Finder results to Polyspace Metrics web dashboard
false (default) | true

This property affects Bug Finder analysis only.

Upload Bug Finder analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryBugFinder = true;`

AddToResultsRepositoryCodeProver — Upload Code Prover results to Polyspace Metrics web dashboard
false (default) | true

This property affects Code Prover analysis only.

Upload Code Prover analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryCodeProver = true;`

BatchBugFinder — Send Bug Finder analysis to remote server
false (default) | true

This property affects Bug Finder analysis only.

Send Bug Finder analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchBugFinder = true;`

BatchCodeProver — Send Code Prover analysis to remote server
false (default) | true

This property affects Code Prover analysis only.

Send Code Prover analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchCodeProver = true;`

FastAnalysis — Run Bug Finder analysis using faster local mode
false (default) | true

This property affects Bug Finder analysis only.

Use fast analysis mode for Bug Finder analysis, specified as true or false.

For more information, see `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Example: `opts.MergedComputingSettings.FastAnalysis = true;`

MergedReporting

EnableReportGeneration — Generate a report after the analysis
false (default) | true

After the analysis, generate a report, specified as true or false.

For more information, see `Generate report`.

Example: `opts.MergedReporting.EnableReportGeneration = true`

ReportOutputFormat — Output format of generated report
`'Word'` (default) | `'HTML'` | `'PDF'`

Output format of generated report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Output format (-report-output-format)`.

Example: `opts.MergedReporting.ReportOutputFormat = 'PDF'`

BugFinderReportTemplate — Template for generating Bug Finder analysis report
`'BugFinderSummary'` (default) | `'BugFinder'` | `'SecurityCWE'` | `'CodeMetrics'` | `'CodingStandards'`

This property affects a Bug Finder analysis only.

Template for generating analysis report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.BugFinderReportTemplate = 'CodeMetrics'`

CodeProverReportTemplate — Template for generating Code Prover analysis report
`'Developer'` (default) | `'CallHierarchy'` | `'CodeMetrics'` | `'CodingStandards'` | `'DeveloperReview'` | `'Developer_withGreenChecks'` | `'Quality'` | `'VariableAccess'`

This property affects a Code Prover analysis only.

Template for generating analysis report, specified as one of the predefined report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.CodeProverReportTemplate = 'CodeMetrics'`

Multitasking

`ArxmlMultitasking` — Specify path of ARXML files to parse for multitasking configuration
cell array of file paths

Specify the path to the ARXML files the software parses to set up your multitasking configuration.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `autosar`.

For more information, see `ARXML files selection (-autosar-multitasking)`

Example: `opts.Multitasking.ArxmlMultitasking={'C:\Polyspace_Workspace\AUTOSAR\myFile.arxml'}`

`CriticalSectionBegin` — Functions that begin critical sections
cell array of critical section function names

Functions that begin critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionEnd`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionBegin = {'function1:cs1','function2:cs2'}`

`CriticalSectionEnd` — Functions that end critical sections
cell array of critical section function names

Functions that end critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionBegin`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionEnd = {'function1:cs1','function2:cs2'}`

`CyclicTasks` — Specify functions that represent cyclic tasks
cell array of function names

Specify functions that represent cyclic tasks.

To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Cyclic tasks (-cyclic-tasks)`.

Example: `opts.Multitasking.CyclicTasks = {'function1','function2'}`

`EnableConcurrencyDetection` — Enable automatic detection of certain families of threading functions
false (default) | true

This property affects Code Prover analysis only.

Enable automatic detection of certain families of threading functions, specified as true or false.

For more information, see `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

Example: `opts.Multitasking.EnableConcurrencyDetection = true`

EnableExternalMultitasking — Enable automatic multitasking configuration from external file definitions
false (default) | true

Enable multitasking configuration of your projects from external files you provide. Configure multitasking from ARXML files for an AUTOSAR project, or from OIL files for an OSEK project.

Activate this option to enable `Multitasking.ArxmlMultitasking` or `Multitasking.OsekMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.EnableExternalMultitasking = 1`

EnableMultitasking — Configure multitasking manually
false (default) | true

Configure multitasking manually by specifying `true`. This property activates the other manual, multitasking properties.

For more information, see `Configure multitasking manually`.

Example: `opts.Multitasking.EnableMultitasking = 1`

EntryPoints — Functions that serve as entry-points to your multitasking application
cell array of entry-point function names

Functions that serve as entry-points to your multitasking application specified as a cell array of entry-point function names. To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Tasks (-entry-points)`.

Example: `opts.Multitasking.EntryPoints = {'function1','function2'}`

ExternalMultitaskingType — Specify type of file to parse for multitasking configuration
'osek' (default) | 'autosar'

Specify the type of file the software parses to set up your multitasking configuration:

- For `osek` type, the analysis looks for OIL files in the file or folder paths that you specify.
- For `autosar` type, the analysis looks for ARXML files in the file paths that you specify.

To activate this option, specify `Multitasking.EnableExternalMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.ExternalMultitaskingType = 'autosar'`

`Interrupts` — Specify functions that represent nonpreemptable interrupts
cell array of function names

Specify functions that represent nonpreemptable interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Interrupts (-interrupts)`.

Example: `opts.Multitasking.Interrupts = {'function1','function2'}`

`InterruptsDisableAll` — Specify routine that disable interrupts
cell array with one function name

This property affects Bug Finder analysis only.

Specify function that disables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsDisableAll = {'function'}`

`InterruptsEnableAll` — Specify routine that reenable interrupts
cell array with one function name

This property affects Bug Finder analysis only.

Specify function that reenables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsEnableAll = {'function'}`

`OsekMultitasking` — Specify path of OIL files to parse for multitasking configuration
`'auto'` (default) | `'custom=folder1[,folder2,...]'`

Specify the path to the OIL files the software parses to set up your multitasking configuration:

- In the mode specified with `'auto'`, the analysis uses OIL files in your project source and include folders, but not their subfolders.
- In the mode specified with `'custom=folder1[,folder2,...]'`, the analysis uses the OIL files at the specified path, and the path subfolders.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `osek`.

For more information, see `OIL files selection (-osek-multitasking)`

Example: `opts.Multitasking.OsekMultitasking = 'custom=file_path, dir_path'`

`TemporalExclusion` — Entry-point functions that cannot execute concurrently
cell array of entry-point function names

5-201

Entry-point functions that cannot execute concurrently specified as a cell array of entry-point function names. Each set of exclusive tasks is one cell array entry with functions separated by spaces. To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Temporally exclusive tasks (-temporal-exclusions-file)`.

Example: `opts.Multitasking.TemporalExclusion = {'function1 function2', 'function3 function4 function5'}` where function1 and function2 are temporally exclusive, and function3, function4, and function 5 are temporally exclusive.

Precision (Affects Code Prover Only)

ContextSensitivity — Store call context information to identify function call that caused errors
`'none'` (default) | `'auto'` | `'custom=function1[,function2,...]'`

This property affects Code Prover analysis only.

Store call context information to identify a function call that caused errors, specified as `none`, `auto`, or as a character array beginning with `custom=` followed by a list of comma-separated function names.

For more information, see `Sensitivity context (-context-sensitivity)`.

Example: `opts.Precision.ContextSensitivity = 'auto'`

Example: `opts.Precision.ContextSensitivity = 'custom=func1'`

ModulesPrecision — Source files you want to verify at higher precision
cell array of file names and precision levels

This property affects Code Prover analysis only.

Source files that you want to verify at higher precision, specified as a cell array of file names without the extension and precision levels using this syntax: *filename*:O*level*

For more information, see `Specific precision (-modules-precision)`.

Example: `opts.Precision.ModulesPrecision = {'file1:O0', 'file2:O3'}`

OLevel — Precision level for the verification
2 (default) | 0 | 1 | 3

This property affects Code Prover analysis only.

Precision level for the verification, specified as 0, 1, 2, or 3.

For more information, see `Precision level (-O)`.

Example: `opts.Precision.OLevel = 3`

PathSensitivityDelta — Avoid certain verification approximations for code with fewer lines
positive integer

This property affects Code Prover analysis only.

Avoid certain verification approximations for code with fewer lines, specified as a positive integer representing how sensitive the analysis is. Higher values can increase verification time exponentially.

For more information, see `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

Example: `opts.Precision.PathSensitivityDelta = 2`

Timeout — Time limit on your verification
character vector

This property affects Code Prover analysis only.

Time limit on your verification, specified as a character vector of time in hours.

For more information, see `Verification time limit (-timeout)`.

Example: `opts.Precision.Timeout = '5.75'`

To — Number of times the verification process runs
`'Software Safety Analysis level 2'` (default) | `'Software Safety Analysis level 0'` | `'Software Safety Analysis level 1'` | `'Software Safety Analysis level 3'` | `'Software Safety Analysis level 4'` | `'Source Compliance Checking'` | `'other'`

This property affects Code Prover analysis only.

Number of times the verification process runs, specified as one of the preset analysis levels.

For more information, see `Verification level (-to)`.

Example: `opts.Precision.To = 'Software Safety Analysis level 3'`

Scaling (Affects Code Prover Only)

Inline — Functions on which separate results must be generated for each function call
cell array of function names

This property affects Code Prover analysis only.

Functions on which separate results must be generated for each function call, specified as a cell array of function names.

For more information, see `Inline (-inline)`.

Example: `opts.Scaling.Inline = {'func1','func2'}`

KLimiting — Limit depth of analysis for nested structures
positive integer

This property affects Code Prover analysis only.

Limit depth of analysis for nested structures, specified as a positive integer indicating how many levels into a nested structure to verify.

For more information, see `Depth of verification inside structures (-k-limiting)`.

Example: `opts.Scaling.KLimiting = 3`

TargetCompiler

Compiler — Compiler that builds your source code
'generic' (default) | 'gnu3.4' | 'gnu4.6' | 'gnu4.7' | 'gnu4.8' | 'gnu4.9' | 'gnu5.x' |
'gnu6.x' | 'gnu7.x' | 'clang3.x' | 'clang4.x' | 'clang5.x' | 'visual9.0' | 'visual10' |
'visual11.0' | 'visual12.0' | 'visual14.0' | 'visual15.x' | 'keil' | 'iar' | 'armcc' |
'armclang' | 'codewarrior' | 'diab' | 'greenhills' | 'iar-ew' | 'renesas' | 'tasking' |
'ti'

Compiler that builds your source code.

For more information, see Compiler (-compiler).

Example: opts.TargetCompiler.Compiler = 'Visual11.0'

CppVersion — Specify C++ standard version followed in code
'defined-by-compiler' (default) | 'cpp03' | 'cpp11' | 'cpp14' | 'cpp17'

Specify C++ standard version followed in code, specified as a character vector.

For more information, see C++ standard version (-cpp-version).

Example: opts.TargetCompiler.CppVersion = 'cpp11';

CVersion — Specify C standard version followed in code
'defined-by-compiler' (default) | 'c90' | 'c99' | 'c11'

Specify C standard version followed in code, specified as a character vector.

For more information, see C standard version (-c-version).

Example: opts.TargetCompiler.CVersion = 'c90';

DivRoundDown — Round down quotients from division or modulus of negative numbers
false (default) | true

Round down quotients from division or modulus of negative numbers, specified as true or false.

For more information, see Division round down (-div-round-down).

Example: opts.TargetCompiler.DivRoundDown = true

EnumTypeDefinition — Base type representation of enum
'defined-by-compiler' (default) | 'auto-signed-first' | 'auto-unsigned-first'

Base type representation of enum, specified by an allowed base-type set. For more information about the different values, see Enum type definition (-enum-type-definition).

Example: opts.TargetCompiler.EnumTypeDefinition = 'auto-unsigned-first'

IgnorePragmaPack — Ignore #pragma pack directives
false (default) | true

Ignore #pragma pack directives, specified as true or false.

For more information, see Ignore pragma pack directives (-ignore-pragma-pack).

Example: opts.TargetCompiler.IgnorePragmaPack = true

Language — Language of analysis
'C-CPP' (default) | 'C' | 'CPP'

This property is read-only.

Language of the analysis, specified during the object construction. This value changes which properties appear.

For more information, see `Source code language (-lang)`.

LogicalSignedRightShift — Treatment of signed bit on signed variables
'Arithmetical' (default) | 'Logical'

Treatment of signed bit on signed variables, specified as `Arithmetical` or `Logical`. For more information, see `Signed right shift (-logical-signed-right-shift)`.

Example: `opts.TargetCompiler.LogicalSignedRightShift = 'Logical'`

NoULiterals — Do not use predefined typedefs for char16_t or char32_t
false (default) | true

Do not use predefined typedefs for char16_t or char32_t, specified as true or false. For more information, see `Block char16/32_t types (-no-uliterals)`.

Example: `opts.TargetCompiler.NoULiterals = true`

PackAlignmentValue — Default structure packing alignment
'defined-by-compiler' (default) | '1' | '2' | '4' | '8' | '16'

Default structure packing alignment, specified as `'defined-by-compiler'`, `'1'`, `'2'`, `'4'`, `'8'`, or `'16'`. This property is available only for Visual C++ code.

For more information, see `Pack alignment value (-pack-alignment-value)`.

Example: `opts.TargetCompiler.PackAlignmentValue = '4'`

SfrTypes — sfr types
cell array of `sfr` keywords

sfr types, specified as a cell array of `sfr` keywords using the syntax *sfr_name=size_in_bits*. For more information, see `Sfr type support (-sfr-types)`.

This option only applies when you set `TargetCompiler.Compiler` to `keil` or `iar`.

Example: `opts.TargetCompiler.SfrTypes = {'sfr32=32'}`

SizeTTypeIs — Underlying type of size_t
'defined-by-compiler' (default) | 'unsigned-int' | 'unsigned-long' | 'unsigned-long-long'

Underlying type of `size_t`, specified as `'defined-by-compiler'`, `'unsigned-int'`, `'unsigned-long'`, or `'unsigned-long-long'`. See `Management of size_t (-size-t-type-is)`.

Example: `opts.TargetCompiler.SizeTTypeIs = 'unsigned-long'`

Target — Target processor
'i386' (default) | 'arm' | 'arm64' | 'avr' | 'c-167' | 'c166' | 'c18' | 'c28x' | 'c6000' | 'coldfire' | 'hc08' | 'hc12' | 'm68k' | 'mcore' | 'mips' | 'mpc5xx' | 'msp430' | 'necv850'

| `'powerpc'` | `'powerpc64'` | `'rh850'` | `'rl78'` | `'rx'` | `'s12z'` | `'sharc21x61'` | `'sparc'` | `'superh'` | `'tms320c3x'` | `'tricore'` | `'x86_64'` | generic target object

Set size of data types and endianness of processor, specified as one of the predefined target processors or a generic target object.

For more information about the predefined processors, see `Target processor type (-target)`.

For more information about creating a generic target, see `polyspace.GenericTargetOptions`.

Example: `opts.TargetCompiler.Target = 'hc12'`

WcharTTypeIs — Underlying type of `wchar_t`
`'defined-by-compiler'` (default) | `'signed-short'` | `'unsigned-short'` | `'signed-int'` | `'unsigned-int'` | `'signed-long'` | `'unsigned-long'`

Underlying type of `wchar_t`, specified as `'defined-by-compiler'`, `'signed-short'`, `'unsigned-short'`, `'signed-int'`, `'unsigned-int'`, `'signed-long'`, or `'unsigned-long'`. See `Management of wchar_t (-wchar-t-type-is)`.

Example: `opts.TargetCompiler.WcharTTypeIs = 'unsigned-int'`

VerificationAssumption (Affects Code Prover Only)

ConsiderVolatileQualifierOnFields — Assume that volatile qualified structure fields can have all possible values at any point in code
false (default) | true

This property affects Code Prover analysis only.

Assume that volatile qualified structure fields can have all possible values at any point in code.

For more information, see `Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields)`.

Example: `opts.VerificationAssumption.ConsiderVolatileQualifierOnFields = true`

ConstraintPointersMayBeNull — Specify that environment pointers can be NULL unless constrained otherwise
false (default) | true

This property affects Code Prover analysis only.

Specify that environment pointers can be NULL unless constrained otherwise.

For more information, see `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.

Example: `opts.VerificationAssumption.ConstraintPointersMayBeNull = true`

FloatRoundingMode — Rounding modes to consider when determining the results of floating-point arithmetic
`to-nearest` (default) | `all`

This property affects Code Prover analysis only.

Rounding modes to consider when determining the results of floating-point arithmetic, specified as `to-nearest` or `all`.

For more information, see `Float rounding mode (-float-rounding-mode)`.

Example: `opts.VerificationAssumption.FloatRoundingMode = 'all'`

RespectTypesInFields — Do not cast nonpointer fields of a structure to pointers
false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer fields of a structure to pointers, specified as true or false.

For more information, see `Respect types in fields (-respect-types-in-fields)`.

Example: `opts.VerificationAssumption.RespectTypesInFields = true`

RespectTypesInGlobals — Do not cast nonpointer global variables to pointers
false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer global variables to pointers, specified as true or false.

For more information, see `Respect types in global variables (-respect-types-in-globals)`.

Example: `opts.VerificationAssumption.RespectTypesInGlobals = true`

Other Properties

Author — Project author
username of current user (default) | character vector

Name of project author, specified as a character vector.

For more information, see `-author`.

Example: `opts.Author = 'JaneDoe'`

ImportComments — Import comments and justifications from previous analysis
character vector

To import comments and justifications from a previous analysis, specify the path to the results folder of the previous analysis.

You can also point to a previous results folder to see only new results compared to the previous run. See "Compare Results from Different Polyspace Runs by Using MATLAB Scripts".

For more information, see `-import-comments`

Example: `opts.ImportComments = fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example','Module_1','BF_Result')`

Prog — Project name
`PolyspaceProject` (default) | character vector

Project name, specified as a character vector.

For more information, see `-prog`.

Example: `opts.Prog = 'myProject'`

ResultsDir — Location to store results
folder path

Location to store results, specified as a folder path. By default, the results are stored in the current folder.

For more information, see `-results-dir`.

You can also create a separate results folder for each new run. See "Compare Results from Different Polyspace Runs by Using MATLAB Scripts".

Example: `opts.ResultsDir = 'C:\project\myproject\results\'`

Sources — Source files
cell array of files

Source files to analyze, specified as a cell array of files.

To specify all files in a folder, use folder path followed by *, for instance, `'C:\src\*'`. To specify all files in a folder and its subfolders, use folder path followed by **, for instance, `'C:\src\**'`. The notation follows the syntax of the `dir` function. See also "Specify Multiple Source Files".

For more information, see `-sources`.

Example: `opts.Sources = {'file1.c', 'file2.c', 'file3.c'}`

Example: `opts.Sources = {'project/src1/file1.c', 'project/src2/file2.c', 'project/src3/file3.c'}`

Version — Project version number
`'1.0'` (default) | character array of a number

Version number of project, specified as a character array of a number. This option is useful if you upload your results to Polyspace Metrics. If you increment version numbers each time that you reanalyze your object, you can compare the results from two versions in Polyspace Metrics.

For more information, see `-v[ersion]`.

Example: `opts.Version = '2.3'`

Compatibility Considerations

Automatic Orange Tester will be removed
Not recommended starting in R2020b

The Automatic Orange Tester will be removed in a future release. If you use these properties in your scripts, remove them (`opts=polyspace.Options('C')`):

- `opts.Advanced.AutomaticOrangeTester`
- `opts.Advanced.AutomaticOrangeTesterLoopMaxIteration`
- `opts.Advanced.AutomaticOrangeTesterTestsNumber`

- `opts.Advanced.AutomaticOrangeTesterTimeout`

See Also

Topics
"Analysis Options in Polyspace Code Prover"

Introduced in R2017a

polyspace.ModelLinkOptions Properties

Customize Polyspace analysis of generated code with options object properties

Description

To customize your Polyspace analysis of generated code, modify the `polyspace.ModelLinkOptions` object properties. Each property corresponds to an analysis option on the **Configuration** pane in the Polyspace user interface.

The properties are grouped using the same categories as the **Configuration** pane. This page only shows what values each property can take. For details about:

- The different options, see the analysis options reference pages.
- How to create and use the object, see `polyspace.ModelLinkOptions`.

  The same properties are also available with the deprecated classes `polyspace.ModelLinkBugFinderOptions` and `polyspace.ModelLinkCodeProverOptions`.

Each property description below also highlights if the option affects only one of Bug Finder or Code Prover.

Note Some options might not be available depending on the language setting of the object. You can set the source code language (`Language`) to `'C'`, `'CPP'` or `'C-CPP'` during object creation, but cannot change it later.

Properties

Advanced

Additional — Additional flags for analysis
character vector

Additional flags for analysis specified as a character vector.

For more information, see `Other`.

Example: `opts.Advanced.Additional = '-extra-flags -option -extra-flags value'`

PostAnalysisCommand — Command or script software should execute after analysis finishes
character vector

Command or script software should execute after analysis finishes, specified as a character vector.

For more information, see `Command/script to apply after the end of the code verification (-post-analysis-command)`.

Example: `opts.Advanced.PostAnalysisCommand = '"C:\Program Files\perl\win32\bin\perl.exe" "C:\My_Scripts\send_email"'`

AutomaticOrangeTester — (To be removed) Run the Automatic Orange Tester
false (default) | true

This property affects Code Prover analysis only.

This property will be removed in a future release.

Run the Automatic Orange Tester after verification, specified as true or false.

For more information, see `Automatic Orange Tester (-automatic-orange-tester)`.

Example: `opts.Advanced.AutomaticOrangeTester = true`

AutomaticOrangeTesterLoopMaxIteration — (To be removed) Number of loop iterations after which Automatic Orange Tester considers infinite loop
1000 (default) | positive integer

This property affects Code Prover analysis only.

This property will be removed in a future release.

Number of loop iterations after which Automatic Orange Tester considers the test an infinite loop, specified as a positive integer, maximum of 1000.

For more information, see `Maximum loop iterations (-automatic-orange-tester-loop-max-iteration)`.

Example: `opts.Advanced.AutomaticOrangeTesterLoopMaxIteration = 500`

AutomaticOrangeTesterTestsNumber — (To be removed) Number of tests that Automatic Orange Tester must run
500 (default) | positive integer

This property affects Code Prover analysis only.

This property will be removed in a future release.

Number of tests that Automatic Orange Tester must run, specified as a positive integer, maximum of 100,000.

For more information, see `Number of automatic tests (-automatic-orange-tester-tests-number)`.

Example: `opts.Advanced.AutomaticOrangeTesterTestsNumber = 1000`

AutomaticOrangeTesterTimeout — (To be removed) Time in seconds allowed for a single test in Automatic Orange Tester
5 (default) | positive integer

This property affects Code Prover analysis only.

This property will be removed in a future release.

Time in seconds allowed for a single test in Automatic Orange Tester, specified as a positive integer, maximum of 60.

For more information, see `Maximum test time (-automatic-orange-tester-timeout)`.

Example: `opts.Advanced.AutomaticOrangeTesterTimeout = 10`

BugFinderAnalysis (Affects Bug Finder Only)

CheckersList — List of custom checkers to activate
`polyspace.DefectsOptions` object | cell array of defect acronyms

This property affects Bug Finder analysis only.

List of custom checkers to activate specified by using the name of a `polyspace.DefectsOptions` object or a cell array of defect acronyms. To use this custom list in your analysis, set `CheckersPreset` to `custom`.

For more information, see `polyspace.DefectsOptions`.

Example: `defects = polyspace.DefectsOptions;`
`opts.BugFinderAnalysis.CheckersList = defects`

Example: `opts.BugFinderAnalysis.CheckersList =`
`{'INT_ZERO_DIV','FLOAT_ZERO_DIV'}`

CheckersPreset — Subset of Bug Finder defects
`'default'` (default) | `'all'` | `'CWE'` | `'custom'`

This property affects Bug Finder analysis only.

Preset checker list, specified as a character vector of one of the preset options: `'default'`, `'all'`, `'CWE'`,or `'custom'`. To use `'custom'`, specify a value for the property `BugFinderAnalysis.CheckersList`.

For more information, see `Find defects (-checkers)`.

Example: `opts.BugFinderAnalysis.CheckersPreset = 'all'`

ChecksUsingSystemInputValues — Activate stricter checks for system inputs
false (default) | true

This property affects Bug Finder analysis only.

Activate stricter checks that consider all possible value for:

- Global variables.
- Reads of volatile variables.
- Returns of stubbed functions.
- Inputs to functions specified with **SystemInputsFrom**.

The analysis considers all possible values for a subset of **Numerical** and **Static memory** defects.

This property is equivalent to the **Run stricter checks considering all values of system inputs** check box in the Polyspace interface.

For more information, see `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`

Example: `opts.BugFinderAnalysis.ChecksUsingSystemInputValues = true`

EnableCheckers — Activate defect checking
true (default) | false

This property affects Bug Finder analysis only.

Activate defect checking, specified as true or false. Setting this property to false disables all defects. If you want to disable defect checking but still get results, turn on coding rules checking or code metric checking.

This property is equivalent to the **Find defects** check box in the Polyspace interface.

Example: `opts.BugFinderAnalysis.EnableCheckers = false`

SystemInputsFrom — List of functions for which you run stricter checks
`'auto'` (default) | `'uncalled'` | `'all'` | `'custom'`

This property affects Bug Finder analysis only.

Functions for which you want to run stricter checks that consider all possible values of the function inputs. Specify the list of functions as `'auto'`, `'uncalled'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names.

To enable this option, set `BugFinderAnalysis.ChecksUsingSystemInputValues = true`.

For more information, see `Consider inputs to these functions (-system-inputs-from)`

Example: `opts.BugFinderAnalysis.SystemInputsFrom = 'custom=foo,bar'`

ChecksAssumption (Affects Code Prover Only)

AllowNegativeOperandInShift — Allow left shift operations on a negative number
true (default) | false

This property affects Code Prover analysis only.

Allow left shift operations on a negative number, specified as true or false.

For more information, see `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`.

Example: `opts.ChecksAssumption.AllowNegativeOperandInShift = true`

AllowNonFiniteFloats — Incorporate infinities and/or NaNs
false (default) | true

This property affects Code Prover analysis only.

Incorporate infinities and/or NaNs, specified as true or false.

For more information, see `Consider non finite floats (-allow-non-finite-floats)`.

Example: `opts.ChecksAssumption.AllowNonFiniteFloats = true`

AllowPtrArithOnStruct — Allow arithmetic on pointer to a structure field so that it points to another field
false (default) | true

This property affects Code Prover analysis only.

Allow arithmetic on pointer to a structure field so that it points to another field, specified as true or false.

For more information, see `Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)`.

Example: `opts.ChecksAssumption.AllowPtrArithOnStruct = true`

CheckInfinite — Detect floating-point operations that result in infinities
`'allow'` (default) | `'warn-first'` | `'forbid'`

This property affects Code Prover analysis only.

Detect floating-point operations that result in infinities.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `Infinities (-check-infinite)`.

Example: `opts.ChecksAssumption.CheckInfinite = 'forbid'`

CheckNan — Detect floating-point operations that result in NaN-s
`'allow'` (default) | `'warn-first'` | `'forbid'`

This property affects Code Prover analysis only.

Detect floating-point operations that result in NaN-s.

To activate this option, specify `ChecksAssumption.AllowNonFiniteFloats`.

For more information, see `NaNs (-check-nan)`.

Example: `opts.ChecksAssumption.CheckNan = 'forbid'`

CheckSubnormal — Detect operations that result in subnormal floating point values
`'allow'` (default) | `'warn-first'` | `'warn-all'` | `'forbid'`

This property affects Code Prover analysis only.

Detect operations that result in subnormal floating point values.

For more information, see `Subnormal detection mode (-check-subnormal)`.

Example: `opts.ChecksAssumption.CheckSubnormal = 'forbid'`

DetectPointerEscape — Find cases where a function returns a pointer to one of its local variables
false (default) | true

This property affects Code Prover analysis only.

Find cases where a function returns a pointer to one of its local variables, specified as true or false.

For more information, see `Detect stack pointer dereference outside scope (-detect-pointer-escape)`.

Example: `opts.ChecksAssumption.DetectPointerEscape = true`

DisableInitializationChecks — Disable checks for noninitialized variables and pointers
false (default) | true

This property affects Code Prover analysis only.

Disable checks for noninitialized variables and pointers, specified as true or false.

For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

Example: `opts.ChecksAssumption.DisableInitializationChecks = true`

PermissiveFunctionPointer — Allow type mismatch between function pointers and the functions they point to
false (default) | true

This property affects Code Prover analysis only.

Allow type mismatch between function pointers and the functions they point to, specified as true or false.

For more information, see `Permissive function pointer calls (-permissive-function-pointer)`.

Example: `opts.ChecksAssumption.PermissiveFunctionPointer = true`

SignedIntegerOverflows — Behavior of signed integer overflows
`'warn-with-wrap-around'` (default) | `'forbid'` | `'allow'`

This property affects Code Prover analysis only.

Enable the check for signed integer overflows and the assumptions to make following an overflow specified as `'forbid'`, `'allow'`, or `'warn-with-wrap-around'`.

For more information, see `Overflow mode for signed integer (-signed-integer-overflows)`.

Example: `opts.ChecksAssumption.SignedIntegerOverflows = 'warn-with-wrap-around'`

SizeInBytes — Allow a pointer with insufficient memory buffer to point to a structure
false (default) | true

This property affects Code Prover analysis only.

Allow a pointer with insufficient memory buffer to point to a structure, specified as true or false.

For more information, see `Allow incomplete or partial allocation of structures (-size-in-bytes)`.

Example: `opts.ChecksAssumption.SizeInBytes = true`

UncalledFunctionCheck — Detect functions that are not called directly or indirectly from main or another entry-point function
`'none'` (default) | `'never-called'` | `'called-from-unreachable'` | `'all'`

This property affects Code Prover analysis only.

Detect functions that are not called directly or indirectly from main or another entry-point function, specified as `'none'`, `'never-called'`, `'called-from-unreachable'`, or `'all'`.

For more information, see `Detect uncalled functions (-uncalled-function-checks)`.

Example: `opts.ChecksAssumption.UncalledFunctionCheck = 'all'`

UnsignedIntegerOverflows — Behavior of unsigned integer overflows
`'allow'` (default) | `'forbid'` | `'warn-with-wrap-around'`

This property affects Code Prover analysis only.

Enable the check for unsigned integer overflows and the assumptions to make following an overflow, specified as `'forbid'`, `'allow'`, or `'warn-with-wrap-around'`.

For more information, see `Overflow mode for unsigned integer (-unsigned-integer-overflows)`.

Example: `opts.ChecksAssumption.UnsignedIntegerOverflows = 'allow'`

CodeProverVerification (Affects Code Prover only)

ClassAnalyzer — Classes that you want to verify
`'none'` (default) | `'all'` | `'custom=class1[,class2,...]'`

This property affects Code Prover analysis only.

Classes that you want to verify, specified as `'all'`, `'none'`, or as a character array beginning with `custom=` followed by a comma-separated list of class names.

For more information, see `Class (-class-analyzer)`.

Example: `opts.CodeProverVerification.ClassAnalyzer = 'none'`

FunctionsCalledAfterLoop — Functions that the generated main must call after the cyclic code loop
cell array of function names

This property affects Code Prover analysis only.

Functions that the generated main must call after the cyclic code loop, specified as a cell array of function names.

For more information, see `Termination functions (-functions-called-after-loop)`.

Example: `opts.CodeProverVerification.FunctionsCalledAfterLoop = {'func1','func2'}`

FunctionsCalledBeforeLoop — Functions that the generated main must call before the cyclic code loop
cell array of function names

This property affects Code Prover analysis only.

Model Link only. Functions that the generated main must call before the cyclic code loop, specified as a cell array of function names.

For more information, see `Initialization functions (-functions-called-before-loop)`.

Example: `opts.CodeProverVerification.FunctionsCalledBeforeLoop = {'func1','func2'}`

FunctionsCalledInLoop — Functions that the generated main must call in the cyclic code loop
`'none'` (default) | `'all'` | `'custom=function1[,function2,...]'`

This property affects Code Prover analysis only.

Functions that the generated main must call in the cyclic code loop, specified as `'none'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of function names..

For more information, see `Step functions (-functions-called-in-loop)`.

Example: `opts.CodeProverVerification.FunctionsCalledInLoop = 'all'`

MainGenerator — Generate a main function if it is not present in source files
true (default) | false

This property affects Code Prover analysis only.

Generate a main function if it is not present in source files, specified as true or false.

For more information, see `Verify module or library (-main-generator)`.

Example: `opts.CodeProverVerification.MainGenerator = false`

VariablesWrittenBeforeLoop — Variables that the generated main must initialize before the cyclic code loop
`'none'` (default) | `'all'` | `'custom=variable1[,variable2,...]'`

This property affects Code Prover analysis only.

Variables that the generated main must initialize before the cyclic code loop, specified as `'none'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of variable names.

For more information, see `Parameters (-variables-written-before-loop)`.

Example: `opts.CodeProverVerification.VariablesWrittenBeforeLoop = 'all'`

VariablesWrittenInLoop — Variables that the generated main must initialize in the cyclic code loop
`'none'` (default) | `'all'` | `'custom=variable1[,variable2,...]'`

This property affects Code Prover analysis only.

Variables that the generated main must initialize in the cyclic code loop, specified as `'none'`, `'all'`, or as a character array beginning with `custom=` followed by a comma-separated list of variable names.

For more information, see `Inputs (-variables-written-in-loop)`.

Example: `opts.CodeProverVerification.VariablesWrittenInLoop = 'all'`

CodingRulesCodeMetrics

AcAgcSubset — Subset of MISRA AC AGC rules to check

`'OBL-rules'` (default) | `'OBL-REC-rules'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | polyspace.CodingRulesOptions object | `'from-file'`

Subset of MISRA AC AGC rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see Check MISRA AC AGC (`-misra-ac-agc`).

- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA AC AGC rules, also set `EnableAcAgc` to true.

Example: `opts.CodingRulesCodeMetrics.AcAgcSubset = 'all-rules'`

Data Types: `char`

AllowedPragmas — Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied

cell array of character vectors

Pragma directives for which MISRA C:2004 rule 3.4 or MISRA C++ 16-6-1 must not be applied, specified as a cell array of character vectors. This property affects only MISRA C:2004 or MISRA AC AGC rule checking.

For more information, see Allowed pragmas (`-allowed-pragmas`).

Example: `opts.CodingRulesCodeMetrics.AllowedPragmas = {'pragma_01','pragma_02'}`

Data Types: `cell`

AutosarCpp14 — Set of AUTOSAR C++ 14 rules to check

`'all'` (default) | `'required'` | `'automated'` | polyspace.CodingRulesOptions object | `'from-file'`

This property affects Bug Finder only.

Set of AUTOSAR C++ 14 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see Check AUTOSAR C++ 14 security checks (`-autosar-cpp14`).

- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check AUTOSAR C++ 14 rules, also set `EnableAutosarCpp14` to true.

Example: `opts.CodingRulesCodeMetrics.AutosarCpp14 = 'all'`

Data Types: `char`

BooleanTypes — Data types the coding rule checker must treat as effectively Boolean
cell array of character vectors

Data types that the coding rule checker must treat as effectively Boolean, specified as a cell array of character vectors.

For more information, see `Effective boolean types (-boolean-types)`.

Example: `opts.CodingRulesCodeMetrics.BooleanTypes = {'boolean1_t','boolean2_t'}`

Data Types: `cell`

CertC — Set of CERT C rules and recommendations to check
`'all'` (default) | `'publish-2016'` | `'all-rules'` | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of CERT C rules and recommendations to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C security checks (-cert-c)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `from-file` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C rules and recommendations, also set `EnableCertC` to true.

Example: `opts.CodingRulesCodeMetrics.CertC = 'all'`

Data Types: `char`

CertCpp — Set of CERT C++ rules to check
`'all'` (default) | `polyspace.CodingRulesOptions` object | `'from-file'`

This property affects Bug Finder only.

Set of CERT C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check CERT-C++ security checks (-cert-cpp)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check CERT C++ rules, also set `EnableCertCpp` to true.

Example: `opts.CodingRulesCodeMetrics.CertCpp = 'all'`

Data Types: `char`

CheckersSelectionByFile — File that defines custom set of coding standard checkers
full file path of `.xml` file

File where you define a custom set of coding standards checkers to check, specified as a `.xml` file. You can, in the same file, define a custom set of checkers for each of the coding standards that Polyspace supports. To create a file that defines a custom selection of coding standard checkers, in the Polyspace interface, select a coding standard on the **Coding Standards & Code Metrics** node of the **Configuration** pane and click **Edit**.

For more information, see `Set checkers by file (-checkers-selection-file)`.

Example: `opts.CodingRulesCodeMetrics.CheckersSelectionByFile = 'C:\ps_settings\coding_rules\custom_rules.xml'`

Data Types: `char`

CodeMetrics — Activate code metric calculations
false (default) | true

Activate code metric calculations, specified as true or false. If this property is turned off, Polyspace does not calculate code metrics even if you upload your results to Polyspace Metrics.

For more information about the code metrics, see `Calculate code metrics (-code-metrics)`.

If you assign a coding rules options object to this property, an XML file gets created automatically with the rules specified.

Example: `opts.CodingRulesCodeMetrics.CodeMetrics = true`

EnableAcAgc — Check MISRA AC AGC rules
false (default) | true

Check MISRA AC AGC rules, specified as true or false. To customize which rules are checked, use `AcAgcSubset`.

For more information about the MISRA AC AGC checker, see `Check MISRA AC AGC (-misra-ac-agc)`.

Example: `opts.CodingRulesCodeMetrics.EnableAcAgc = true;`

EnableAutosarCpp14 — Check AUTOSAR C++ 14 rules
false (default) | true

This property affects Bug Finder only.

Check AUTOSAR C++ 14 rules, specified as true or false. To customize which rules are checked, use `AutosarCpp14`.

For more information about the AUTOSAR C++ 14 checker, see `Check AUTOSAR C++ 14 security checks (-autosar-cpp14)`.

Example: `opts.CodingRulesCodeMetrics.EnableAutosarCpp14 = true;`

EnableCertC — check CERT C rules and recommendations
false (default) | true

This property affects Bug Finder only.

Check CERT C rules and recommendations, specified as true or false. To customize which rules are checked, use `CertC`.

For more information about the CERT C checker, see `Check CERT-C security checks (-cert-c)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertC = true;`

EnableCertCpp — check CERT C++ rules
false (default) | true

This property affects Bug Finder only.

Check CERT C++ rules, specified as true or false. To customize which rules are checked, use `CertCpp`.

For more information about the CERT C++ checker, see `Check CERT-C++ security checks (-cert-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableCertCpp = true;`

EnableCheckersSelectionByFile — Check custom set of coding standard checkers
false (default) | true

Check custom set of coding standard checkers, specified as true or false. Use with `CheckersSelectionByFile` and these coding standards:

- `opts.CodingRulesCodeMetrics.AutosarCpp14='from-file'`
- `opts.CodingRulesCodeMetrics.CertC='from-file'`
- `opts.CodingRulesCodeMetrics.CertCpp='from-file'`
- `opts.CodingRulesCodeMetrics.Iso17961='from-file'`
- `opts.CodingRulesCodeMetrics.JsfSubset='from-file'`

- `opts.CodingRulesCodeMetrics.MisraC3Subset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCSubset='from-file'`
- `opts.CodingRulesCodeMetrics.MisraCppSubset='from-file'`

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCheckersSelectionByFile = true;`

EnableCustomRules — Check custom coding rules
false (default) | true

Check custom coding rules, specified as true or false. The file you specify with `CheckersSelectionByFile` defines the custom coding rules.

Use with `EnableCheckersSelectionByFile`.

For more information, see `Check custom rules (-custom-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableCustomRules = true;`

EnableIso17961 — check ISO-17961 rules
false (default) | true

This property affects Bug Finder only.

Check ISO/IEC TS 17961 rules, specified as true or false. To customize which rules are checked, use `Iso17961`.

For more information about the ISO-17961 checker, see `Check ISO-17961 security checks (-iso-17961)`.

Example: `opts.CodingRulesCodeMetrics.EnableIso17961 = true;`

EnableJsf — Check JSF C++ rules
false (default) | true

Check JSF C++ rules, specified as true or false. To customize which rules are checked, use `JsfSubset`.

For more information, see `Check JSF C++ rules (-jsf-coding-rules)`.

Example: `opts.CodingRulesCodeMetrics.EnableJsf = true;`

EnableMisraC — Check MISRA C:2004 rules
false (default) | true

Check MISRA C:2004 rules, specified as true or false. To customize which rules are checked, use `MisraCSubset`.

For more information, see `Check MISRA C:2004 (-misra2)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC = true;`

EnableMisraC3 — Check MISRA C:2012 rules
false (default) | true

Check MISRA C:2012 rules, specified as true or false. To customize which rules are checked, use `MisraC3Subset`.

For more information about the MISRA C:2012 checker, see `Check MISRA C:2012 (-misra3)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraC3 = true;`

EnableMisraCpp — Check MISRA C++:2008 rules
false (default) | true

Check MISRA C++:2008 rules, specified as true or false. To customize which rules are checked, use `MisraCppSubset`.

For more information about the MISRA C++:2008 checker, see `Check MISRA C++ rules (-misra-cpp)`.

Example: `opts.CodingRulesCodeMetrics.EnableMisraCpp = true;`

Iso17961 — Set of ISO-17961 rules to check
'all' (default) | 'decidable' | polyspace.CodingRulesOptions object | 'from-file'

This property affects Bug Finder only.

Set of ISO/IEC TS 17961 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check ISO-17961 security checks (-iso-17961)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check ISO/IEC TS 17961 rules, also set `EnableIso17961` to true.

Example: `opts.CodingRulesCodeMetrics.Iso17961 = 'all'`

Data Types: `char`

JsfSubset — Subset of JSF C++ rules to check
'shall-rules' (default) | 'shall-will-rules' | 'all-rules' | polyspace.CodingRulesOptions object | 'from-file'

Subset of JSF C++ rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check JSF C++ rules (-jsf-coding-rules)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check JSF C++ rules, set `EnableJsf` to true.

Example: `opts.CodingRulesCodeMetrics.JsfSubset = 'all-rules'`

Data Types: `char`

Misra3AgcMode — Use the MISRA C:2012 categories for automatically generated code
false (default) | true

Use the MISRA C:2012 categories for automatically generated code, specified as true or false.

For more information, see `Use generated code requirements (-misra3-agc-mode)`.

Example: `opts.CodingRulesCodeMetrics.Misra3AgcMode = true;`

MisraC3Subset — Subset of MISRA C:2012 rules to check
`'mandatory-required'` (default) | `'mandatory'` | `'single-unit-rules'` | `'system-decidable-rules'` | `'all'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2012 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2012 (-misra3)`.
- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.
- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2012 rules, also set `EnableMisraC3` to true.

Example: `opts.CodingRulesCodeMetrics.MisraC3Subset = 'all'`

Data Types: `char`

MisraCSubset — Subset of MISRA C:2004 rules to check
`'required-rules'` (default) | `'single-unit-rules'` | `'system-decidable-rules'` | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C:2004 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C:2004 (-misra2)`.

- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C:2004 rules, also set `EnableMisraC` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCSubset = 'all-rules'`

Data Types: `char`

MisraCppSubset — Subset of MISRA C++ rules
`'required-rules'` (default) | `'all-rules'` | `'SQO-subset1'` | `'SQO-subset2'` | `polyspace.CodingRulesOptions` object | `'from-file'`

Subset of MISRA C++:2008 rules to check, specified by:

- Character vector of one of the subset names. For more information about the different subsets, see `Check MISRA C++ rules (-misra-cpp)`.

- A coding rules options object. To create a coding rules options object, see `polyspace.CodingRulesOptions`.

- An XML file specifying coding standard checkers. Use `'from-file'` for this property and then use the `EnableCheckersSelectionByFile` and `CheckersSelectionByFile` property to specify the full path to the file where you define a custom subset of checkers.

  You can create this file manually or in the Polyspace interface. See "Check for Coding Standard Violations". If you assign a coding rules options object to this property, an XML file is created automatically and assigned to the `CheckersSelectionByFile` property. The XML file enables rules extracted from the coding rules options object.

To check MISRA C++ rules, set `EnableMisraCpp` to true.

Example: `opts.CodingRulesCodeMetrics.MisraCppSubset = 'all-rules'`

Data Types: `char`

EnvironmentSettings

Dos — Consider that file paths are in MS-DOS style
true (default) | false

Consider that file paths are in MS-DOS style, specified as true or false.

For more information, see `Code from DOS or Windows file system (-dos)`.

Example: `opts.EnvironmentSettings.Dos = true;`

IncludeFolders — Include folders needed for compilation
cell array of include folder paths

Include folders needed for compilation, specified as a cell array of the include folder paths.

To specify all subfolders of a folder, use folder path followed by **, for instance, `'C:\includes\**'`. The notation follows the syntax of the `dir` function. See also "Specify Multiple Source Files".

For more information, see `-I`.

Example: `opts.EnvironmentSettings.IncludeFolders = {'/includes','/com1/inc'};`

Example: `opts.EnvironmentSettings.IncludeFolders = {'C:\project1\common\includes'};`

Data Types: `cell`

Includes — Files to be #include-ed by each C file
cell array of files

Files to be `#include`-ed by each C source file in the analysis, specified by a cell array of files.

For more information, see `Include (-include)`.

Example: `opts.EnvironmentSettings.Includes = {'/inc/inc_file.h','/inc/inc_math.h'}`

NoExternC — Ignore linking errors inside extern blocks
false (default) | true

Ignore linking errors inside extern blocks, specified as true or false.

For more information, see `Ignore link errors (-no-extern-c)`.

Example: `opts.EnvironmentSettings.NoExternC = false;`

PostPreProcessingCommand — Command or script to run on source files after preprocessing
character vector

Command or script to run on source files after preprocessing, specified as a character vector of the command to run.

For more information, see `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Example: Linux — `opts.EnvironmentSettings.PostPreProcessingCommand = [pwd,'/replace_keyword.pl']`

Example: Windows — `opts.EnvironmentSettings.PostPreProcessingCommand = '"C:\Program Files\MATLAB\R2015b\sys\perl\win32\bin\perl.exe" "C:\My_Scripts\replace_keyword.pl"'`

StopWithCompileError — Stop analysis if a file does not compile
false (default) | true

Stop analysis if a file does not compile, specified as true or false.

For more information, see `Stop analysis if a file does not compile (-stop-if-compile-error)`.

Example: `opts.EnvironmentSettings.StopWithCompileError = true;`

InputsStubbing

DataRangeSpecifications — Constrain global variables, function inputs, and return values of stubbed functions
file path

Constrain global variables, function inputs, and return values of stubbed functions specified by the path to an XML constraint file. For more information about the constraint file, see "Specify External Constraints".

For more information about this option, see `Constraint setup (-data-range-specifications)`.

Example: `opts.InputsStubbing.DataRangeSpecifications = 'C:\project\constraint_file.xml'`

DoNotGenerateResultsFor — Files on which you do not want analysis results
`'include-folders'` (default) | `'all-headers'` | `'custom=folder1[,folder2,...]'`

Files on which you do not want analysis results, specified by `'include-folders'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.GenerateResultsFor`. For more information, see `Do not generate results for (-do-not-generate-results-for)`.

Example: `opts.InputsStubbing.DoNotGenerateResultsFor = 'custom=C:\project\file1.c,C:\project\file2.c'`

GenerateResultsFor — Files on which you want analysis results
`'source-headers'` (default) | `'all-headers'` | `'custom=folder1[,folder2,...]'`

Files on which you want analysis results, specified by `'source-headers'`, `'all-headers'`, or a character array beginning with `custom=` followed by a comma-separated list of file or folder names.

Use this option with `InputsStubbing.DoNotGenerateResultsFor`. For more information, see `Generate results for sources and (-generate-results-for)`.

Example: `opts.InputsStubbing.GenerateResultsFor = 'custom=C:\project\includes_common_1,C:\project\includes_common_2'`

FunctionsToStub — Functions to stub during analysis
cell array of function names

This property affects Code Prover analysis only.

Functions to stub during analysis, specified as a cell array of function names.

For more information, see `Functions to stub (-functions-to-stub)`.

Example: `opts.InputsStubbing.FunctionsToStub = {'func1', 'func2'}`

NoDefInitGlob — Consider global variables as uninitialized
false (default) | true

This property affects Code Prover analysis only.

Consider global variables as uninitialized, specified as true or false.

For more information, see `Ignore default initialization of global variables (-no-def-init-glob)`.

Example: `opts.InputsStubbing.NoDefInitGlob = true`

NoStlStubs — Do not use Polyspace implementations of functions in the Standard Template Library
false (default) | true

This property applies only to a Code Prover analysis of C++ code.

Do not use Polyspace implementations of functions in the Standard Template Library, specified as true or false.

For more information, see `No STL stubs (-no-stl-stubs)`.

Example: `opts.InputsStubbing.NoStlStubs = true`

StubECoderLookupTables — Specify that the analysis must stub functions in the generated code that use lookup tables
true (default) | false

This property applies only to a Code Prover analysis of code generated from models.

Specify that the analysis must stub functions in the generated code that use lookup tables. By replacing the functions with stubs, the analysis assumes more precise return values for the functions.

For more information, see `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`.

Example: `opts.InputsStubbing.StubECoderLookupTables = true`

Macros

DefinedMacros — Macros to be replaced
cell array of macros

In preprocessed code, macros are replaced by the definition, specified in a cell array of macros and definitions. Specify the macro as `Macro=Value`. If you want Polyspace to ignore the macro, leave the `Value` blank. A macro with no equal sign replaces all instances of that macro by 1.

For more information, see `Preprocessor definitions (-D)`.

Example: `opts.Macros.DefinedMacros = {'uint32=int','name3=','var'}`

UndefinedMacros — Macros to undefine
cell array of macros

In preprocessed code, macros are undefined, specified by a cell array of macros to undefine.

For more information, see `Disabled preprocessor definitions (-U)`.

Example: `opts.Macros.DefinedMacros = {'name1','name2'}`

MergedComputingSettings

AddToResultsRepositoryBugFinder — Upload Bug Finder results to Polyspace Metrics web dashboard
false (default) | true

This property affects Bug Finder analysis only.

Upload Bug Finder analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryBugFinder = true;`

AddToResultsRepositoryCodeProver — Upload Code Prover results to Polyspace Metrics web dashboard
false (default) | true

This property affects Code Prover analysis only.

Upload Code Prover analysis results to Polyspace Metrics web dashboard, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Upload results to Polyspace Metrics (-add-to-results-repository)`.

Example: `opts.MergedComputingSettings.AddToResultsRepositoryCodeProver = true;`

BatchBugFinder — Send Bug Finder analysis to remote server
false (default) | true

This property affects Bug Finder analysis only.

Send Bug Finder analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchBugFinder = true;`

BatchCodeProver — Send Code Prover analysis to remote server
false (default) | true

This property affects Code Prover analysis only.

Send Code Prover analysis to remote server, specified as true or false. To use this option, in your Polyspace preferences, you must specify a metrics server.

For more information, see `Run Bug Finder or Code Prover analysis on a remote cluster (-batch)`.

Example: `opts.MergedComputingSettings.BatchCodeProver = true;`

FastAnalysis — Run Bug Finder analysis using faster local mode
false (default) | true

This property affects Bug Finder analysis only.

Use fast analysis mode for Bug Finder analysis, specified as true or false.

For more information, see `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Example: `opts.MergedComputingSettings.FastAnalysis = true;`

MergedReporting

EnableReportGeneration — Generate a report after the analysis
false (default) | true

After the analysis, generate a report, specified as true or false.

For more information, see `Generate report`.

Example: `opts.MergedReporting.EnableReportGeneration = true`

ReportOutputFormat — Output format of generated report
`'Word'` (default) | `'HTML'` | `'PDF'`

Output format of generated report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Output format (-report-output-format)`.

Example: `opts.MergedReporting.ReportOutputFormat = 'PDF'`

BugFinderReportTemplate — Template for generating Bug Finder analysis report
`'BugFinderSummary'` (default) | `'BugFinder'` | `'SecurityCWE'` | `'CodeMetrics'` | `'CodingStandards'`

This property affects a Bug Finder analysis only.

Template for generating analysis report, specified as one of the report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.BugFinderReportTemplate = 'CodeMetrics'`

CodeProverReportTemplate — Template for generating Code Prover analysis report
`'Developer'` (default) | `'CallHierarchy'` | `'CodeMetrics'` | `'CodingStandards'` | `'DeveloperReview'` | `'Developer_withGreenChecks'` | `'Quality'` | `'VariableAccess'`

This property affects a Code Prover analysis only.

Template for generating analysis report, specified as one of the predefined report formats. To activate this option, specify `Reporting.EnableReportGeneration`.

For more information about the different values, see `Bug Finder and Code Prover report (-report-template)`.

Example: `opts.MergedReporting.CodeProverReportTemplate = 'CodeMetrics'`

Multitasking

`ArxmlMultitasking` — Specify path of ARXML files to parse for multitasking configuration
cell array of file paths

Specify the path to the ARXML files the software parses to set up your multitasking configuration.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `autosar`.

For more information, see `ARXML files selection (-autosar-multitasking)`

Example: `opts.Multitasking.ArxmlMultitasking={'C:\Polyspace_Workspace\AUTOSAR\myFile.arxml'}`

`CriticalSectionBegin` — Functions that begin critical sections
cell array of critical section function names

Functions that begin critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionEnd`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionBegin = {'function1:cs1','function2:cs2'}`

`CriticalSectionEnd` — Functions that end critical sections
cell array of critical section function names

Functions that end critical sections specified as a cell array of critical section function names. To activate this option, specify `Multitasking.EnableMultitasking` and `Multitasking.CriticalSectionBegin`.

For more information, see `Critical section details (-critical-section-begin -critical-section-end)`.

Example: `opts.Multitasking.CriticalSectionEnd = {'function1:cs1','function2:cs2'}`

`CyclicTasks` — Specify functions that represent cyclic tasks
cell array of function names

Specify functions that represent cyclic tasks.

To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Cyclic tasks (-cyclic-tasks)`.

Example: `opts.Multitasking.CyclicTasks = {'function1','function2'}`

`EnableConcurrencyDetection` — Enable automatic detection of certain families of threading functions
false (default) | true

This property affects Code Prover analysis only.

Enable automatic detection of certain families of threading functions, specified as true or false.

For more information, see `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

Example: `opts.Multitasking.EnableConcurrencyDetection = true`

EnableExternalMultitasking — Enable automatic multitasking configuration from external file definitions
false (default) | true

Enable multitasking configuration of your projects from external files you provide. Configure multitasking from ARXML files for an AUTOSAR project, or from OIL files for an OSEK project.

Activate this option to enable `Multitasking.ArxmlMultitasking` or `Multitasking.OsekMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.EnableExternalMultitasking = 1`

EnableMultitasking — Configure multitasking manually
false (default) | true

Configure multitasking manually by specifying `true`. This property activates the other manual, multitasking properties.

For more information, see `Configure multitasking manually`.

Example: `opts.Multitasking.EnableMultitasking = 1`

EntryPoints — Functions that serve as entry-points to your multitasking application
cell array of entry-point function names

Functions that serve as entry-points to your multitasking application specified as a cell array of entry-point function names. To activate this option, also specify `Multitasking.EnableMultitasking`.

For more information, see `Tasks (-entry-points)`.

Example: `opts.Multitasking.EntryPoints = {'function1','function2'}`

ExternalMultitaskingType — Specify type of file to parse for multitasking configuration
'osek' (default) | 'autosar'

Specify the type of file the software parses to set up your multitasking configuration:

- For `osek` type, the analysis looks for OIL files in the file or folder paths that you specify.
- For `autosar` type, the analysis looks for ARXML files in the file paths that you specify.

To activate this option, specify `Multitasking.EnableExternalMultitasking`.

For more information, see `OIL files selection (-osek-multitasking)` and `ARXML files selection (-autosar-multitasking)`.

Example: `opts.Multitasking.ExternalMultitaskingType = 'autosar'`

`Interrupts` — Specify functions that represent nonpreemptable interrupts
cell array of function names

Specify functions that represent nonpreemptable interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Interrupts (-interrupts)`.

Example: `opts.Multitasking.Interrupts = {'function1','function2'}`

`InterruptsDisableAll` — Specify routine that disable interrupts
cell array with one function name

This property affects Bug Finder analysis only.

Specify function that disables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsDisableAll = {'function'}`

`InterruptsEnableAll` — Specify routine that reenable interrupts
cell array with one function name

This property affects Bug Finder analysis only.

Specify function that reenables all interrupts.

To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

Example: `opts.Multitasking.InterruptsEnableAll = {'function'}`

`OsekMultitasking` — Specify path of OIL files to parse for multitasking configuration
`'auto'` (default) | `'custom=folder1[,folder2,...]'`

Specify the path to the OIL files the software parses to set up your multitasking configuration:

- In the mode specified with `'auto'`, the analysis uses OIL files in your project source and include folders, but not their subfolders.
- In the mode specified with `'custom=folder1[,folder2,...]'`, the analysis uses the OIL files at the specified path, and the path subfolders.

To activate this option, specify `Multitasking.EnableExternalMultitasking` and set `Multitasking.ExternalMultitaskingType` to `osek`.

For more information, see `OIL files selection (-osek-multitasking)`

Example: `opts.Multitasking.OsekMultitasking = 'custom=file_path, dir_path'`

`TemporalExclusion` — Entry-point functions that cannot execute concurrently
cell array of entry-point function names

Entry-point functions that cannot execute concurrently specified as a cell array of entry-point function names. Each set of exclusive tasks is one cell array entry with functions separated by spaces. To activate this option, specify `Multitasking.EnableMultitasking`.

For more information, see `Temporally exclusive tasks (-temporal-exclusions-file)`.

Example: `opts.Multitasking.TemporalExclusion = {'function1 function2', 'function3 function4 function5'}` where function1 and function2 are temporally exclusive, and function3, function4, and function 5 are temporally exclusive.

Precision (Affects Code Prover Only)

ContextSensitivity — Store call context information to identify function call that caused errors
`'none'` (default) | `'auto'` | `'custom=function1[,function2,...]'`

This property affects Code Prover analysis only.

Store call context information to identify a function call that caused errors, specified as `none`, `auto`, or as a character array beginning with `custom=` followed by a list of comma-separated function names.

For more information, see `Sensitivity context (-context-sensitivity)`.

Example: `opts.Precision.ContextSensitivity = 'auto'`

Example: `opts.Precision.ContextSensitivity = 'custom=func1'`

ModulesPrecision — Source files you want to verify at higher precision
cell array of file names and precision levels

This property affects Code Prover analysis only.

Source files that you want to verify at higher precision, specified as a cell array of file names without the extension and precision levels using this syntax: *filename*:O*level*

For more information, see `Specific precision (-modules-precision)`.

Example: `opts.Precision.ModulesPrecision = {'file1:O0', 'file2:O3'}`

OLevel — Precision level for the verification
2 (default) | 0 | 1 | 3

This property affects Code Prover analysis only.

Precision level for the verification, specified as 0, 1, 2, or 3.

For more information, see `Precision level (-O)`.

Example: `opts.Precision.OLevel = 3`

PathSensitivityDelta — Avoid certain verification approximations for code with fewer lines
positive integer

This property affects Code Prover analysis only.

Avoid certain verification approximations for code with fewer lines, specified as a positive integer representing how sensitive the analysis is. Higher values can increase verification time exponentially.

For more information, see `Improve precision of interprocedural analysis (-path-sensitivity-delta)`.

Example: `opts.Precision.PathSensitivityDelta = 2`

Timeout — Time limit on your verification
character vector

This property affects Code Prover analysis only.

Time limit on your verification, specified as a character vector of time in hours.

For more information, see `Verification time limit (-timeout)`.

Example: `opts.Precision.Timeout = '5.75'`

To — Number of times the verification process runs
`'Software Safety Analysis level 2'` (default) | `'Software Safety Analysis level 0'` | `'Software Safety Analysis level 1'` | `'Software Safety Analysis level 3'` | `'Software Safety Analysis level 4'` | `'Source Compliance Checking'` | `'other'`

This property affects Code Prover analysis only.

Number of times the verification process runs, specified as one of the preset analysis levels.

For more information, see `Verification level (-to)`.

Example: `opts.Precision.To = 'Software Safety Analysis level 3'`

Scaling (Affects Code Prover Only)

Inline — Functions on which separate results must be generated for each function call
cell array of function names

This property affects Code Prover analysis only.

Functions on which separate results must be generated for each function call, specified as a cell array of function names.

For more information, see `Inline (-inline)`.

Example: `opts.Scaling.Inline = {'func1','func2'}`

KLimiting — Limit depth of analysis for nested structures
positive integer

This property affects Code Prover analysis only.

Limit depth of analysis for nested structures, specified as a positive integer indicating how many levels into a nested structure to verify.

For more information, see `Depth of verification inside structures (-k-limiting)`.

Example: `opts.Scaling.KLimiting = 3`

TargetCompiler

Compiler — Compiler that builds your source code
'generic' (default) | 'gnu3.4' | 'gnu4.6' | 'gnu4.7' | 'gnu4.8' | 'gnu4.9' | 'gnu5.x' |
'gnu6.x' | 'gnu7.x' | 'clang3.x' | 'clang4.x' | 'clang5.x' | 'visual9.0' | 'visual10' |
'visual11.0' | 'visual12.0' | 'visual14.0' | 'visual15.x' | 'keil' | 'iar' | 'armcc' |
'armclang' | 'codewarrior' | 'diab' | 'greenhills' | 'iar-ew' | 'renesas' | 'tasking' |
'ti'

Compiler that builds your source code.

For more information, see Compiler (-compiler).

Example: opts.TargetCompiler.Compiler = 'Visual11.0'

CppVersion — Specify C++ standard version followed in code
'defined-by-compiler' (default) | 'cpp03' | 'cpp11' | 'cpp14' | 'cpp17'

Specify C++ standard version followed in code, specified as a character vector.

For more information, see C++ standard version (-cpp-version).

Example: opts.TargetCompiler.CppVersion = 'cpp11';

CVersion — Specify C standard version followed in code
'defined-by-compiler' (default) | 'c90' | 'c99' | 'c11'

Specify C standard version followed in code, specified as a character vector.

For more information, see C standard version (-c-version).

Example: opts.TargetCompiler.CVersion = 'c90';

DivRoundDown — Round down quotients from division or modulus of negative numbers
false (default) | true

Round down quotients from division or modulus of negative numbers, specified as true or false.

For more information, see Division round down (-div-round-down).

Example: opts.TargetCompiler.DivRoundDown = true

EnumTypeDefinition — Base type representation of enum
'defined-by-compiler' (default) | 'auto-signed-first' | 'auto-unsigned-first'

Base type representation of enum, specified by an allowed base-type set. For more information about
the different values, see Enum type definition (-enum-type-definition).

Example: opts.TargetCompiler.EnumTypeDefinition = 'auto-unsigned-first'

IgnorePragmaPack — Ignore #pragma pack directives
false (default) | true

Ignore #pragma pack directives, specified as true or false.

For more information, see Ignore pragma pack directives (-ignore-pragma-pack).

Example: opts.TargetCompiler.IgnorePragmaPack = true

Language — Language of analysis
'C-CPP' (default) | 'C' | 'CPP'

This property is read-only.

Language of the analysis, specified during the object construction. This value changes which properties appear.

For more information, see Source code language (-lang).

LogicalSignedRightShift — Treatment of signed bit on signed variables
'Arithmetical' (default) | 'Logical'

Treatment of signed bit on signed variables, specified as Arithmetical or Logical. For more information, see Signed right shift (-logical-signed-right-shift).

Example: opts.TargetCompiler.LogicalSignedRightShift = 'Logical'

NoUliterals — Do not use predefined typedefs for char16_t or char32_t
false (default) | true

Do not use predefined typedefs for char16_t or char32_t, specified as true or false. For more information, see Block char16/32_t types (-no-uliterals).

Example: opts.TargetCompiler.NoUliterals = true

PackAlignmentValue — Default structure packing alignment
'defined-by-compiler' (default) | '1' | '2' | '4' | '8' | '16'

Default structure packing alignment, specified as 'defined-by-compiler', '1', '2', '4', '8', or '16'. This property is available only for Visual C++ code.

For more information, see Pack alignment value (-pack-alignment-value).

Example: opts.TargetCompiler.PackAlignmentValue = '4'

SfrTypes — sfr types
cell array of sfr keywords

sfr types, specified as a cell array of sfr keywords using the syntax *sfr_name=size_in_bits*. For more information, see Sfr type support (-sfr-types).

This option only applies when you set TargetCompiler.Compiler to keil or iar.

Example: opts.TargetCompiler.SfrTypes = {'sfr32=32'}

SizeTTypeIs — Underlying type of size_t
'defined-by-compiler' (default) | 'unsigned-int' | 'unsigned-long' | 'unsigned-long-long'

Underlying type of size_t, specified as 'defined-by-compiler', 'unsigned-int', 'unsigned-long', or 'unsigned-long-long'. See Management of size_t (-size-t-type-is).

Example: opts.TargetCompiler.SizeTTypeIs = 'unsigned-long'

Target — Target processor
'i386' (default) | 'arm' | 'arm64' | 'avr' | 'c-167' | 'c166' | 'c18' | 'c28x' | 'c6000' | 'coldfire' | 'hc08' | 'hc12' | 'm68k' | 'mcore' | 'mips' | 'mpc5xx' | 'msp430' | 'necv850'

| 'powerpc' | 'powerpc64' | 'rh850' | 'rl78' | 'rx' | 's12z' | 'sharc21x61' | 'sparc' | 'superh' | 'tms320c3x' | 'tricore' | 'x86_64' | generic target object

Set size of data types and endianness of processor, specified as one of the predefined target processors or a generic target object.

For more information about the predefined processors, see `Target processor type (-target)`.

For more information about creating a generic target, see `polyspace.GenericTargetOptions`.

Example: `opts.TargetCompiler.Target = 'hc12'`

WcharTTypeIs — Underlying type of `wchar_t`
`'defined-by-compiler'` (default) | `'signed-short'` | `'unsigned-short'` | `'signed-int'` | `'unsigned-int'` | `'signed-long'` | `'unsigned-long'`

Underlying type of `wchar_t`, specified as `'defined-by-compiler'`, `'signed-short'`, `'unsigned-short'`, `'signed-int'`, `'unsigned-int'`, `'signed-long'`, or `'unsigned-long'`. See `Management of wchar_t (-wchar-t-type-is)`.

Example: `opts.TargetCompiler.WcharTTypeIs = 'unsigned-int'`

VerificationAssumption (Affects Code Prover Only)

ConsiderVolatileQualifierOnFields — Assume that volatile qualified structure fields can have all possible values at any point in code
false (default) | true

This property affects Code Prover analysis only.

Assume that volatile qualified structure fields can have all possible values at any point in code.

For more information, see `Consider volatile qualifier on fields (-consider-volatile-qualifier-on-fields)`.

Example: `opts.VerificationAssumption.ConsiderVolatileQualifierOnFields = true`

ConstraintPointersMayBeNull — Specify that environment pointers can be NULL unless constrained otherwise
false (default) | true

This property affects Code Prover analysis only.

Specify that environment pointers can be NULL unless constrained otherwise.

For more information, see `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.

Example: `opts.VerificationAssumption.ConstraintPointersMayBeNull = true`

FloatRoundingMode — Rounding modes to consider when determining the results of floating-point arithmetic
`to-nearest` (default) | all

This property affects Code Prover analysis only.

Rounding modes to consider when determining the results of floating-point arithmetic, specified as `to-nearest` or `all`.

For more information, see `Float rounding mode (-float-rounding-mode)`.

Example: `opts.VerificationAssumption.FloatRoundingMode = 'all'`

RespectTypesInFields — Do not cast nonpointer fields of a structure to pointers
false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer fields of a structure to pointers, specified as true or false.

For more information, see `Respect types in fields (-respect-types-in-fields)`.

Example: `opts.VerificationAssumption.RespectTypesInFields = true`

RespectTypesInGlobals — Do not cast nonpointer global variables to pointers
false (default) | true

This property affects Code Prover analysis only.

Do not cast nonpointer global variables to pointers, specified as true or false.

For more information, see `Respect types in global variables (-respect-types-in-globals)`.

Example: `opts.VerificationAssumption.RespectTypesInGlobals = true`

Other Properties

Author — Project author
username of current user (default) | character vector

Name of project author, specified as a character vector.

For more information, see `-author`.

Example: `opts.Author = 'JaneDoe'`

ImportComments — Import comments and justifications from previous analysis
character vector

To import comments and justifications from a previous analysis, specify the path to the results folder of the previous analysis.

You can also point to a previous results folder to see only new results compared to the previous run. See "Compare Results from Different Polyspace Runs by Using MATLAB Scripts".

For more information, see `-import-comments`

Example: `opts.ImportComments = fullfile(polyspaceroot,'polyspace','examples','cxx','Bug_Finder_Example','Module_1','BF_Result')`

Prog — Project name
`PolyspaceProject` (default) | character vector

Project name, specified as a character vector.

For more information, see `-prog`.

Example: `opts.Prog = 'myProject'`

ResultsDir — Location to store results
folder path

Location to store results, specified as a folder path. By default, the results are stored in the current folder.

For more information, see `-results-dir`.

You can also create a separate results folder for each new run. See "Compare Results from Different Polyspace Runs by Using MATLAB Scripts".

Example: `opts.ResultsDir = 'C:\project\myproject\results\'`

Sources — Source files
cell array of files

Source files to analyze, specified as a cell array of files.

To specify all files in a folder, use folder path followed by *, for instance, `'C:\src\*'`. To specify all files in a folder and its subfolders, use folder path followed by **, for instance, `'C:\src\**'`. The notation follows the syntax of the `dir` function. See also "Specify Multiple Source Files".

For more information, see `-sources`.

Example: `opts.Sources = {'file1.c', 'file2.c', 'file3.c'}`

Example: `opts.Sources = {'project/src1/file1.c', 'project/src2/file2.c', 'project/src3/file3.c'}`

Version — Project version number
`'1.0'` (default) | character array of a number

Version number of project, specified as a character array of a number. This option is useful if you upload your results to Polyspace Metrics. If you increment version numbers each time that you reanalyze your object, you can compare the results from two versions in Polyspace Metrics.

For more information, see `-v[ersion]`.

Example: `opts.Version = '2.3'`

See Also

Topics
"Analysis Options in Polyspace Code Prover"

Introduced in R2017a

MISRA C 2012

MISRA C:2012 Dir 1.1

Any implementation-defined behavior on which the output of the program depends shall be documented and understood

Description

Directive Definition

Any implementation-defined behavior on which the output of the program depends shall be documented and understood.

Rationale

A code construct has implementation-defined behavior if the C standard allows compilers to choose their own specifications for the construct. The full list of implementation-defined behavior is available in Annex J.3 of the standard ISO/IEC 9899:1999 (C99) and in Annex G.3 of the standard ISO/IEC 9899:1990 (C90).

If you understand and document all implementation-defined behavior, you can be assured that all output of your program is intentional and not produced by chance.

Polyspace Implementation

The analysis detects the following possibilities of implementation-defined behavior in C99 and their counterparts in C90. If you know the behavior of your compiler implementation, justify the analysis result with appropriate comments. To justify a result, assign one of these statuses: `Justified`, `No action planned`, or `Not a defect`.

Tip To mass-justify all results that indicate the same implementation-defined behavior, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps

J.3.2: Environment	An alternative manner in which `main` function may be defined.	The analysis flags `main` with arguments and return types other than:
		
		`int main(void) { ... }`
		
		or
		
		`int main(int argc, char *argv[]) { ... }`
		
		See section 5.1.2.2.1 of the C99 Standard.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
J.3.2: Environment	The set of environment names and the method for altering the environment list used by the `getenv` function.	The analysis flags uses of the `getenv` function. For this function, you need to know the list of environment variables and how the list is modified. See section 7.20.4.5 of the C99 Standard.
J.3.6: Floating Point	The rounding behaviors characterized by non-standard values of `FLT_ROUNDS`.	The analysis flags the include of `float.h` if values of `FLT_ROUNDS` are outside the set, {-1, 0, 1, 2, 3}. Only the values in this set lead to well-defined rounding behavior. See section 5.2.4.2.2 of the C99 Standard.
J.3.6: Floating Point	The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD`.	The analysis flags the include of `float.h` if values of `FLT_EVAL_METHOD` are outside the set, {-1, 0, 1, 2}. Only the values in this set lead to well-defined behavior for floating-point operations. See section 5.2.4.2.2 of the C99 Standard.
J.3.6: Floating Point	The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value.	The analysis flags conversions from integer to floating-point data types of smaller size (for example, 64-bit `int` to 32-bit `float`). See section 6.3.1.4 of the C99 Standard.
J.3.6: Floating Point	The direction of rounding when a floating-point number is converted to a narrower floating-point number.	The analysis flags these conversions: • `double` to `float` • `long double` to `double` or `float` See section 6.3.1.5 of the C99 Standard.

J.3.6: Floating Point	The default state for the `FENV_ACCESS` pragma.	The analysis flags use of the pragma other than:
		
		`#pragma STDC FENV_ACCESS ON`
		
		or
		
		`#pragma STDC FENV_ACCESS OFF`
		
		See section 7.6.1 of the C99 Standard.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps

J.3.6: Floating Point	The default state for the `FP_CONTRACT` pragma.	The analysis flags use of the pragma other than:
		
		`#pragma STDC FP_CONTRACT ON`
		
		or
		
		`#pragma STDC FP_CONTRACT OFF`
		
		See section 7.12.2 of the C99 Standard.

J.3.11: Preprocessing Directives	The behavior on each recognized non-STDC #pragma directive.	The analysis flags the pragma usage: `#pragma` *pp-tokens* where the processing token `STDC` does not immediately follow`pragma`. For instance: `#pragma FENV_ACCESS ON` See section 6.10.6 of the C99 Standard.
J.3.12: Library Functions	Whether the `feraiseexcept` function raises the ''inexact'' floating-point exception in addition to the ''overflow'' or ''underflow'' floating-point exception.	The analysis flags calls to the `feraiseexcept` function. See section 7.6.2.3 of the C99 Standard.
J.3.12: Library Functions	Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function.	The analysis flags calls to the `setlocale` function when its second argument is not `"C"` or `""`. See section 7.11.1.1 of the C99 Standard.
J.3.12: Library Functions	The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2.	The analysis flags the include of `math.h` if `FLT_EVAL_METHOD` has values outside the set {0,1,2}. See section 7.12 of the C99 Standard.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
J.3.12: Library Functions	The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient.	The analysis flags calls to the `remquo`, `remquof` and `remquol` function. See section 7.12.10.3 of the C99 Standard.
J.3.12: Library Functions	The termination status returned to the host environment by the `abort`, `exit`, or `_Exit` function.	The analysis flags calls to the `abort`, `exit`, or `_Exit` function. See sections 7.20.4.1, 7.20.4.3 or 7.20.4.4 of the C99 Standard.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: The implementation
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2017b

MISRA C:2012 Dir 2.1

All source files shall compile without any compilation errors

Description

Directive Definition

All source files shall compile without any compilation errors.

Rationale

A conforming compiler is permitted to produce an object module despite the presence of compilation errors. However, execution of the resulting program can produce unexpected behavior.

Polyspace Implementation

The software raises a violation of this directive if it finds a compilation error. Because Code Prover is more strict about compilation errors compared to Bug Finder, the coding rules checking in the two products can produce different results for this directive.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Compilation and build
Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (-misra3)|MISRA C:2012 Rule 1.1

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2015b

MISRA C:2012 Dir 4.1

Run-time failures shall be minimized

Description

Directive Definition

Run-time failures shall be minimized.

Rationale

Some areas to concentrate on are:

- Arithmetic errors
- Pointer arithmetic
- Array bound errors
- Function parameters
- Pointer dereferencing
- Dynamic memory

Polyspace Implementation

This directive is checked through the Polyspace analysis. For more information, see:

- "Defects".
- "Run-Time Checks".

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Code design
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Dir 4.11|MISRA C:2012 Rule 1.3|MISRA C:2012 Rule 18.1|MISRA C:2012 Rule 18.2|MISRA C:2012 Rule 18.3|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Dir 4.3

Assembly language shall be encapsulated and isolated

Description

Directive Definition

Assembly language shall be encapsulated and isolated.

Rationale

Encapsulating assembly language is beneficial because:

- It improves readability.
- The name, and documentation, of the encapsulating macro or function makes the intent of the assembly language clear.
- All uses of assembly language for a given purpose can share encapsulation, which improves maintainability.
- You can easily substitute the assembly language for a different target or for purposes of static analysis.

Polyspace Implementation

Polyspace does not raise a warning on assembly language code encapsulated in the following:

- `asm` functions or `asm` pragmas
- Macros

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Assembly Language Code in C Function

```
enum boolVal {TRUE, FALSE};
enum boolVal isTaskActive;
void taskHandler(void);

void taskHandler(void) {
    isTaskActive = FALSE;
    // Software interrupt for task switching
    asm volatile   /* Non-compliant */
    (
        "SWI &02"     /* Service #1: calculate run-time */
    );
    return;
}
```

In this example, the rule violation occurs because the assembly language code is embedded directly in a C function `taskHandler` that contains other C language statements.

Correction: Encapsulate Assembly Code in Macro

One possible correction is to encapsulate the assembly language code in a macro and invoke the macro in the function `taskHandler`.

```
#define  RUN_TIME_CALC \
asm volatile \
    ( \
        "SWI &02"    /* Service #1: calculate run-Time */ \
    )\

enum boolVal {TRUE, FALSE};
enum boolVal isTaskActive;
void taskHandler(void);

void taskHandler(void) {
    isTaskActive = FALSE;
    RUN_TIME_CALC;
    return;
}
```

Check Information
Group: Code design
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 1.2|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Dir 4.4

Sections of code should not be "commented out"

Description

Directive Definition

Sections of code should not be "commented out".

Rationale

C comments enclosed in /* */ do not support nesting. A comment beginning with /* ends at the first */ even when the */ is intended as the end of a later nested comment. If a section of code that is commented out already contains comments, you can encounter compilation errors (or at least comment out less code than you intend).

Commenting out code is not a good practice. The commented out code can remain out of sync with the surrounding code without causing compilation errors. Later, if you uncomment the code, you can encounter unexpected issues.

Use comments only to explain aspects of the code that are not apparent from the code itself.

Polyspace Implementation

The checker uses internal heuristics to detect commented out code. For instance, characters such as #, ;, { or } indicate comments that might potentially contain code. These comments are then evaluated against other metrics to determine the likelihood of code masquerading as comment. For instance, several successive words without a symbol in between reduces this likelihood.

The checker does not flag the following comments even if they contain code:

- Doxygen comments beginning with /**, /*!, /// or //!.
- Comments that repeat the same symbol several times, for instance, the symbol = here:

  ```
  /* ==================================
   * A comment
   * ==================================*/
  ```
- Comments on the first line of a file.
- Comments that mix the C style (/* */) and C++ style (//).

The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Code Commented Out

```
#include <stdlib.h>

int32_t getRandInt();
void print32_t(int32_t);

/* Function to print32_t random int32_tegers*/
void print32_tInteger() {
    /* int32_t val = getRandInt();
     * val++;
     * print32_t(val); */
    print32_t(getRandInt());
}
```

This example contains a block of commented out code that violates the rule.

Check Information
Group: Code design
Category: Advisory
AGC Category: Advisory

See Also
```
Check MISRA C:2012 (-misra3)
```

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2020b

MISRA C:2012 Dir 4.5

Identifiers in the same name space with overlapping visibility should be typographically unambiguous

Description

Directive Definition

Identifiers in the same name space with overlapping visibility should be typographically unambiguous.

Rationale

What "unambiguous" means depends on the alphabet and language in which source code is written. When you use identifiers that are typographically close, you can confuse between them.

For the Latin alphabet as used in English words, at a minimum, the identifiers should not differ by:

- The interchange of a lowercase letter with its uppercase equivalent.
- The presence or absence of the underscore character.
- The interchange of the letter O and the digit 0.
- The interchange of the letter I and the digit 1.
- The interchange of the letter I and the letter l.
- The interchange of the letter S and the digit 5.
- The interchange of the letter Z and the digit 2.
- The interchange of the letter n and the letter h.
- The interchange of the letter B and the digit 8.
- The interchange of the letters rn and the letter m.

Polyspace Implementation

The checker flags identifiers in the same scope that differ from each other only in the above characters.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  /* Non-compliant */

    int id2_numval;
    int id2_numVal;   /* Non-compliant */
```

```
        int id3_lvalue;
        int id3_Ivalue;    /* Non-compliant */

        int id4_xyZ;
        int id4_xy2;       /* Non-compliant */

        int id5_zer0;
        int id5_zer0;      /* Non-compliant */

        int id6_rn;
        int id6_m;         /* Non-compliant */
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

Check Information

Group: Code design
Category: Advisory
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2015b

MISRA C:2012 Dir 4.6

typedefs that indicate size and signedness should be used in place of the basic numerical types

Description

Directive Definition

typedefs that indicate size and signedness should be used in place of the basic numerical types.

Rationale

When the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

Polyspace Implementation

The rule checker flags use of basic data types in variable or function declarations and definitions. The rule enforces use of typedefs instead.

The rule checker does not flag the use of basic types in the typedef statements themselves.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Direct Use of Basic Types in Definitions

```
typedef unsigned int uint32_t;

int x = 0;        /* Non compliant */
uint32_t y = 0;   /* Compliant */
```

In this example, the declaration of x is noncompliant because it uses a basic type directly.

Check Information
Group: Code design
Category: Advisory
AGC Category: Advisory

See Also
Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Dir 4.8

If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden

Description

Rule Definition

If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.

Rationale

If a pointer to a structure or union is not dereferenced in a file, the implementation details of the structure or union need not be available in the translation unit for the file. You can hide the implementation details such as structure members and protect them from unintentional changes.

Define an opaque type that can be referenced via pointers but whose contents cannot be accessed.

Polyspace Implementation

If a structure or union is defined in a file or a header file included in the file, a pointer to this structure or union declared but the pointer never dereferenced in the file, the checker flags a coding rule violation. The structure or union definition should not be visible to this file.

If you see a violation of this rule on a structure definition, identify if you have defined a pointer to the structure in the same file or in a header file included in the file. Then check if you dereference the pointer anywhere in the file. If you do not dereference the pointer, the structure definition should be hidden from this file and included header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Object Implementation Revealed

`file.h`: Contains structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct  {
  int a;
} myStruct;

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

myStruct* getObj(void);
void useObj(myStruct*);

void func() {
  myStruct *sPtr = getObj();
  useObj(sPtr);
}
```

In this example, the pointer to the type `myStruct` is not dereferenced. The pointer is simply obtained from the `getObj` function and passed to the `useObj` function.

The implementation of `myStruct` is visible in the translation unit consisting of `file.c` and `file.h`.

Correction — Define Opaque Type

One possible correction is to define an opaque data type in the header file `file.h`. The opaque data type `ptrMyStruct` points to the `myStruct` structure without revealing what the structure contains. The structure `myStruct` itself can be defined in a separate translation unit, in this case, consisting of the file `file2.c`. The common header file `file.h` must be included in both `file.c` and `file2.c` for linking the structure definition to the opaque type definition.

`file.h`: Does not contain structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct myStruct *ptrMyStruct;

ptrMyStruct getObj(void);
void useObj(ptrMyStruct);

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

void func() {
  ptrMyStruct sPtr = getObj();
  useObj(sPtr);
}
```

`file2.c`: Includes `file.h` and dereferences structure.

```
#include "file.h"

struct myStruct {
  int a;
};

void useObj(ptrMyStruct ptr) {
    (ptr->a)++;
}
```

Check Information
Group: Code design
Category: Advisory
AGC Category: Advisory

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2018a

MISRA C:2012 Dir 4.9

A function should be used in preference to a function-like macro where they are interchangeable

Description

Directive Definition

A function should be used in preference to a function-like macro where they are interchangeable.

Rationale

In most circumstances, use functions instead of macros. Functions perform argument type-checking and evaluate their arguments once, avoiding problems with potential multiple side effects.

Polyspace Implementation

Polyspace considers all function-like macro definitions.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Code design
Category: Advisory
AGC Category: Advisory

See Also

MISRA C:2012 Rule 13.2 | MISRA C:2012 Rule 20.7 | Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Dir 4.12

Dynamic memory allocation shall not be used

Description

Rule Definition

Dynamic memory allocation shall not be used.

Rationale

Using dynamic memory allocation and deallocation routines provided by the Standard Library or third-party libraries can cause undefined behavior. For instance:

- You use `free` to deallocate memory that you did not allocate with `malloc`, `calloc`, or `realloc`.
- You use a pointer that points to a freed memory location.
- You access allocated memory that has no value stored into it.

Dynamic memory allocation and deallocation routines from third-party libraries are likely to exhibit similar undefined behavior.

If you choose to use dynamic memory allocation and deallocation routines, ensure that your program behavior is predictable. For example, ensure that you safely handle allocation failure due to insufficient memory.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of `malloc`, `calloc`, `realloc` and `free`

```
#include <stdlib.h>

static int foo(void);

typedef struct struct_1 {
    int a;
    char c;
} S_1;

static int foo(void) {

    S_1 * ad_1;
    int  * ad_2;
    int  * ad_3;

    ad_1 = (S_1*)calloc(100U, sizeof(S_1));        /* Non-compliant */
    ad_2 = malloc(100U * sizeof(int));             /* Non-compliant */
    ad_3 = realloc(ad_3, 60U * sizeof(long));      /* Non-compliant */
```

```
    free(ad_1);                                    /* Non-compliant */
    free(ad_2);                                    /* Non-compliant */
    free(ad_3);                                    /* Non-compliant */

    return 1;
}
```

In this example, the rule is violated when the functions `malloc`, `calloc`, `realloc` and `free` are used.

Check Information
Group: Code Design
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2019b

MISRA C:2012 Dir 4.10

Precautions shall be taken in order to prevent the contents of a header file being included more than once

Description

Directive Definition

Precautions shall be taken in order to prevent the contents of a header file being included more than once.

Rationale

When a translation unit contains a complex hierarchy of nested header files, it is possible for a particular header file to be included more than once, leading to confusion. If this multiple inclusion produces multiple or conflicting definitions, then your program can have undefined or erroneous behavior.

For instance, suppose that a header file contains:

```
#ifdef _WIN64
    int env_var;
#elseif
    long int env_var;
#endif
```

If the header file is contained in two inclusion paths, one that defines the macro _WIN64 and another that undefines it, you can have conflicting definitions of env_var.

Polyspace Implementation

If you include a header file whose contents are not guarded from multiple inclusion, the analysis raises a violation of this directive. The violation is shown at the beginning of the header file.

You can guard the contents of a header file from multiple inclusion by using one of the following methods:

```
<start-of-file>
#ifndef <control macro>
#define <control macro>
    /* Contents of file */
#endif
<end-of-file>
```

or

```
<start-of-file>
#ifdef <control macro>
#error ...
#else
#define <control macro>
```

```
    /* Contents of file */
#endif
<end-of-file>
```

Unless you use one of these methods, Polyspace flags the header file inclusion as noncompliant.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Code After Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func(void);
#endif
void func2(void);
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func2(void)` is outside the guard.

Note You can have comments outside the macro guard.

Code Before Macro Guard

```
void func(void);
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func(void)` is outside the guard.

Note You can have comments outside the macro guard.

Mismatch in Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MARCO__
    void func(void);
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro name in the `#ifndef` statement is different from the name in the following `#define` statement.

Check Information
Group: Code Design

Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Dir 4.11

The validity of values passed to library functions shall be checked

Description

Directive Definition

The validity of values passed to library functions shall be checked.

Rationale

Many Standard C functions do not check the validity of parameters passed to them. Even if checks are performed by a compiler, there is no guarantee that the checks are adequate. For example, you should not pass negative numbers to `sqrt` or `log`.

Polyspace Implementation

Polyspace raises a violation result for library function arguments if the following are all true:

- Argument is a local variable.
- Local variable is not tested between last assignment and call to the library function.
- Corresponding parameter of the library function has a restricted input domain.
- Library function is one of the following common mathematical functions:

  - `sqrt`
  - `tan`
  - `pow`
  - `log`
  - `log10`
  - `fmod`
  - `acos`
  - `asin`
  - `acosh`
  - `atanh`
  - or `atan2`

Bug Finder and Code Prover check this rule differently. The analysis can produce different results.

Tip To mass-justify all results related to the same library function, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Code design
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Dir 4.1 | Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 1.1

The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits

Description

Rule Definition

The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.

Polyspace Implementation

The rule checker checks for the issues below. Note that:

- The specifications can depend on the version of the C standard used in the analysis. See `C standard version (-c-version)`.
- You can change some of the limits used by the checker using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

Issue	C Standard Dependence	Additional Information
An integer constant falls outside the range of `long int` (if the constant is signed) or `unsigned long int` (if the constant is unsigned).	Checked for C90 only.	The rule checker uses your specifications for the size of a `long int` variable (typically 32 bits). See also `Target processor type (-target)`.
An array of size zero is used.	Checked for C90 only.	
The number of macros defined in a translation unit exceeds the limit specified in the standard.	Number of macro definitions allowed: • C90: 1024 • C99 and later: 4095	The rule checker considers a translation unit as a source file and header files included directly or indirectly in the source file.
The depth of nesting in control flow statements (like `if`, `while`, etc.) exceeds the limit specified in the standard.	Maximum nesting depth allowed: • C90: 15 • C99 and later: 127	
The number of levels of inclusion using include files exceeds the limit specified in the standard.	Maximum number of levels of inclusion allowed: • C90: 8 • C99 and later: 15	

Issue	C Standard Dependence	Additional Information
The number of members of a structure or union exceeds the limit specified in the standard.	Maximum number of members in a structure or union: • C90: 127 • C99 and later: 1023	
The number of levels of nesting in a structure exceeds the limit specified in the standard.	Maximum depth of nesting: • C90: 15 • C99 and later: 63	
The number of constants in a single enumeration exceeds the limit specified in the standard.	Maximum number of enumeration constants allowed: • C90: 127 • C99 and later: 1023	
An assembly language statement is used.	Checked for all C standards.	
A nonstandard preprocessor directive is used.	Checked for all C standards.	The rule checker flags uses of preprocessor directives that are not found in the C standard, for instance, `#ident`, `#alias` and `#assert`.
Unrecognized text follows a preprocessor directive.	Checked for all C standards.	The rule checker flags extraneous text following a preprocessor directive (line beginning with #). For instance: `#include <header> code`

Standard compilation error messages do not lead to a violation of this MISRA rule.

Tip To mass-justify all results that come from the same cause, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the Shift key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Standard C Environment
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 1.2|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 1.2

Language extensions should not be used

Description

Rule Definition

Language extensions should not be used.

Rationale

If a program uses language extensions, its portability is reduced. Even if you document the language extensions, the documentation might not describe the behavior in all circumstances.

Polyspace Implementation

The rule checker flags these language extensions, depending on the version of the C standard used in the analysis. See `C standard version (-c-version)`.

- C90:

  - `long long int` type including constants
  - `long double` type
  - `inline` keyword
  - `_Bool` keyword
  - `short long int` type
  - Hexadecimal floating-point constants
  - Universal character names
  - Designated initializers
  - Local label declarations
  - `typeof` operator
  - Casts to union
  - Compound literals
  - Statements and declarations in expressions
  - `__func__` predefined identifier
  - `_Pragma` preprocessing operator
  - Macros with variable arguments list
- C99:

  - `short long int` type
  - Local label declarations
  - `typeof` operator
  - Casts to union
  - Statements and declarations in expressions

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Standard C Environment
Category: Advisory
AGC Category: Advisory

See Also
MISRA C:2012 Rule 1.1|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 1.3

There shall be no occurrence of undefined or critical unspecified behaviour

Description

Rule Definition

There shall be no occurrence of undefined or critical unspecified behaviour.

Additional Message in Report

There shall be no occurrence of undefined or critical unspecified behavior

- 'defined' without an identifier.
- macro 'XX' used with too few arguments.
- macro 'XX used with too many arguments.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Standard C Environment
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.1|Check MISRA C:2012 (-misra3)`

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 1.4

Emergent language features shall not be used

Description

Rule Definition

Emergent language features shall not be used.

Rationale

Some new language features in the C11 Standard have undefined, unspecified or implementation-defined behavior. These features might also exhibit well-defined behavior that defies developer expectations. Though rule 1.3 and directive 1.1 prohibits undefined and implementation-defined behavior, to avoid well-defined behavior that defies expectations, some language features are summarily discouraged using rule 1.4.

Polyspace Implementation

The rule forbids use of the following language features:

- The `_Generic` operator.
- The `_Noreturn` function specifier and the `<stdnoreturn.h>` header file
- The `_Atomic` type specifier and the facilities provided by `<stdatomic.h>` (for instance, the macros beginning with `ATOMIC_` and functions beginning with `atomic_` implemented as macros in `<stdatomic.h>`).
- The `_Thread_local` storage class specifier and the facilities provided by `<threads.h>` (for instance, types such as `thrd_t` and functions such as `thrd_create`).
- The `_Alignas` alignment specifier, the `_Alignof` operator and the `<stdalign.h>` header file, and facilities therein (such as the `alignas` and `alignof` macros).
- All facilities in Annex K of the C11 Standard about 'Bound-checking interfaces', other than defining `__STDC_WANT_LIB_EXT1__` to '0'

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of Facilities in Annex K of C11 Standard

```
#define __STDC_WANT_LIB_EXT1__ 1 //Noncompliant
#include <string.h>

void Copying_functions(void) {
    char buf1[10];
    char buf2[10];
    errno_t e;  //Noncompliant
    e = memcpy_s(buf1,sizeof(buf1),buf2,5); //Noncompliant
    e = memmove_s(buf1,sizeof(buf1),buf2,5); //Noncompliant
```

```
    e = strcpy_s(buf1,sizeof(buf1),buf2); //Noncompliant
    e = strncpy_s(buf1,sizeof(buf1),buf2,5); //Noncompliant
}
```

In this example, the macro __STDC_WANT_LIB_EXT1__ is set to 1 so that the type `errno_t` as defined in the header `stdlib.h` can be used (in accordance with Annex K of the C11 Standard).

The checker flags both the setting of the macro to 1 and the definition of the `errno_t` variable, along with other functions from Annex K.

Check Information
Group: Standard C Environment
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Dir 4.1|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 2.1

A project shall not contain unreachable code

Description

Rule Definition

A project shall not contain unreachable code.

Rationale

Unless a program exhibits any undefined behavior, unreachable code cannot execute. The unreachable code cannot affect the program output. The presence of unreachable code can indicate an error in the program logic. Unreachable code that the compiler does not remove wastes resources, for example:

- It occupies space in the target machine memory.
- Its presence can cause a compiler to select longer, slower jump instructions when transferring control around the unreachable code.
- Within a loop, it can prevent the entire loop from residing in an instruction cache.

Polyspace Implementation

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

The Code Prover run-time check for unreachable code shows more cases than the MISRA checker for rule 2.1. See also `Unreachable code`. The run-time check performs a more exhaustive analysis. In the process, the check can show some instances that are not strictly unreachable code but unreachable only in the context of the analysis. For instance, in the following code, the run-time check shows a potential division by zero in the first line and then removes the zero value of `flag` for the rest of the analysis. Therefore, it considers the `if` block unreachable.

```
val=1.0/flag;
if(!flag) {}
```

The MISRA checker is designed to prevent these kinds of results.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Code Following `return` Statement

```
enum light { red, amber, red_amber, green };

enum light next_light ( enum light color )
{
    enum light res;
```

```
    switch ( color )
    {
    case red:
        res = red_amber;
        break;
    case red_amber:
        res = green;
        break;
    case green:
        res = amber;
        break;
    case amber:
        res = red;
        break;
    default:
    {
        error_handler ();
        break;
    }
    }

    res = color;
    return res;
    res = color;      /* Non-compliant */
}
```

In this example, the rule is violated because there is an unreachable operation following the `return` statement.

Check Information

Group: Unused Code
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 14.3|MISRA C:2012 Rule 16.4|Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 2.2

There shall be no dead code

Description

Rule Definition

There shall be no dead code.

Rationale

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

Operations involving language extensions such as `__asm ( "NOP" );` are not considered dead code.

Polyspace Implementation

Polyspace Bug Finder detects useless write operations during analysis.

Polyspace Code Prover does not detect useless write operations. For instance, if you assign a value to a local variable but do not read it later, Polyspace Code Prover does not detect this useless assignment. Use Polyspace Bug Finder to detect such useless write operations.

In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. See "Check for Coding Standard Violations".

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Redundant Operations

```
extern volatile unsigned int v;
extern char *p;

void f ( void ) {
    unsigned int x;


    ( void ) v;      /* Compliant - Exception*/
    ( int ) v;       /* Non-compliant  */
    v >> 3;          /* Non-compliant  */

    x = 3;           /* Non-compliant - Detected in Bug Finder only */
```

```
    *p++;              /* Non-compliant  */
    ( *p )++;          /* Compliant  */
}
```

In this example, the rule is violated when an operation is performed on a variable, but the result of that operation is not used. For instance,

- The operations `(int)` and `>>` on the variable v are redundant because the results are not used.
- The operation = is redundant because the local variable x is not read after the operation.
- The operation * on p++ is redundant because the result is not used.

The rule is not violated when:

- A variable is cast to `void`. The cast indicates that you are intentionally not using the value.
- The result of an operation is used. For instance, the operation * on p is not redundant, because *p is incremented.

Redundant Function Call

```
void g ( void ) {
               /* Compliant  */
}

void h ( void) {
    g( );     /* Non-compliant */
}
```

In this example, g is an empty function. Though the function itself does not violate the rule, a call to the function violates the rule.

Check Information
Group: Unused Code
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 17.7|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 2.3

A project should not contain unused type declarations

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

A project should not contain unused type declarations.

Rationale

If a type is declared but not used, a reviewer does not know if the type is redundant or if it is unused by mistake.

Additional Message in Report

A project should not contain unused type declarations: type XX is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Unused Local Type

```
signed short unusedType (void){

    typedef signed short myType;   /* Non-compliant */
    return 67;

}

signed short usedType (void){

    typedef signed short myType;  /* Compliant */
    myType tempVar = 67;
    return tempVar;

}
```

In this example, in function `unusedType`, the `typedef` statement defines a new local type `myType`. However, this type is never used in the function. Therefore, the rule is violated.

The rule is not violated in the function `usedType` because the new type `myType` is used.

Check Information
Group: Unused Code

Category: Advisory
AGC Category: Readability

See Also
`MISRA C:2012 Rule 2.4`|`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 2.4

A project should not contain unused tag declarations

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

A project should not contain unused tag declarations.

Rationale

If a tag is declared but not used, a reviewer does not know if the tag is redundant or if it is unused by mistake.

Additional Message in Report

A project should not contain unused tag declarations: tag *tag_name* is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Tag Defined in Function but Not Used

```
void unusedTag ( void )
{
    enum state1 { S_init, S_run, S_sleep };    /* Non-compliant  */
}

void usedTag ( void )
{
    enum state2 { S_init, S_run, S_sleep };    /* Compliant  */
    enum state2 my_State = S_init;
}
```

In this example, in the function `unusedTag`, the tag `state1` is defined but not used. Therefore, the rule is violated.

Tag Used in `typedef` Only

```
typedef struct record_t        /* Non-compliant  */
{
    unsigned short key;
    unsigned short val;
} record1_t;


typedef struct              /* Compliant */
```

```
{
    unsigned short key;
    unsigned short val;
} record2_t;

record1_t myRecord1_t;
record2_t myRecord2_t;
```

In this example, the tag `record_t` appears only in the `typedef` of `record1_t`. In the rest of the translation unit, the type `record1_t` is used. Therefore, the rule is violated.

Check Information
Group: Unused Code
Category: Advisory
AGC Category: Readability

See Also
`MISRA C:2012 Rule 2.3` | `Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 2.5

A project should not contain unused macro declarations

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

A project should not contain unused macro declarations.

Rationale

If a macro is declared but not used, a reviewer does not know if the macro is redundant or if it is unused by mistake.

Additional Message in Report

A project should not contain unused macro declarations: macro *macro_name* is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Unused Macro Definition

```
void use_macro (void)
{
    #define SIZE 4
    #define DATA 3

    use_int16(SIZE);
}
```

In this example, the macro DATA is never used in the use_macro function.

Check Information

Group: Unused Code
Category: Advisory
AGC Category: Readability

See Also

Check MISRA C:2012 (-misra3)

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"

"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 2.6

A function should not contain unused label declarations

Description

Rule Definition

A function should not contain unused label declarations.

Rationale

If you declare a label but do not use it, it is not clear to a reviewer of your code if the label is redundant or unused by mistake.

Additional Message in Report

A function should not contain unused label declarations.

Label *label_name* is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Unused Label Declarations

```
void use_var(signed short);

void unused_label ( void )
{
    signed short x = 6;

label1:                         /* Non-compliant - label1 not used */
    use_var ( x );
}

void used_label ( void )
{
    signed short x = 6;

    for (int i=0; i < 5; i++) {
        if ( i==2 ) goto label1;
    }

label1:                         /* Compliant - label1 used */
    use_var ( x );
}
```

In this example, the rule is violated when the label `label1` in function `unused_label` is not used.

Check Information
Group: Unused code
Category: Advisory
AGC Category: Readability

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2015b

MISRA C:2012 Rule 2.7

There should be no unused parameters in functions

Description

Rule Definition

There should be no unused parameters in functions.

Rationale

If a parameter is unused, it is possible that the implementation of the function does not match its specifications. This rule can highlight such mismatches.

Additional Message in Report

There should be no unused parameters in functions.

Parameter *parameter_name* is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Unused Function Parameters

```
double func(int param1, int* param2) { /* Non-compliant */
    return (param1/2.0);
}
```

In this example, the rule is violated because the parameter `param2` is not used.

Check Information
Group: Unused code
Category: Advisory
AGC Category: Readability

See Also
Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2015b

MISRA C:2012 Rule 3.1

The character sequences /* and // shall not be used within a comment

Description

Rule Definition

The character sequences / and // shall not be used within a comment.*

Rationale

These character sequences are not allowed in code comments because:

- If your code contains a /* or a // in a /* */ comment, it typically means that you have inadvertently commented out code.
- If your code contains a /* in a // comment, it typically means that you have inadvertently uncommented a /* */ comment.

Polyspace Implementation

You cannot annotate this rule in the source code.

For information on annotations, see "Annotate Code and Hide Known or Acceptable Results".

Additional Message in Report

The character sequence /* shall not appear within a comment.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

/* Used in // Comments

```
int x;
int y;
int z;

void non_compliant_comments ( void )
{
    x = y //      /* Non-compliant
        + z
        //  */
        ;
    z++;    //    Compliant with exception: // permitted within a // comment
}

void compliant_comments ( void )
{
    x = y /*      Compliant
```

```
        + z
        */
          ;
    z++;    //     Compliant with exception: // is permitted within a // comment
}
```

In this example, in the `non_compliant_comments` function, the `/*` character occurs in what appears to be a `//` comment, violating the rule. Because of the comment structure, the operation that takes place is `x = y + z;`. However, without the two `//`-s, an entirely different operation `x=y;` takes place. It is not clear which operation is intended.

Use a comment format that makes your intention clear. For instance, in the `compliant_comments` function, it is clear that the operation `x=y;` is intended.

Check Information
Group: Comments
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 3.2

Line-splicing shall not be used in `//` comments

Description

Rule Definition

Line-splicing shall not be used in // comments.

Rationale

Line-splicing occurs when the \ character is immediately followed by a new-line character. Line splicing is used for statements that span multiple lines.

If you use line-splicing in a `//` comment, the following line can become part of the comment. In most cases, the \ is spurious and can cause unintentional commenting out of code.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Line Splicing in `//` Comment

```
#include <stdbool.h>

extern _Bool b;

void func ( void )
{
    unsigned short x = 0;   // Non-compliant - Line-splicing \
    if ( b )
    {
        ++b;
    }
}
```

Because of line-splicing, the statement `if ( b )` is a part of the previous `//` comment. Therefore, the statement `b++` always executes, making the `if` block redundant.

Check Information
Group: Comments
Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"

"Software Quality Objective Subsets (C:2012)"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 4.1

Octal and hexadecimal escape sequences shall be terminated

Description

Rule Definition

Octal and hexadecimal escape sequences shall be terminated.

Rationale

There is potential for confusion if an octal or hexadecimal escape sequence is followed by other characters. For example, the character constant `'\x1f'` consists of a single character, whereas the character constant `'\x1g'` consists of the two characters `'\x1'` and `'g'`. The manner in which multi-character constants are represented as integers is implementation-defined.

If every octal or hexadecimal escape sequence in a character constant or string literal is terminated, you reduce potential confusion.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Compliant and Noncompliant Escape Sequences

```
const char *s1 = "\x41g";      /* Non-compliant */
const char *s2 = "\x41" "g";   /* Compliant - Terminated by end of literal */
const char *s3 = "\x41\x67";   /* Compliant - Terminated by another escape sequence*/

int c1 = '\141t';              /* Non-compliant */
int c2 = '\141\t';             /* Compliant - Terminated by another escape sequence*/
```

In this example, the rule is violated when an escape sequence is not terminated with the end of string literal or another escape sequence.

Check Information

Group: Character Sets and Lexical Conventions
Category: Required
AGC Category: Required

See Also

Check MISRA C:2012 (`-misra3`)

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 4.2

Trigraphs should not be used

Description

Rule Definition

Trigraphs should not be used.

Rationale

You denote trigraphs with two question marks followed by a specific third character (for instance,`'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

Note Digraphs (`<:` `:>`, `<%` `%>`, `%:`, `%:%:`) are permitted because they are tokens.

Polyspace Implementation

The Polyspace analysis converts trigraphs to the equivalent character for the run-time verification. However, Polyspace also raises a MISRA violation.

The standard requires that trigraphs must be transformed *before* comments are removed during preprocessing. Therefore, Polyspace raises a violation of this rule even if a trigraph appears in code comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Character Sets and Lexical Conventions
Category: Advisory
AGC Category: Advisory

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 5.1

External identifiers shall be distinct

Description

This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).

Rule Definition

External identifiers shall be distinct.

Rationale

External identifiers are ones declared with global scope or storage class `extern`.

If the difference between two names occurs far later in the names, they can be easily mistaken for each other. The readability of the code is reduced.

Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 31 characters. For C90, the difference must occur between the first 6 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`. You can change the number of characters compared using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

Additional Message in Report

External `%s %s` conflicts with the external identifier XX in file YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

C90: First Six Characters of Identifiers Not Unique

```
int engine_temperature_raw;
int engine_temperature_scaled;    /* Non-compliant */
int engin2_temperature;           /* Compliant */
```

In this example, the identifier `engine_temperature_scaled` has the same first six characters as a previous identifier, `engine_temperature_raw`.

C99: First 31 Characters of Identifiers Not Unique

```
int engine_exhaust_gas_temperature_raw;
int engine_exhaust_gas_temperature_scaled; /* Non-compliant */
```

```
int eng_exhaust_gas_temp_raw;
int eng_exhaust_gas_temp_scaled;              /* Compliant */
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same first 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

C90: First Six Characters Identifiers in Different Translation Units Differ in Case Alone

```
/* file1.c */
int abc = 0;


/* file2.c */
int ABC = 0; /* Non-compliant */
```

In this example, the implementation supports 6 significant case-insensitive characters in *external identifiers*. The identifiers in the two translation are different but are not distinct in their significant characters.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required

See Also

MISRA C:2012 Rule 5.2|MISRA C:2012 Rule 5.4|MISRA C:2012 Rule 5.5|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 5.2

Identifiers declared in the same scope and name space shall be distinct

Description

Rule Definition

Identifiers declared in the same scope and name space shall be distinct.

Rationale

If the difference between two names occurs far later in the names, they can be easily mistaken for each other. The readability of the code is reduced.

Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 63 characters. In C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`. You can change the number of characters compared using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

Additional Message in Report

Identifier XX has same significant characters as identifier YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

C90: First 31 Characters of Identifiers Not Unique

```
extern int engine_exhaust_gas_temperature_raw;
static int engine_exhaust_gas_temperature_scaled;      /* Non-compliant */

extern double raw_engine_exhaust_gas_temperature;
static double scaled_engine_exhaust_gas_temperature;  /* Compliant */

void func ( void )
{
  /* Not in the same scope */
  int engine_exhaust_gas_temperature_local;            /* Compliant */
}
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

The rule does not apply if the two identifiers have the same 31 characters but have different scopes. For instance, `engine_exhaust_gas_temperature_local` has the same 31 characters as `engine_exhaust_gas_temperature_raw` but different scope.

Check Information
Group: Identifiers
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 5.1|MISRA C:2012 Rule 5.3|MISRA C:2012 Rule 5.4|MISRA C:2012 Rule 5.5|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 5.3

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Description

Rule Definition

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

Rationale

If two identifiers have the same name but different scope, the identifier in the inner scope hides the identifier in the outer scope. All uses of the identifier name refers to the identifier in the inner scope. This behavior forces the developer to keep track of the scope and reduces code readability.

Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 63 characters. In C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`. You can change the number of characters compared using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

If the identifier that is hidden is declared in a Standard Library header and you do not provide the header for the analysis, the issue is not shown. To see potential conflicts with identifiers declared in a Standard Library header, provide your compiler implementation of the headers for the Polyspace analysis. See "Provide Standard Library Headers for Polyspace Analysis".

Additional Message in Report

Variable XX hides variable XX (FILE line LINE column COLUMN).

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Local Variable Hidden by Another Local Variable in Inner Block

```
typedef signed short int16_t;

void func( void )
{
    int16_t i;
    {
        int16_t i;                    /* Non-compliant */
        i = 3;
    }
}
```

In this example, the identifier `i` defined in the inner block in `func` hides the identifier `i` with function scope.

It is not immediately clear to a reader which `i` is referred to in the statement `i=3`.

Global Variable Hidden by Function Parameter

```
typedef signed short int16_t;

struct astruct
{
    int16_t m;
};

extern void g ( struct astruct *p );
int16_t xyz = 0;

void func ( struct astruct xyz )  /* Non-compliant */
{
    g ( &xyz );
}
```

In this example, the parameter `xyz` of function `func` hides the global variable `xyz`.

It is not immediately clear to a reader which `xyz` is referred to in the statement `g (&xyz )`.

Check Information
Group: Identifiers
Category: Required
AGC Category: Advisory

See Also
`MISRA C:2012 Rule 5.2|MISRA C:2012 Rule 5.8|Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 5.4

Macro identifiers shall be distinct

Description

Rule Definition

Macro identifiers shall be distinct.

Rationale

The names of macro identifiers must be distinct from both other macro identifiers and their parameters.

Polyspace Implementation

The checker raises a violation if two macros that have the same first 63 characters are defined with different values. The checker does not raise a violation if:

- Two macros with the same first 63 characters are defined with the same value (even an empty value).
- The same macro is defined with different values but the macro is undefined in between.

The cutoff of 63 characters applies to a C99-based analysis. In C90, the cutoff is 31 characters. In other words, the checker considers two macros as effectively the same if there is no difference in their first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`. You can change the number of characters compared using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

Additional Message in Report

- Macro identifiers shall be distinct. Macro XX has same significant characters as macro YY.
- Macro identifiers shall be distinct. Macro parameter XX has same significant characters as macro parameter YY in macro ZZ.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

C90: First 31 Characters of Macro Names Not Unique

```
#define engine_exhaust_gas_temperature_raw egt_r
#define engine_exhaust_gas_temperature_scaled egt_s   /* Non-compliant */

#define engine_exhaust_gas_temp_raw egt_r
#define engine_exhaust_gas_temp_scaled egt_s          /* Compliant */
```

In this example, the macro `engine_exhaust_gas_temperature_scaled egt_s` has the same first 31 characters as a previous macro `engine_exhaust_gas_temperature_scaled`.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required

See Also

MISRA C:2012 Rule 5.1|MISRA C:2012 Rule 5.2|MISRA C:2012 Rule 5.5|Check
MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 5.5

Identifiers shall be distinct from macro names

Description

Rule Definition

Identifiers shall be distinct from macro names.

Rationale

The rule requires that macro names that exist only prior to processing must be different from identifier names that also exist after preprocessing. Keeping macro names and identifiers distinct help avoid confusion.

Polyspace Implementation

Polyspace considers two names as distinct if there is a difference between their first 63 characters. In C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`. You can change the number of characters compared using the option `-code-behavior-specifications`. See `-code-behavior-specifications`.

Additional Message in Report

Identifier XX has same significant characters as macro YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Macro Names Same as Identifier Names

```
#define Sum_1(x, y) ( ( x ) + ( y ) )
short Sum_1;                    /* Non-compliant */

#define Sum_2(x, y) ( ( x ) + ( y ) )
short x = Sum_2 ( 1, 2 );       /* Compliant */
```

In this example, `Sum_1` is both the name of an identifier and a macro. `Sum_2` is used only as a macro.

C90: First 31 Characters of Macro Name Same as Identifier Name

```
#define low_pressure_turbine_temperature_1 lp_tb_temp_1
static int low_pressure_turbine_temperature_2;   /* Non-compliant  */
```

In this example, the identifier `low_pressure_turbine_temperature_2` has the same first 31 characters as a previous macro `low_pressure_turbine_temperature_1`.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required

See Also

MISRA C:2012 Rule 5.1|MISRA C:2012 Rule 5.2|MISRA C:2012 Rule 5.4|Check
MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 5.6

A typedef name shall be a unique identifier

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

A typedef name shall be a unique identifier.

Rationale

Reusing a `typedef` name as another `typedef` or as the name of a function, object or `enum` constant can cause developer confusion.

Additional Message in Report

XX conflicts with the typedef name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

typedef Names Reused

```
void func ( void ){
  {
    typedef unsigned char u8_t;
  }
  {
    typedef unsigned char u8_t; /* Non-compliant */
  }
}

typedef float mass;
void func1 ( void ){
  float mass = 0.0f;            /* Non-compliant */
}
```

In this example, the `typedef` name `u8_t` is used twice. The `typedef` name `mass` is also used as an identifier name.

typedef Name Same as Structure Name

```
typedef struct list{          /* Compliant - exception */
  struct list *next;
  unsigned short element;
} list;
```

```
typedef struct{
  struct chain{                    /* Non-compliant */
    struct chain *list2;
    unsigned short element;
  } s1;
  unsigned short length;
} chain;
```

In this example, the `typedef` name `list` is the same as the original name of the `struct` type. The rule allows this exceptional case.

However, the `typedef` name `chain` is not the same as the original name of the `struct` type. The name `chain` is associated with a different `struct` type. Therefore, it clashes with the `typedef` name.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 5.7` | `Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 5.7

A tag name shall be a unique identifier

Description

Rule Definition

A tag name shall be a unique identifier.

Rationale

Reusing a tag name can cause developer confusion.

Additional Message in Report

XX conflicts with the tag name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Identifiers
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 5.6|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 5.8

Identifiers that define objects or functions with external linkage shall be unique

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

Identifiers that define objects or functions with external linkage shall be unique.

Rationale

External identifiers are those declared with global scope or with storage class `extern`. Reusing an external identifier name can cause developer confusion.

Identifiers defined within a function have smaller scope. Even if names of such identifiers are not unique, they are not likely to cause confusion.

Additional Message in Report

- Object XX conflicts with the object name YY.
- Function XX conflicts with the function name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Identifiers
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 5.3|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 5.9

Identifiers that define objects or functions with internal linkage should be unique

Description

This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).

Rule Definition

Identifiers that define objects or functions with internal linkage should be unique.

Polyspace Implementation

This rule checker assumes that rule 5.8 is not violated.

Additional Message in Report

- Object XX conflicts with the object name YY.
- Function XX conflicts with the function name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Identifiers
Category: Advisory
AGC Category: Readability

See Also
MISRA C:2012 Rule 8.10|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 6.1

Bit-fields shall only be declared with an appropriate type

Description

Rule Definition

Bit-fields shall only be declared with an appropriate type.

Rationale

Using `int` for a bit-field type is implementation-defined because bit-fields of type `int` can be either `signed` or `unsigned`.

The use of `enum`, `short char`, or any other type of bit-field is not permitted in C90 because the behavior is undefined.

In C99, the implementation can potentially define other integer types that are permitted in bit-field declarations.

Polyspace Implementation

The checker flags data types for bit-fields other than these allowed types:

- C90: `signed int` or `unsigned int` (or `typedef`-s that resolve to these types)
- C99: `signed int`, `unsigned int` or `_Bool` (or `typedef`-s that resolve to these types)

The results depend on the version of the C standard used in the analysis. See `C standard version (-c-version)`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Types
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 6.2

Single-bit named bit fields shall not be of a signed type

Description

Rule Definition

Single-bit named bit fields shall not be of a signed type.

Rationale

According to the C99 Standard Section 6.2.6.2, a single-bit signed bit-field has one sign bit and no value bits. In any representation of integers, zero value bits cannot specify a meaningful value.

A single-bit signed bit-field is therefore unlikely to behave in a useful way. Its presence is likely to indicate programmer confusion.

Although the C90 Standard does not provide much detail regarding the representation of types, the same single-bit bit-field considerations apply.

Polyspace Implementation

This rule does not apply to unnamed bit fields because their values cannot be accessed.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Types
Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 7.1

Octal constants shall not be used

Description

Rule Definition

Octal constants shall not be used.

Rationale

Octal constants are denoted by a leading zero. Developers can mistake an octal constant as a decimal constant with a redundant leading zero.

Polyspace Implementation

If you use octal constants in a macro definition, the rule checker flags the issue even if the macro is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of octal constants

```
#define CST     021             /* Non-Compliant - decimal 17 */
#define VALUE   010             /* Compliant - constant not used */
#if 010 == 01                   /* Non-Compliant - constant used */
#define CST 021                 /* Non-Compliant - constant not used */
#endif

extern short code[5];
static char* str2 = "abcd\0efg";  /* Compliant */

void main(void) {
    int value1 = 0;             /* Compliant */
    int value2 = 01;            /* Non-Compliant - decimal 01 */
    int value3 = 1;             /* Compliant */
    int value4 = '\109';        /* Compliant */

    code[1] = 109;              /* Compliant     - decimal 109 */
    code[2] = 100;              /* Compliant     - decimal 100 */
    code[3] = 052;              /* Non-Compliant - decimal 42 */
    code[4] = 071;              /* Non-Compliant - decimal 57 */

    if (value1 != CST) {
        value1 = !(value1 != 0);  /* Compliant */
    }
}
```

In this example, the rule is not violated when octal constants are used to define macros CST and VALUE. The rule is violated only when the macros are used.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Advisory

See Also

`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 7.2

A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type

Description

Rule Definition

A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.

Rationale

The signedness of a constant is determined from:

- Value of the constant.
- Base of the constant: octal, decimal or hexadecimal.
- Size of the various types.
- Any suffixes used.

Unless you use a suffix u or U, another developer looking at your code cannot determine easily whether a constant is signed or unsigned.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Literals and Constants
Category: Required
AGC Category: Readability

See Also
Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 7.3

The lowercase character "l" shall not be used in a literal suffix

Description

Rule Definition

The lowercase character "l" shall not be used in a literal suffix.

Rationale

The lowercase character "l" can be confused with the digit "1". Use the uppercase "L" instead.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Literals and Constants
Category: Required
AGC Category: Readability

See Also
Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 7.4

A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char"

Description

Rule Definition

A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Rationale

This rule prevents assignments that allow modification of a string literal.

An attempt to modify a string literal can result in undefined behavior. For example, some implementations can store string literals in read-only memory. An attempt to modify the string literal can result in an exception or crash.

Polyspace Implementation

The rule checker flags assignment of string literals to:

- Pointers with data type other than `const char*`.
- Arrays with data type other than `const char`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Incorrect Assignment of String Literal

```
char *str1 = "xxxxxx";              // Non-Compliant
const char *str2 = "xxxxxx";        // Compliant

void checkSystem1(char*);
void checkSystem2(const char*);

void main() {
 checkSystem1("xxxxxx");    // Non-Compliant
 checkSystem2("xxxxxx");    // Compliant
}
```

In this example, the rule is not violated when string literals are assigned to `const char*` pointers, either directly or through copy of function arguments. The rule is violated only when the `const` qualifier is not used.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Required

See Also

MISRA C:2012 Rule 11.4|MISRA C:2012 Rule 11.8|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.1

Types shall be explicitly specified

Description

Rule Definition

Types shall be explicitly specified.

Rationale

In some circumstances, you can omit types from the C90 standard. In those cases, the `int` type is implicitly specified. However, the omission of an explicit type can lead to confusion. For example, in the declaration `extern void foo (char c, const k);`, the type of `k` is `const int`, but you might expect `const char`.

You might be using an implicit type in:

- Object declarations
- Parameter declarations
- Member declarations
- `typedef` declarations
- Function return types

Polyspace Implementation

The rule checker flags situations where a function parameter or return type is not explicitly specified.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Implicit Types

```
static foo(int a);   /* Non compliant */
static void bar(void);      /* Compliant */
```

In this example, the rule is violated because the return type of `foo` is implicit.

Check Information
Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 8.2|Check MISRA C:2012 (-misra3)

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.2

Function types shall be in prototype form with named parameters

Description

Rule Definition

Function types shall be in prototype form with named parameters.

Rationale

The rule requires that you specify names and data types for all the parameters in a declaration. The parameter names provide useful information regarding the function interface. A mismatch between a declaration and definition can indicate a programming error. For instance, you mixed up parameters when defining the function. By insisting on parameter names, the rule allows a code reviewer to detect this mismatch.

Polyspace Implementation

The rule checker shows a violation if the parameters in a function declaration or definition are missing names or data types.

Additional Message in Report

- Too many arguments to *function_name*.
- Too few arguments to *function_name*.
- Function types shall be in prototype form with named parameters.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Function Prototype Without Named Parameters

```
extern int func(int);    /* Non compliant */
extern int func2(int n);   /* Compliant */

extern int func3();    /* Non compliant */
extern int func4(void);   /* Compliant */
```

In this example, the declarations of `func` and `func3` are noncompliant because the parameters are missing or do not have names.

Check Information
Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 8.1` | `MISRA C:2012 Rule 8.4` | `MISRA C:2012 Rule 17.3` | `Check MISRA C:2012 (-misra3)`

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.3

All declarations of an object or function shall use the same names and type qualifiers

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

All declarations of an object or function shall use the same names and type qualifiers.

Rationale

Consistently using parameter names and types across declarations of the same object or function encourages stronger typing. It is easier to check that the same function interface is used across all declarations.

Polyspace Implementation

The rule checker detects situations where parameter names or data types are different between multiple declarations or the declaration and the definition. The checker considers declarations in all translation units and flags issues that are not likely to be detected by a compiler.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

- Definition of function *function_name* incompatible with its declaration.
- Global declaration of *function_name* function has incompatible type with its definition.
- Global declaration of *variable_name* variable has incompatible type with its definition.
- All declarations of an object or function shall use the same names and type qualifiers.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Mismatch in Parameter Names

```
extern int div (int num, int den);

int div(int den, int num) { /* Non compliant */
    return(num/den);
}
```

In this example, the rule is violated because the parameter names in the declaration and definition are switched.

Mismatch in Parameter Data Types

```
typedef unsigned short width;
typedef unsigned short height;
typedef unsigned int area;

extern area calculate(width w, height h);

area calculate(width w, width h) { /* Non compliant */
    return w*h;
}
```

In this example, the rule is violated because the second argument of the `calculate` function has data type:

- `height` in the declaration.
- `width` in the definition.

The rule is violated even though the underlying type of `height` and `width` are identical.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 8.4|Check MISRA C:2012 (-misra3)`

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.4

A compatible declaration shall be visible when an object or function with external linkage is defined

Description

Rule Definition

A compatible declaration shall be visible when an object or function with external linkage is defined.

Rationale

If a declaration is visible when an object or function is defined, it allows the compiler to check that the declaration and the definition are compatible.

This rule with `MISRA C:2012 Rule 8.5` enforces the practice of declaring an object (or function) in a header file and including the header file in source files that define or use the object (or function).

Polyspace Implementation

The rule checker detects situations where:

- An object or function is defined without a previous declaration.
- There is a data type mismatch between the object or function declaration and definition. Such a mismatch also causes a compilation error.

The checker now flags tentative definitions (variables declared without an `extern` specifier and not explicitly defined). To avoid the rule violation, declare the variable `static` (defined in one file only), or declare the variable `extern` and follow the declaration with a definition.

Additional Message in Report

- Global definition of *variable_name* variable has no previous declaration.
- Function *function_name* has no visible compatible prototype at definition.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Definition Without Previous Declaration

Header file:

```
/* file.h */
extern int var2;
void func2(void);
```

Source file:

```
/* file.c */
#include "file.h"

int var1 = 0;    /* Non compliant */
int var2 = 0;    /* Compliant */

void func1(void) {   /* Non compliant */
}

void func2(void) {   /* Compliant */
}
```

In this example, the definitions of `var1` and `func1` are noncompliant because they are not preceded by declarations.

Mismatch in Parameter Data Types

```
void func(int param1, int param2);

void func(int param1, unsigned int param2) { /* Non compliant */
}
```

In this example, the definition of `func` has a different parameter type from its declaration. The declaration mismatch might cause a compilation error. Polyspace flags the mismatch.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Advisory

See Also

MISRA C:2012 Rule 8.2|MISRA C:2012 Rule 8.3|MISRA C:2012 Rule 8.5|MISRA C:2012 Rule 17.3|Check MISRA C:2012 (-misra3)

Topics

"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.5

An external object or function shall be declared once in one and only one file

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

An external object or function shall be declared once in one and only one file.

Rationale

If you declare an identifier in a header file, you can include the header file in any translation unit where the identifier is defined or used. In this way, you ensure consistency between:

- The declaration and the definition.
- The declarations in different translation units.

The rule enforces the practice of declaring external objects or functions in header files.

Polyspace Implementation

The rule checker checks only explicit `extern` declarations (tentative definitions are ignored). The checker flags variables or functions declared `extern` in a non-header file.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

- Object *object_name* has external declarations in multiple files.
- Function *function_name* has external declarations in multiple files.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Extern Declaration in Non-Header File

Header file:

```
/* file.h */
extern int var;
extern void func1(void);    /* Compliant */
```

Source file:

```
/* file.c */
#include "file.h"

extern void func2(void);    /* Non compliant */

/* Definitions */
int var = 0;
void func1(void) {}
```

In this example, the declaration of external function `func2` is noncompliant because it occurs in a non-header file. The other external object and function declarations occur in a header file and comply with this rule.

Check Information
Group: Declarations and Definitions
Category: Required
AGC Category: Advisory

See Also
`MISRA C:2012 Rule 8.4`|`Check MISRA C:2012 (-misra3)`

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.6

An identifier with external linkage shall have exactly one external definition

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

An identifier with external linkage shall have exactly one external definition.

Rationale

If you use an identifier for which multiple definitions exist in different files or no definition exists, the behavior is undefined.

Multiple definitions in different files are not permitted by this rule even if the definitions are the same.

Polyspace Implementation

The checker flags multiple definitions only if the definitions occur in different files.

The checker does not consider tentative definitions as definitions. For instance, the following code does not violate the rule:

```
int val;
int val=1;
```

The checker does not show a violation if a function is not defined at all but declared with external linkage and called in the source code.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

- Forbidden multiple definitions for function *function_name*.
- Forbidden multiple tentative definitions for object *object_name*.
- Global variable *variable_name* multiply defined.
- Function *function_name* multiply defined.
- Global variable has multiple tentative definitions.
- Undefined global variable *variable_name*.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Variable Multiply Defined

First source file:

```
/* file1.c */
extern int var = 1;
```

Second source file:

```
/* file2.c */
int var = 0;    /* Non compliant */
```

In this example, the global variable `var` is multiply defined. Unless explicitly specified with the `static` qualifier, the variables have external linkage.

Function Multiply Defined

Header file:

```
/* file.h */
int func(int param);
```

First source file:

```
/* file1.c */
#include "file.h"

int func(int param) {
    return param+1;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

int func(int param) {  /* Non compliant */
    return param-1;
}
```

In this example, the function `func` is multiply defined. Unless explicitly specified with the `static` qualifier, the functions have external linkage.

Check Information
Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also

Check MISRA C:2012 (`-misra3`)

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.7

Functions and objects should not be defined with external linkage if they are referenced in only one translation unit

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.

Rationale

Compliance with this rule avoids confusion between your identifier and an identical identifier in another translation unit or library. If you restrict or reduce the visibility of an object by giving it internal linkage or no linkage, you or someone else is less likely to access the object inadvertently.

Polyspace Implementation

The rule checker flags:

- Objects that are defined at file scope without the `static` specifier but used only in one file.
- Functions that are defined without the `static` specifier but called only in one file.

If you intend to use the object or function in one file only, declare it static.

If your code does not contain a `main` function and you use options such as `Variables to initialize (-main-generator-writes-variables)` with value `custom` to explicitly specify a set of variables to initialize, the checker does not flag those variables. The checker assumes that in a real application, the file containing the `main` must initialize the variables in addition to any file that currently uses them. Therefore, the variables are used in more than one translation unit.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

- Variable *variable_name* should have internal linkage.
- Function *function_name* should have internal linkage.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Variable with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
```

First source file:

```
/* file1.c */
#include "file.h"

int var;     /* Compliant */
int var2;    /* Non compliant */
static int var3; /* Compliant */

void reset(void);

void reset(void) {
    var = 0;
    var2 = 0;
    var3 = 0;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment(int var2);

void increment(int var2) {
    var++;
    var2++;
}
```

In this example:

- The declaration of `var` is compliant because `var` is declared with external linkage and used in multiple files.
- The declaration of `var2` is noncompliant because `var2` is declared with external linkage but used in one file only.

  It might appear that `var2` is defined in both files. However, in the second file, `var2` is a parameter with no linkage and is not the same as the `var2` in the first file.
- The declaration of `var3` is compliant because `var3` is declared with internal linkage (with the `static` specifier) and used in one file only.

Function with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
extern void increment1 (void);
```

First source file:

```
/* file1.c */
#include "file.h"

int var;

void increment2(void);
static void increment3(void);
void func(void);

void increment2(void) { /* Non compliant */
    var+=2;
}

static void increment3(void) { /* Compliant */
    var+=3;
}

void func(void) {
    increment1();
    increment2();
    increment3();
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment1(void) { /* Compliant */
    var++;
}
```

In this example:

- The definition of `increment1` is compliant because `increment1` is defined with external linkage and called in a different file.

- The declaration of `increment2` is noncompliant because `increment2` is defined with external linkage but called in the same file and nowhere else.

- The declaration of `increment3` is compliant because `increment3` is defined with internal linkage (with the `static` specifier) and called in the same file and nowhere else.

Check Information

Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.8

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage

Description

Rule Definition

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

Rationale

If you do not use the `static` specifier consistently in all declarations of objects with internal linkage, you might declare the same object with external and internal linkage.

In this situation, the linkage follows the earlier specification that is visible (C99 Standard, Section 6.2.2). For instance, if the earlier specification indicates internal linkage, the object has internal linkage even though the latter specification indicates external linkage. If you notice the latter specification alone, you might expect otherwise.

Polyspace Implementation

The rule checker detects situations where:

- The same object is declared multiple times with different storage specifiers.
- The same function is declared and defined with different storage specifiers.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Linkage Conflict Between Variable Declarations

```
static int foo = 0;
extern int foo;          /* Non-compliant */

extern int hhh;
static int hhh;          /* Non-compliant */
```

In this example, the first line defines `foo` with internal linkage. The first line is compliant because the example uses the `static` keyword. The second line does not use `static` in the declaration, so the declaration is noncompliant. By comparison, the third line declares `hhh` with an `extern` keyword creating external linkage. The fourth line declares `hhh` with internal linkage, but this declaration conflicts with the first declaration of `hhh`.

Correction — Consistent `static` and `extern` Use

One possible correction is to use `static` and `extern` consistently:

```
static int foo = 0;
static int foo;

extern int hhh;
extern int hhh;
```

Linkage Conflict Between Function Declaration and Definition

```
static int fee(void);  /* Compliant - declaration: internal linkage */
int fee(void){         /* Non-compliant */
  return 1;
}

static int ggg(int);  /* Compliant - declaration: internal linkage */
extern int ggg(int x){  /* Non-compliant */
  return 1 + x;
}
```

This example shows two internal linkage violations. Because `fee` and `ggg` have internal linkage, you must use a `static` class specifier in the definition to be compliant with MISRA.

Check Information
Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.9

An object should be defined at block scope if its identifier only appears in a single function

Description

Rule Definition

An object should be defined at block scope if its identifier only appears in a single function.

Rationale

If you define an object at block scope, you or someone else is less likely to access the object inadvertently outside the block.

Polyspace Implementation

The rule checker flags `static` objects that are accessed in one function only but declared at file scope.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Object Declared at File Scope but Used in One Function

```
static int ctr;   /* Non compliant */

int checkStatus(void);
void incrementCount(void);

void incrementCount(void) {
    ctr=0;
    while(1) {
        if(checkStatus())
            ctr++;
    }
}
```

In this example, the declaration of `ctr` is noncompliant because it is declared at file scope but used only in the function `incrementCount`. Declare `ctr` in the body of `incrementCount` to be MISRA C-compliant.

Check Information
Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory

See Also
Check MISRA C:2012 (-misra3)

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.10

An inline function shall be declared with the static storage class

Description

Rule Definition

An inline function shall be declared with the static storage class.

Rationale

If you call an inline function that is declared with external linkage but not defined in the same translation unit, the function might not be inlined. You might not see the reduction in execution time that you expect from inlining.

If you want to make an inline function available to several translation units, you can still define it with the `static` specifier. In this case, place the definition in a header file. Include the header file in all the files where you want the function inlined.

Polyspace Implementation

The rule checker flags definitions that contain the `inline` specifier without an accompanying `static` specifier.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Inlining Functions with External Linkage

```
inline double mult(int val);
inline double mult(int val) {    /* Non compliant */
    return val * 2.0;
}

static inline double div(int val);
static inline double div(int val) {  /* Compliant */
    return val / 2.0;
}
```

In this example, the definition of `mult` is noncompliant because it is inlined without the `static` storage specifier.

Check Information
Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 5.9`|`Check MISRA C:2012 (-misra3)`

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.11

When an array with external linkage is declared, its size should be explicitly specified

Description

Rule Definition

When an array with external linkage is declared, its size should be explicitly specified.

Rationale

Although it is possible to declare an array with an incomplete type and access its elements, it is safer to state the size of the array explicitly. If you provide size information for each declaration, a code reviewer can check multiple declarations for their consistency. With size information, a static analysis tool can perform array bounds analysis without analyzing more than one unit.

Polyspace Implementation

The rule checker flags arrays declared with the `extern` specifier if the declaration does not explicitly specify the array size.

Additional Message in Report

Size of array *array_name* should be explicitly stated. When an array with external linkage is declared, its size should be explicitly specified.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Array Declarations

```
#include <stdint.h>

extern int32_t array1[10];   /*  Compliant  */
extern int32_t array2[];     /*  Non-compliant  */
```

In this example, two arrays are declared `array1` and `array2`. `array1` has external linkage (the `extern` keyword) and a size of 10. `array2` also has external linkage, but no specified size. `array2` is noncompliant because for arrays with external linkage, you must explicitly specify a size.

Check Information
Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory

See Also
Check MISRA C:2012 (-misra3)

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.12

Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique

Description

Rule Definition

Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.

Rationale

An implicitly specified enumeration constant has a value one greater than its predecessor. If the first enumeration constant is implicitly specified, then its value is 0. An explicitly specified enumeration constant has the specified value.

If implicitly and explicitly specified constants are mixed within an enumeration list, it is possible for your program to replicate values. Such replications can be unintentional and can cause unexpected behavior.

Polyspace Implementation

The rule checker flags an enumeration if it has an implicitly specified enumeration constant with the same value as another enumeration constant.

Additional Message in Report

The constant *constant1* has same value as the constant *constant2*.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Replication of Value in Implicitly Specified Enum Constants

```
enum color1 {red_1, blue_1, green_1};   /* Compliant */
enum color2 {red_2 = 1, blue_2 = 2, green_2 = 3};        /* Compliant */
enum color3 {red_3 = 1, blue_3, green_3};     /* Compliant */
enum color4 {red_4, blue_4, green_4 = 1};     /* Non Compliant */
enum color5 {red_5 = 2, blue_5, green_5 = 2};     /* Compliant */
enum color6 {red_6 = 2, blue_6, green_6 = 2, yellow_6};     /* Non Compliant */
```

Compliant situations:

- `color1`: All constants are implicitly specified.
- `color2`: All constants are explicitly specified.
- `color3`: Though there is a mix of implicit and explicit specification, all constants have unique values.
- `color5`: The implicitly specified constants have unique values.

Noncompliant situations:

- `color4`: The implicitly specified constant `blue_4` has the same value as `green_4`.
- `color6`: The implicitly specified constant `blue_6` has the same value as `yellow_6`.

Check Information
Group: Declarations and Definitions
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.13

A pointer should point to a const-qualified type whenever possible

Description

Rule Definition

A pointer should point to a const-qualified type whenever possible.

Rationale

This rule ensures that you do not inadvertently use pointers to modify objects.

Polyspace Implementation

The rule checker flags a pointer to a non-`const` function parameter if the pointer does not modify the addressed object. The assumption is that the pointer is not meant to modify the object and so must point to a `const`-qualified type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Pointer That Should Point to `const`-Qualified Types

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(uint16_t *p) {      /* Non-compliant */
    return *p;
}

char last_char(char * const s){      /* Non-compliant */
    return s[strlen(s) - 1u];
}

uint16_t first(uint16_t a[5]){      /* Non-compliant */
    return a[0];
}
```

This example shows three different noncompliant pointer parameters.

- In the `ptr_ex` function, `p` does not modify an object. However, the type to which `p` points is not `const`-qualified, so it is noncompliant.
- In `last_char`, the pointer `s` is `const`-qualified but the type it points to is not. This parameter is noncompliant because `s` does not modify an object.
- The function `first` does not modify the elements of the array `a`. However, the element type is not `const`-qualified, so `a` is also noncompliant.

Correction — Use `const` Keywords

One possible correction is to add `const` qualifiers to the definitions.

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(const uint16_t *p){      /* Compliant */
    return *p;
}

char last_char(const char * const s){   /* Compliant */
    return s[strlen( s ) - 1u];
}

uint16_t first(const uint16_t a[5]) {   /* Compliant */
    return a[0];
}
```

Check Information
Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Avoid Violations of MISRA C:2012 Rules 8.x"
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 8.14

The restrict type qualifier shall not be used

Description

Rule Definition

The restrict type qualifier shall not be used.

Rationale

When you use a `restrict` qualifier carefully, it improves the efficiency of code generated by a compiler. It can also improve static analysis. However, when using the `restrict` qualifier, it is difficult to make sure that the memory areas operated on by two or more pointers do not overlap.

Polyspace Implementation

The rule checker flags all uses of the `restrict` qualifier.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of `restrict` Qualifier

```
void f(int n, int * restrict p, int * restrict q)/*Noncompliant*/
{
}
```

In this example, both uses of the `restrict` qualifier are flagged.

Check Information
Group: Declarations and Definitions
Category: Required
AGC Category: Advisory

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 9.1

The value of an object with automatic storage duration shall not be read before it has been set

Description

Message in Report:

Rule Definition

The value of an object with automatic storage duration shall not be read before it has been set.

Rationale

A variable with an automatic storage duration is allocated memory at the beginning of an enclosing code block and deallocated at the end. All non-global variables have this storage duration, except those declared `static` or `extern`.

Variables with automatic storage duration are not automatically initialized and have indeterminate values. Therefore, you must not read such a variable before you have set its value through a write operation.

Polyspace Implementation

The Polyspace analysis checks some of the violations as non-initialized variables. For more information, see `Non-initialized local variable`.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results. In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. See "Check for Coding Standard Violations".

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Initialization
Category: Mandatory
AGC Category: Mandatory

See Also
`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.3` | `Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 9.2

The initializer for an aggregate or union shall be enclosed in braces

Description

Rule Definition

The initializer for an aggregate or union shall be enclosed in braces.

Rationale

The rule applies to both objects and subobjects. For example, when initializing a structure that contains an array, the values assigned to the structure must be enclosed in braces. Within these braces, the values assigned to the array must be enclosed in another pair of braces.

Enclosing initializers in braces improves clarity of code that contains complex data structures such as multidimensional arrays and arrays of structures.

Tip To avoid nested braces for subobjects, use the syntax {0}, which sets all values to zero.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Initialization of Two-dimensional Arrays

```
void initialize(void) {
    int x[4][2] = {{0,0},{1,0},{0,1},{1,1}}; /* Compliant */
    int y[4][2] = {{0},{1,0},{0,1},{1,1}};   /* Compliant */
    int z[4][2] = {0};                       /* Compliant */
    int w[4][2] = {0,0,1,0,0,1,1,1};         /* Non-compliant */
}
```

In this example, the rule is not violated when:

- Initializers for each row of the array are enclosed in braces.
- The syntax {0} initializes all elements to zero.

The rule is violated when a separate pair of braces is not used to enclose the initializers for each row.

Check Information
Group: Initialization
Category: Required
AGC Category: Readability

See Also
Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 9.3

Arrays shall not be partially initialized

Description

Rule Definition

Arrays shall not be partially initialized.

Rationale

Providing an explicit initialization for each array element makes it clear that every element has been considered.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Partial and Complete Initializations

```
void func(void) {
    int x[3] = {0,1,2};            /* Compliant */
    int y[3] = {0,1};              /* Non-compliant */
    int z[3] = {0};                /* Compliant - exception */
    int a[30] = {[1] = 1,[15]=1};  /* Compliant - exception */
    int b[30] = {[1] = 1, 1};      /* Non-compliant */
    char c[20] = "Hello World";    /* Compliant - exception */
}
```

In this example, the rule is not violated when each array element is explicitly initialized.

The rule is violated when some elements of the array are implicitly initialized. Exceptions include the following:

- The initializer has the form {0}, which initializes all elements to zero.
- The array initializer consists *only* of designated initializers. Typically, you use this approach for sparse initialization.
- The array is initialized using a string literal.

Check Information
Group: Initialization
Category: Required
AGC Category: Readability

See Also
Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 9.4

An element of an object shall not be initialized more than once

Description

Rule Definition

An element of an object shall not be initialized more than once.

Rationale

Designated initializers allow explicitly initializing elements of objects such as arrays in any order. However, using designated initializers, one can inadvertently initialize the same element twice and therefore overwrite the first initialization.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Array Initialization Using Designated Initializers

```
void func(void) {
    int a[5] = {-2,-1,0,1,2};                   /* Compliant */
    int b[5] = {[0]=-2, [1]=-1, [2]=0, [3]=1, [4]=2}; /* Compliant */
    int c[5] = {[0]=-2, [1]=-1, [1]=0, [3]=1, [4]=2}; /* Non-compliant */

}
```

In this example, the rule is violated when the array element `c[1]` is initialized twice using a designated initializer.

Structure Initialization Using Designated Initializers

```
struct myStruct {
    int a;
    int b;
    int c;
    int d;
};

void func(void) {
    struct myStruct struct1 = {-4,-2,2,4}; /* Compliant */
    struct myStruct struct2 = {.a=-4, .b=-2, .c=2, .d=4}; /* Compliant */
    struct myStruct struct3 = {.a=-4, .b=-2, .b=2, .d=4}; /* Non-compliant */
}
```

In this example, the rule is violated when `struct3.b` is initialized twice using a designated initializer.

Check Information

Group: Initialization
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 9.5

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly

Description

Rule Definition

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

Rationale

If the size of an array is not specified explicitly, it is determined by the highest index of the elements that are initialized. When using long designated initializers, it might not be immediately apparent which element has the highest index.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Using Designated Initializers Without Specifying Array Size

```
int a[5] = {[0]= 1, [2] = 1, [4]= 1, [1] = 1};          /* Compliant */
int b[] = {[0]= 1, [2] = 1, [4]= 1, [1] = 1};           /* Non-compliant */
int c[] = {[0]= 1, [1] = 1, [2]= 1, [3]=0, [4] = 1};  /* Non-compliant */

void display(int);

void main() {
    func(a,5);
    func(b,5);
    func(c,5);
}

void func(int* arr, int size) {
    for(int i=0; i<size; i++)
        display(arr[i]);
}
```

In this example, the rule is violated when the arrays b and c are initialized using designated initializers but the array size is not specified.

Check Information
Group: Initialization
Category: Required
AGC Category: Readability

See Also

Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 10.1

Operands shall not be of an inappropriate essential type

Description

Rule Definition

Operands shall not be of an inappropriate essential type.

Rationale

What Are Essential Types?

An essential type category defines the essential type of an object or expression.

Essential type category	Standard types
Essentially Boolean	`bool` or `_Bool` (defined in `stdbool.h`) You can also define types that are essentially Boolean using the option `Effective boolean types (-boolean-types)`.
Essentially character	`char`
Essentially enum	named `enum`
Essentially signed	signed `char`, signed `short`, signed `int`, signed `long`, signed `long long`
Essentially unsigned	unsigned `char`, unsigned `short`, unsigned `int`, unsigned `long`, unsigned `long long`
Essentially floating	`float`, `double`, `long double`

Amplification and Rationale

For operands of some operators, you cannot use certain essential types. In the table below, each row represents an operator/operand combination. If the essential type column is not empty for that row, there is a MISRA restriction when using that type as the operand. The number in the table corresponds to the rationale list after the table.

Operation		Essential type category of arithmetic operand					
Operator	Operand	Boolean	character	enum	signed	unsigned	floating
[ ]	integer	3	4				1
+ (unary)		3	4	5			
- (unary)		3	4	5		8	
+ -	either	3		5			
* /	either	3	4	5			
%	either	3	4	5			1
< > <= >=	either	3					

Operation		Essential type category of arithmetic operand					
== !=	either						
! && ||	any		2	2	2	2	2
<< >>	left	3	4	5,6	6		1
<< >>	right	3	4	7	7		1
~ & | ^	any	3	4	5,6	6		1
?:	1st		2	2	2	2	2
?:	2nd and 3rd						

1   An expression of essentially floating type for these operands is a constraint violation.

2   When an operand is interpreted as a Boolean value, use an expression of essentially Boolean type.

3   When an operand is interpreted as a numeric value, do not use an operand of essentially Boolean type.

4   When an operand is interpreted as a numeric value, do not use an operand of essentially character type. The numeric values of character data are implementation-defined.

5   In an arithmetic operation, do not use an operand of essentially enum type. An enum object uses an implementation-defined integer type. An operation involving an enum object can therefore yield a result with an unexpected type.

6   Perform only shift and bitwise operations on operands of essentially unsigned type. When you use shift and bitwise operations on essentially signed types, the resulting numeric value is implementation-defined.

7   To avoid undefined behavior on negative shifts, use an essentially unsigned right-hand operand.

8   For the unary minus operator, do not use an operand of essentially unsigned type. The implemented size of int determines the signedness of the result.

Note that for a bit-field type, if the bit-field is implemented as:

- A Boolean, the bit-field is essentially Boolean.
- Signed or unsigned type, the bit-field is essentially signed or unsigned respectively.

   The type of the bit-field is the smallest type that can represent the bit-field. For instance, the type `stmp` here is essentially 8 bits integer:

```
typedef signed int mybitfield;
typedef struct { mybitfield f1 : 1; } stmp;
```

Additional Message in Report

The *operand_name* operand of the *operator_name* operator is of an inappropriate essential type category *category_name*.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Violation of Rule 10.1, Rationale 2: Inappropriate Operand Types for Operators That Take Essentially Boolean Operands

```
#include<stdbool.h>
extern float f32a;
extern char cha;
extern signed char s8a;
extern unsigned char u8a,u8b,ru8a;
enum enuma { a1, a2, a3 } ena, enb;
extern bool bla, blb, rbla;
void foo(void) {

    rbla = cha && bla;        /* Non-compliant: cha is essentially char  */
    enb = ena ? a1 : a2;      /* Non-compliant: ena is essentially enum  */
    rbla = s8a && bla;        /* Non-compliant: s8a is essentially signed char  */
    ena = u8a ? a1 : a2;      /* Non-compliant: u8a is essentially unsigned char  */
    rbla = f32a && bla;       /* Non-compliant: f32a is essentially float */
    rbla = bla && blb;        /* Compliant */
    ru8a = bla ? u8a : u8b;   /* Compliant */
}
```

In the noncompliant examples, rule 10.1 is violated because:

- The operator && expects only essentially Boolean operands. However, at least one of the operands used has a different type.

- The first operand of ?: is expected to be essentially Boolean. However, a different operand type is used.

> **Note** For Polyspace to detect the rule violation, you must define the type name `boolean` as an effective Boolean type. For more information, see `Effective boolean types (-boolean-types)`.

Violation of Rule 10.1, Rationale 3: Inappropriate Boolean Operands

```
#include<stdbool.h>
enum enuma { a1, a2, a3 } ena;
enum { K1 = 1, K2 = 2 };    /* Essentially signed */
extern char cha, chb;
extern bool bla, blb, rbla;
extern signed char rs8a, s8a;
extern unsigned char u8a;

void foo(void) {

  rbla = bla * blb;       /* Non-compliant - Boolean used as a numeric value */
  rbla = bla > blb;       /* Non-compliant - Boolean used as a numeric value */

  rbla = bla && blb;     /* Compliant */
  rbla = cha > chb;      /* Compliant */
  rbla = ena > a1;       /* Compliant */
  rbla = u8a > 0U;       /* Compliant */
  rs8a = K1 * s8a;       /* Compliant - K1 obtained from anonymous enum */

}
```

In the noncompliant examples, rule 10.1 is violated because the operators * and > do not expect essentially Boolean operands. However, the operands used here are essentially Boolean.

> **Note** For Polyspace to detect the rule violation, you must define the type name `boolean` as an effective Boolean type. For more information, see `Effective boolean types (-boolean-types)`.

Violation of Rule 10.1, Rationale 4: Inappropriate Character Operands

```
extern char rcha, cha, chb;
extern unsigned char ru8a, u8a;

void foo(void) {

  rcha = cha & chb;      /* Non-compliant - char type used as a numeric value */
  rcha = cha << 1;       /* Non-compliant - char type used as a numeric value */

  ru8a = u8a & 2U;       /* Compliant */
  ru8a = u8a << 2U;      /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because the operators & and << do not expect essentially character operands. However, at least one of the operands used here has essentially character type.

Violation of Rule 10.1, Rationale 5: Inappropriate Enum Operands

```
typedef unsigned char boolean;

enum enuma { a1, a2, a3 } rena, ena, enb;

void foo(void) {

  ena--;                 /* Non-Compliant - arithmetic operation with enum type*/
  rena = ena * a1;       /* Non-Compliant - arithmetic operation with enum type*/
  ena += a1;             /* Non-Compliant - arithmetic operation with enum type*/

}
```

In the noncompliant examples, rule 10.1 is violated because the arithmetic operators - -, * and += do not expect essentially enum operands. However, at least one of the operands used here has essentially enum type.

Violation of Rule 10.1, Rationale 6: Inappropriate Signed Operand for Bitwise Operations

```
extern signed char s8a;
extern unsigned char ru8a, u8a;

void foo(void) {

  ru8a = s8a & 2;        /* Non-compliant - bitwise operation on signed type */
  ru8a = 2 << 3U;        /* Non-compliant - shift operation on signed type */

  ru8a = u8a << 2U;      /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because the & and << operations must not be performed on essentially signed operands. However, the operands used here are signed.

Violation of Rule 10.1, Rationale 7: Inappropriate Signed Right Operand for Shift Operations

```
extern signed char s8a;
extern unsigned char ru8a, u8a;

void foo(void) {

  ru8a = u8a << s8a;     /* Non-compliant - shift magnitude uses signed type */
  ru8a = u8a << -1;      /* Non-compliant - shift magnitude uses signed type */

  ru8a = u8a << 2U;      /* Compliant */
  ru8a = u8a << 1;       /* Compliant - exception */

}
```

In the noncompliant examples, rule 10.1 is violated because the operation << does not expect an essentially signed right operand. However, the right operands used here are signed.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 10.2|Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

MISRA C:2012 Rule 10.2

Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations

Description

Rule Definition

Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.

Rationale

Essentially character type expressions are `char` variables. Do not use `char` in arithmetic operations because the data does not represent numeric values.

It is appropriate to use `char` with addition and subtraction operations only in the following cases:

- When one operand of the addition (+) operation is a `char` and the other is a signed or unsigned `char`, `short`, `int`, `long` or `long long`. In this case, the operation returns a `char`.
- When the first operand of the subtraction (-) operation is a `char` and the second is a signed or unsigned `char`, `short`, `int`, `long` or `long long`. If both operands are `char`, the operation returns a *standard* type. Otherwise, the operation returns a `char`.

The above uses allow manipulation of character data such as conversion between lowercase and uppercase characters or conversion between digits and their ordinal values.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

Additional Message in Report

- The *operand_name* operand of the + operator applied to an expression of essentially character type shall have essentially signed or unsigned type.
- The right operand of the - operator applied to an expression of essentially character type shall have essentially signed or unsigned or character type.
- The left operand of the - operator shall have essentially character type if the right operand has essentially character type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Inappropriate use of `char` with Addition and Subtraction Operators

```
#include<stdint.h>
typedef double float64_t;
extern uint8_t u8a;
extern int8_t s8a;
```

```
extern int16_t s16a;
extern int32_t s32a;
extern float64_t fla;

void foo ( void )
{
    char cha;

    s16a = s16a - 'a';  /* Noncompliant*/

    cha = '0' + fla;    /* Noncompliant*/

    cha = cha  + ':';   /* Noncompliant*/
}
```

- You cannot subtract a `char`-type variable from an integer. When you subtract `'a'` from the integer `s16a`, Polyspace raises a violation.

- In addition operations, `char` type variables can only be added to integer type variables. When you add the floating point number `fla` to `'0'`, Polyspace raises a violation.

- The arithmetic operation `cha+':'` is not a conversion from upper to lower case or from digit to cardinal value. Polyspace raises a violation when `char` variables are used in arithmetic expressions.

Permissible use of `char` in Arithmetic Operation

```
#include<stdint.h>
typedef double float64_t;
extern uint8_t u8a;
extern int8_t s8a;
extern int16_t s16a;
extern int32_t s32a;
void foo ( void )
{
    char cha;

    cha = '0' + u8a;     /* Compliant*/

    cha = s8a + '0';     /* Compliant*/

    s32a = cha  - '0';   /* Compliant*/

    cha = '0' - s8a;     /* Compliant*/

    cha++;               /* Compliant*/
}
```

`char` type variables can be used in certain addition or subtraction operations to perform `char` data manipulations. For instance:

- You can add an unsigned integer `u8a` to the `char` type data `'0'` to convert from `'0'` to a different character.

- Similarly, you can add the signed integer `s8a` to `'0'` to perform a desired character conversion.

- You can also subtract `s8a` from the `char` data `'0'`.

- Incrementing and decrementing `char` data is also permissible.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 10.1|Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

MISRA C:2012 Rule 10.3

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category

Description

Rule Definition

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.

Rationale

The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

Polyspace Implementation

The checker raises a violation if an expression is assigned to a variable with a narrower essential type or a different essential type category.

The checker does not raise a violation of this rule:

* If the expression is simply the constant zero.
* If a signed constant is assigned to an unsigned variable but the signed constant has the same representation as its unsigned equivalent. For instance, the checker does not flag statements such as:

  ```
  unsigned int u = 1;
  ```

Additional Message in Report

* The expression is assigned to an object with a different essential type category.
* The expression is assigned to an object with a narrower essential type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also
MISRA C:2012 Rule 10.4|MISRA C:2012 Rule 10.5|MISRA C:2012 Rule 10.6|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Justify Coding Rule Violations Using Code Prover Checks"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

MISRA C:2012 Rule 10.4

Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category

Description

Rule Definition

Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.

Rationale

The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

Polyspace Implementation

The checker raises a violation of this rule if the two operands of an operation have different essential types. The checker message states the types detected on the two sides of the operation.

The checker does not raise a violation of this rule:

- If one of the operands is the constant zero.
- If one of the operands is a signed constant and the other operand is unsigned, and the signed constant has the same representation as its unsigned equivalent.

  For instance, the statement `u8b = u8a + 3;`, where `u8a` and `u8b` are `unsigned char` variables, does not violate the rule because the constants 3 and 3U have the same representation.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Operands with Different Essential Types

```
#define S64_MAX (9223372036854775807LL)
#define S64_MIN (-9223372036854775808LL)
long long input_s64_a, input_s64_b, result_s64;

void my_func(void){
   if (input_s64_a < S64_MIN + input_s64_b) { //Noncompliant: 2 violations
      result_s64 = S64_MIN;
   }
}
```

In this example, the type of `S64_MIN` is essentially unsigned. The value `9223372036854775808LL` is one more than the largest value that can be represented by a 64-bit variable. Therefore, the value

overflows and the result wraps around to a negative value, so `-9223372036854775808LL` is essentially unsigned.

The operation `input_s64_a < S64_MIN + input_s64_b` violates the rule twice.

- The + operation violates the rule. The left operand is essentially unsigned and the right operand is signed.
- The < operation also violates the rule. As a result of type promotion, the result of the + operation is essentially unsigned. Now, the left operand of the < operation is essentially signed but the right operand is essentially unsigned.

Check Information
Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also
`MISRA C:2012 Rule 10.3`|`MISRA C:2012 Rule 10.7`|`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

MISRA C:2012 Rule 10.5

The value of an expression should not be cast to an inappropriate essential type

Description

Rule Definition

The value of an expression should not be cast to an inappropriate essential type.

Rationale

Converting Between Variable Types

		From					
		Boolean	**character**	**enum**	**signed**	**unsigned**	**floating**
To	**Boolean**		Avoid	Avoid	Avoid	Avoid	Avoid
	character	Avoid					Avoid
	enum	Avoid	Avoid	Avoid	Avoid	Avoid	Avoid
	signed	Avoid					
	unsigned	Avoid					
	floating	Avoid	Avoid				

Some inappropriate explicit casts are:

- In C99, the result of a cast of assignment to `_Bool` is always 0 or 1. This result is not necessarily the case when casting to another type which is defined as essentially Boolean.
- A cast to an essential enum type may result in a value that does not lie within the set of enumeration constants for that type.
- A cast from essential Boolean to any other type is unlikely to be meaningful.
- Converting between floating and character types is not meaningful as there is no precise mapping between the two representations.

Some acceptable explicit casts are:

- To change the type in which a subsequent arithmetic operation is performed.
- To truncate a value deliberately.
- To make a type conversion explicit in the interests of clarity.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: The Essential Type Model

Category: Advisory
AGC Category: Advisory

See Also
`MISRA C:2012 Rule 10.3`|`MISRA C:2012 Rule 10.8`|`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

MISRA C:2012 Rule 10.6

The value of a composite expression shall not be assigned to an object with wider essential type

Description

Rule Definition

The value of a composite expression shall not be assigned to an object with wider essential type.

Rationale

A *composite expression* is a nonconstant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (`&`, `|`, `^`)
- Shift (`<<`, `>>`)
- Conditional (`?`, `:`)

If you assign the result of a composite expression to a larger type, the implicit conversion can result in loss of value, sign, precision, or layout.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

Additional Message in Report

The composite expression is assigned to an object with a wider essential type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also
`MISRA C:2012 Rule 10.3`|`MISRA C:2012 Rule 10.7`|`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

MISRA C:2012 Rule 10.7

If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type

Description

Rule Definition

If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed, then the other operand shall not have wider essential type.

Rationale

A *composite expression* is a nonconstant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (`&`, `|`, `^`)
- Shift (`<<`, `>>`)
- Conditional (`?`, `:`)

Restricting implicit conversion on composite expressions mean that sequences of arithmetic operations within expressions must use the same essential type. This restriction reduces confusion and avoids loss of value, sign, precision, or layout. However, this rule does not imply that all operands in an expression are of the same essential type.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

Additional Message in Report

- The right operand shall not have wider essential type than the left operand which is a composite expression.
- The left operand shall not have wider essential type than the right operand which is a composite expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"
"Essential Types in MISRA C:2012 Rules 10.x"

MISRA C:2012 Rule 10.8

The value of a composite expression shall not be cast to a different essential type category or a wider essential type

Description

Rule Definition

The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Rationale

A *composite expression* is a non-constant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (`&`, `|`, `^`)
- Shift (`<<`, `>>`)
- Conditional (`?`, `:`)

Casting to a wider type is not permitted because the result may vary between implementations. Consider this expression:

```
(uint32_t) (u16a +u16b);
```

On a 16-bit machine the addition is performed in 16 bits. The result is wrapped before it is cast to 32 bits. On a 32-bit machine, the addition takes place in 32 bits and preserves high-order bits that are lost on a 16-bit machine. Casting to a narrower type with the same essential type category is acceptable as the explicit truncation of the results always leads to the same loss of information.

For more information on essential types, see `MISRA C:2012 Rule 10.1`.

Polyspace Implementation

The rule checker raises a defect only if the result of a composite expression is cast to a different or wider essential type.

For instance, in this example, a violation is shown in the first assignment to `i` but not the second. In the first assignment, a composite expression `i+1` is directly cast from a signed to an unsigned type. In the second assignment, the composite expression is first cast to the same type and then the result is cast to a different type.

```
typedef int int32_T;
typedef unsigned char uint8_T;
...
...
int32_T i;
i = (uint8_T)(i+1); /* Noncompliant */
i = (uint8_T)((int32_T)(i+1)); /* Compliant */
```

Additional Message in Report

- The value of a composite expression shall not be cast to a different essential type category.
- The value of a composite expression shall not be cast to a wider essential type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Casting to Different or Wider Essential Type

```
extern unsigned short ru16a, u16a, u16b;
extern unsigned int   u32a, ru32a;
extern signed int     s32a, s32b;

void foo(void)
{
  ru16a  = (unsigned short) (u32a + u32a);/* Compliant                        */
  ru16a += (unsigned short) s32a;    /* Compliant - s32a is not composite       */
  ru32a  = (unsigned int) (u16a + u16b);  /* Noncompliant - wider essential type */
}
```

In this example, rule 10.8 is violated in the following cases:

- `s32a` and `s32b` are essentially `signed` variables. However, the result (`s32a + s32b`) is cast to an essentially `unsigned` type.
- `u16a` and `u16b` are essentially `unsigned short` variables. However, the result (`s32a + s32b`) is cast to a wider essential type, `unsigned int`.

Check Information
Group: The Essential Type Model
Category: Required
AGC Category: Advisory

See Also
`MISRA C:2012 Rule 10.5|Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 11.1

Conversions shall not be performed between a pointer to a function and any other type

Description

Rule Definition

Conversions shall not be performed between a pointer to a function and any other type.

Rationale

The rule forbids the following two conversions:

- Conversion from a function pointer to any other type. This conversion causes undefined behavior.
- Conversion from a function pointer to another function pointer, if the function pointers have different argument and return types.

  The conversion is forbidden because calling a function through a pointer with incompatible type results in undefined behavior.

Polyspace Implementation

Polyspace considers both explicit and implicit casts when checking this rule. However, casts from NULL or `(void*)0` do not violate this rule.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Cast between two function pointers

```
typedef void (*fp16) (short n);
typedef void (*fp32) (int n);

#include <stdlib.h>                    /* To obtain macro  NULL */

void func(void) {   /* Exception 1 - Can convert a null pointer
                     * constant into a pointer to a function */
  fp16 fp1 = NULL;                /* Compliant - exception  */
  fp16 fp2 = (fp16) fp1;          /* Compliant */
  fp32 fp3 = (fp32) fp1;          /* Non-compliant */
  if (fp2 != NULL) {}             /* Compliant - exception  */
  fp16 fp4 = (fp16) 0x8000;       /* Non-compliant - integer to
                                   * function pointer */}
```

In this example, the rule is violated when:

- The pointer `fp1` of type `fp16` is cast to type `fp32`. The function pointer types `fp16` and `fp32` have different argument types.

- An integer is cast to type `fp16`.

The rule is not violated when function pointers `fp1` and `fp2` are cast to `NULL`.

Check Information
Group: Pointer Type Conversions
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 11.2

Conversions shall not be performed between a pointer to an incomplete type and any other type

Description

Rule Definition

Conversions shall not be performed between a pointer to an incomplete type and any other type.

Rationale

An incomplete type is a type that does not contain sufficient information to determine its size. For example, the statement `struct s;` describes an incomplete type because the fields of `s` are not defined. The size of a variable of type `s` cannot be determined.

Conversions to or from a pointer to an incomplete type result in undefined behavior. Typically, a pointer to an incomplete type is used to hide the full representation of an object. This encapsulation is broken if another pointer is implicitly or explicitly cast to such a pointer.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Casts from incomplete type

```
#include <stdio.h>
struct s *sp;
struct t *tp;
short  *ip;
struct ct *ctp1;
struct ct *ctp2;


void foo(void) {

    ip = (short *) sp;          /* Non-compliant */
    sp = (struct s *) 1234;     /* Non-compliant */
    tp = (struct t *) sp;       /* Non-compliant */
    ctp1 = (struct ct *) ctp2;  /* Compliant */

    /* You can convert a null pointer constant to
     * a pointer to an incomplete type */
    sp = NULL;                  /* Compliant - exception  */

    /* A pointer to an incomplete type may be converted into void */
    struct s *f(void);
    (void) f();                 /* Compliant - exception  */

}
```

In this example, types `s`, `t` and `ct` are incomplete. The rule is violated when:

- The variable `sp` with an incomplete type is cast to a basic type.
- The variable `sp` with an incomplete type is cast to a different incomplete type `t`.

The rule is not violated when:

- The variable `ctp2` with an incomplete type is cast to the same incomplete type.
- The `NULL` pointer is cast to the variable `sp` with an incomplete type.
- The return value of `f` with incomplete type is cast to `void`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required

See Also

MISRA C:2012 Rule 11.5 | Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 11.3

A cast shall not be performed between a pointer to object type and a pointer to a different object type

Description

Rule Definition

A cast shall not be performed between a pointer to object type and a pointer to a different object type.

Rationale

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of the following types:

* `char`
* `signed char`
* `unsigned char`

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Noncompliant: Cast to Pointer Pointing to Object of Wider Type

```
signed   char *p1;
unsigned int *p2;

void foo(void){
  p2 = ( unsigned int * ) p1;      /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

Noncompliant: Cast to Pointer Pointing to Object of Narrower Type

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );

void foo ( void ){
  unsigned int u = read_value ( );
  unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant  */
  *hi_p = 0;
```

```
  display ( u );
}
```

In this example, u is an `unsigned int` variable. &u is cast to a pointer that points to an object of narrower type, `unsigned short`.

On a big-endian machine, the statement *hi_p = 0 attempts to clear the high bits of the memory location that &u points to. But, from the result of `display(u)`, you might find that the high bits have not been cleared.

Compliant: Cast Adding a Type Qualifier

```
const short *p;
const volatile short *q;
void foo (void){
  q = ( const volatile short * ) p;  /* Compliant */
}
```

In this example, both p and q can point to `short` objects. The cast between them adds a `volatile` qualifier only and is therefore compliant.

Check Information
Group: Pointer Type Conversions
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 11.4|MISRA C:2012 Rule 11.5|MISRA C:2012 Rule 11.8|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 11.4

A conversion should not be performed between a pointer to object and an integer type

Description

Rule Definition

A conversion should not be performed between a pointer to object and an integer type.

Rationale

Conversion between integers and pointers can cause errors or undefined behavior.

- If an integer is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an integer, the resulting value can be outside the allowed range for the integer type.

Polyspace Implementation

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Casts between pointer and integer

```
#include <stdbool.h>

typedef unsigned char     uint8_t;
typedef          char     char_t;
typedef unsigned short     uint16_t;
typedef signed   int       int32_t;

typedef _Bool bool_t;
uint8_t *PORTA = (uint8_t *) 0x0002;            /* Non-compliant */

void foo(void) {

    char_t c = 1;
    char_t *pc = &c;                            /* Compliant */


    uint16_t ui16   = 7U;
    uint16_t *pui16 = &ui16;                     /* Compliant */
    pui16 = (uint16_t *) ui16;                   /* Non-compliant */


    uint16_t *p;
```

```
    int32_t addr = (int32_t) p;                /* Non-compliant */
    bool_t b = (bool_t) p;                     /* Non-compliant */
    enum etag { A, B } e = ( enum etag ) p;    /* Non-compliant */
}
```

In this example, the rule is violated when:

*   The integer `0x0002` is cast to a pointer.

    If the integer defines an absolute address, it is more common to assign the address to a pointer in a header file. To avoid the assignment being flagged, you can then exclude headers files from coding rules checking. For more information, see `Do not generate results for (-do-not-generate-results-for)`.

*   The pointer `p` is cast to integer types such as `int32_t`, `bool_t` or `enum etag`.

The rule is not violated when the address `&ui16` is assigned to a pointer.

Check Information
Group: Pointer Type Conversions
Category: Advisory
AGC Category: Advisory

See Also
`MISRA C:2012 Rule 11.3` | `MISRA C:2012 Rule 11.7` | `MISRA C:2012 Rule 11.9` | `Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 11.5

A conversion should not be performed from pointer to void into pointer to object

Description

Rule Definition

A conversion should not be performed from pointer to void into pointer to object.

Rationale

If a pointer to `void` is cast into a pointer to an object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior. However, such a cast can sometimes be necessary, for example, when using memory allocation functions.

Polyspace Implementation

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Cast from Pointer to void

```
void foo(void) {

    unsigned int  u32a = 0;
    unsigned int  *p32 = &u32a;
    void          *p;
    unsigned int  *p16;

    p   = p32;              /* Compliant - pointer to uint32_t
                            *           into pointer to void */
    p16 = p;               /* Non-compliant */

    p   = (void *) p16;     /* Compliant */
    p32 = (unsigned int *) p; /* Non-compliant */
}
```

In this example, the rule is violated when the pointer `p` of type `void*` is cast to pointers to other types.

The rule is not violated when `p16` and `p32`, which are pointers to non-`void` types, are cast to `void*`.

Check Information
Group: Pointer Type Conversions
Category: Advisory
AGC Category: Advisory

See Also
MISRA C:2012 Rule 11.2|MISRA C:2012 Rule 11.3|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 11.6

A cast shall not be performed between pointer to void and an arithmetic type

Description

Rule Definition

A cast shall not be performed between pointer to void and an arithmetic type.

Rationale

Conversion between integer types and pointers to `void` can cause errors or undefined behavior.

- If an integer type is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an arithmetic type, the resulting value can be outside the allowed range for the type.

Conversion between non-integer arithmetic types and pointers to `void` is undefined.

Polyspace Implementation

Casts or implicit conversions from NULL or `(void*)0` do not generate a warning.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Casts Between Pointer to void and Arithmetic Types

```
void foo(void) {

    void          *p;
    unsigned int  u;
    unsigned short r;

    p = (void *) 0x1234u;          /* Non-compliant - undefined */
    u = (unsigned int) p;          /* Non-compliant - undefined */

    p = (void *) 0;                /* Compliant - Exception */

}
```

In this example, `p` is a pointer to `void`. The rule is violated when:

- An integer value is cast to `p`.
- `p` is cast to an `unsigned int` type.

The rule is not violated if an integer constant with value 0 is cast to a pointer to `void`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 11.7

A cast shall not be performed between pointer to object and a non-integer arithmetic type

Description

Rule Definition

A cast shall not be performed between pointer to object and a non-integer arithmetic type.

Rationale

This rule covers types that are essentially Boolean, character, enum or floating.

- If an essentially Boolean, character or enum variable is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior. If a pointer is cast to one of those types, the resulting value can be outside the allowed range for the type.

- Casts to or from a pointer to a floating type results in undefined behavior.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Casts from Pointer to Non-Integer Arithmetic Types

```
int foo(void) {

    short *p;
    float  f;
    long  *l;

    f = (float)  p;            /* Non-compliant */
    p = (short *) f;           /* Non-compliant */

    l = (long *)  p;           /* Compliant */
}
```

In this example, the rule is violated when:

- The pointer p is cast to `float`.

- A `float` variable is cast to a pointer to `short`.

Casting between a pointer and a non-integerer variable might cause a compilation failure. Polyspace flags such casts.

The rule is not violated when the pointer p is cast to `long*`.

Check Information

Group: Pointer Type Conversions

Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 11.4|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 11.8

A cast shall not remove any const or volatile qualification from the type pointed to by a pointer

Description

Rule Definition

A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.

Rationale

This rule forbids:

- Casts from a pointer to a `const` object to a pointer that does not point to a `const` object.
- Casts from a pointer to a `volatile` object to a pointer that does not point to a `volatile` object.

Such casts violate type qualification. For example, the `const` qualifier indicates the read-only status of an object. If a cast removes the qualifier, the object is no longer read-only.

Polyspace Implementation

Polyspace flags both implicit and explicit conversions that violate this rule.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Casts That Remove Qualifiers

```
void foo(void) {

    /* Cast on simple type */
    unsigned short          x;
    unsigned short * const   cpi = &x;  /* const pointer */
    unsigned short * const  *pcpi;   /* pointer to const pointer */
    unsigned short **ppi;
    const unsigned short    *pci;    /* pointer to const */
    volatile unsigned short *pvi;    /* pointer to volatile  */
    unsigned short          *pi;

    pi = cpi;                         /* Compliant - no cast required */
    pi  = (unsigned short *)  pci;   /* Non-compliant */
    pi  = (unsigned short *)  pvi;   /* Non-compliant */
    ppi = (unsigned short **)pcpi;   /* Non-compliant */
}
```

In this example:

- The variables `pci` and `pcpi` have the `const` qualifier in their type. The rule is violated when the variables are cast to types that do not have the `const` qualifier.

- The variable `pvi` has a `volatile` qualifier in its type. The rule is violated when the variable is cast to a type that does not have the `volatile` qualifier.

Even though `cpi` has a `const` qualifier in its type, the rule is not violated in the statement `p=cpi;`. The assignment does not cause a type conversion because both `p` and `cpi` have type `unsigned short`.

Check Information
Group: Pointer Type Conversions
Category: Required
AGC Category: Required

See Also
`MISRA C:2012 Rule 11.3|Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 11.9

The macro NULL shall be the only permitted form of integer null pointer constant

Description

Rule Definition

The macro NULL shall be the only permitted form of integer null pointer constant.

Rationale

The following expressions allow the use of a null pointer constant:

- Assignment to a pointer
- The == or != operation, where one operand is a pointer
- The ?: operation, where one of the operands on either side of : is a pointer

Using NULL rather than 0 makes it clear that a null pointer constant was intended.

Polyspace Implementation

The checker flags the assignment of the constant zero to pointers, equalities (or inequalities) comparing pointers with the constant zero, and other similar expressions listed in the MISRA C: 2012 documentation.

Following the updates in MISRA C: 2012 Technical Corrigendum 1, the checker allows the use of {0} to initialize aggregates containing only pointers, for instance, arrays of pointers or structures (or unions) with only a pointer field. If an aggregate contains multiple fields, the initialization is still flagged. In these cases, you should use the macro NULL for pointer fields and 0 for integer fields to distinguish between them.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Using 0 in Pointer Assignments and Comparisons

```
void main(void) {

    int *p1 = 0;              /* Non-compliant */
    int *p2 = ( void * ) 0;   /* Compliant     */

#define MY_NULL_1 0    /* Non-compliant */
#define MY_NULL_2 ( void * ) 0

    if ( p1 == MY_NULL_1 )
    { }
    if ( p2 == MY_NULL_2 )    /* Compliant     */
    { }
```

```
}
```

In this example, the rule is violated when the constant 0 is used instead of (`void*`) `0` for pointer assignments and comparisons.

Initialization of Aggregates with Pointer Members Using {0}

```
void init () {
    int *myArray[5] = {0}; //Compliant

    struct structPtr {
        int *ptr;
    } structPtr = {0}; //Compliant

    struct StructIntPtr {
        int data;
        int *ptr;
    } StructIntPtr = {0,0}; //Non-compliant
}
```

Following the updates in MISRA C: 2012 Technical Corrigendum 1, the checker allows the use of {0} to initialize aggregates containing only pointers such as:

- Arrays of pointers, for instance, `myArray`
- Structures with one pointer field only, for instance, `structPtr`

If an aggregate contains multiple fields, such as `StructIntPtr`, the initialization is still flagged. In these cases, you should use the macro NULL for pointer fields and 0 for integer fields to distinguish between them.

Check Information
Group: Pointer Type Conversions
Category: Required
AGC Category: Readability

See Also
MISRA C:2012 Rule 11.4 | Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 12.1

The precedence of operators within expressions should be made explicit

Description

Rule Definition

The precedence of operators within expressions should be made explicit.

Rationale

The C language has a large number of operators and their precedence is not intuitive. Inexperienced programmers can easily make mistakes. Remove any ambiguity by using parentheses to explicitly define operator precedence.

The following table list the MISRA C definition of operator precedence for this rule.

Description	Operator and Operand	Precedence
Primary	identifier, constant, string literal, (expression)	16
Postfix	[] () (function call) . -> ++(post-increment) --(post-decrement) () {}(C99: compound literals)	15
Unary	++(pre-increment) --(pre-decrement) & * + - ~ ! sizeof _Alignof defined (preprocessor)	14
Cast	()	13
Multiplicative	* / %	12
Additive	+ -	11
Bitwise shift	<< >>	10
Relational	<> <= >=	9
Equality	== !=	8
Bitwise AND	&	7
Bitwise XOR	^	6
Bitwise OR	|	5
Logical AND	&&	4
Logical OR	||	3
Conditional	?:	2
Assignment	= *= /= += -= <<= >>= &= ^= |=	1
Comma	,	0

Additional Message in Report

Operand of logical %s is not a primary expression. The precedence of operators within expressions should be made explicit.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Ambiguous Precedence in Multi-Operation Expressions

```
int a, b, c, d, x;

void foo(void) {
  x = sizeof a + b;                      /* Non-compliant - MISRA-12.1 */

  x = a == b ? a : a - b;                /* Non-compliant - MISRA-12.1 */

  x = a <<  b + c ;                      /* Non-compliant - MISRA-12.1 */

  if (a || b && c) { }                   /* Non-compliant - MISRA-12.1 */

  if ( (a>x) && (b>x) || (c>x) )   { }  /* Non-compliant - MISRA-12.1 */
}
```

This example shows various violations of MISRA rule 12.1. In each violation, if you do not know the order of operations, the code could execute unexpectedly.

Correction — Clarify With Parentheses

To comply with this MISRA rule, add parentheses around individual operations in the expressions. One possible solution is shown here.

```
int a, b, c, d, x;

void foo(void) {
  x = sizeof(a) + b;

  x = ( a == b ) ? a : ( a - b );

  x = a << ( b + c );

  if ( ( a || b ) && c) { }

  if ( ((a>x) && (b>x)) || (c>x) ) { }
}
```

Ambiguous Precedence In Preprocessing Expressions

```
# if defined X && X + Y > Z    /* Non-compliant - MISRA-12.1 */
# endif
```

In this example, a violation of MISRA rule 12.1 is shown in preprocessing code. In this violation, if you do not know the correct order of operations, the results can be unexpected and cause problems.

Correction — Clarify with Parentheses

To comply with this MISRA rule, add parentheses around individual operations in the expressions. One possible solution is shown here.

```
# if defined (X) && ( (X + Y) > Z )
# endif
```

Compliant Expressions Without Parentheses

```
int a, b, c, x,i = 0;
struct {int a; } s, *ps, *pp[2];

void foo(void) {
    ps = &s;
    pp[i]-> a;          /* Compliant - no need to write (pp[i])->a */
    *ps++;              /* Compliant - no need to write *( p++ ) */

    x = f ( a + b, c ); /* Compliant - no need to write f ( (a+b),c) */

    x = a, b;           /* Compliant - parsed as ( x = a ), b */

    if (a && b && c ){  /* Compliant - all operators have
                     * the same precedence */
    }
}
```

In this example, the expressions shown have multiple operations. However, these expressions are compliant because operator precedence is already clear.

Check Information

Group: Expressions
Category: Advisory
AGC Category: Advisory

See Also

MISRA C:2012 Rule 12.2|MISRA C:2012 Rule 12.3|MISRA C:2012 Rule 12.4|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 12.2

The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand

Description

Rule Definition

The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.

Rationale

Consider this statement:

```
var = abc << num;
```

If `abc` is a 16-bit integer, then `num` must be in the range 0–15, (nonnegative and less than 16). If `num` is negative or greater than 16, then the shift behavior is undefined.

Polyspace Implementation

Polyspace raises a violation when the right operand of a shift operator exceeds the range defined in this rule. When the right operand is a variable, the violation is raised unless all possible value of the operand remains within the range defined in this rule.

In Polyspace, the numbers that are manipulated in preprocessing directives are 64 bits wide. The valid shift range is between 0 and 63. When bitfields are within a complex expression, Polyspace extends this check onto the bitfield field width or the width of the base type.

Additional Message in Report

- Shift amount is bigger than *size*.
- Shift amount is negative.
- The right operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Avoid Shift Operations That Have Unacceptable Right Operand

```
void foo(void) {
  int i;
  unsigned int BitPack = 0U;

  for (i = 0; i < 32; i++) {
    BitPack |= (1U << ((unsigned int)i));  //Noncompliant
```

```
    }
}
```

In this example, the left operand 1U of the shift operator has an essential type `unsigned char`. Acceptable values for the right operand lies in the range from zero to seven. Because the right operand `i` ranges from zero to 31, Polyspace flags the shift operation.

Check Information

Group: Expressions
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 12.1|Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 12.3

The comma operator should not be used

Description

Rule Definition

The comma operator should not be used.

Rationale

The comma operator can be detrimental to readability. You can often write the same code in another form.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Comma Usage in C Code

```
typedef signed int abc, xyz, jkl;
static void func1 ( abc, xyz, jkl );       /* Compliant - case 1 */
int foo(void)
{
    volatile int rd = 1;                     /* Compliant - case 2*/
    int var=0, foo=0, k=0, n=2, p, t[10];  /* Compliant - case 3*/
    int abc = 0, xyz = abc + 1;            /* Compliant - case 4*/
    int jkl = ( abc + xyz, abc + xyz );    /* Noncompliant - case 1*/
    var = 1, foo += var, n = 3;            /* Noncompliant - case 2*/
    var = (n = 1, foo = 2);                /* Noncompliant - case 3*/
    for ( int *ptr = &t[ 0 ],var = 0 ;
          var < n; ++var, ++ptr){}    /* Noncompliant - case 4*/
    if ((abc,xyz)<0) { return 1; }         /* Noncompliant - case 5*/
}
```

In this example, the code shows various uses of commas in C code.

Noncompliant Cases

Case	Reason for noncompliance
1	When reading the code, it is not immediately obvious what `jkl` is initialized to. For example, you could infer that `jkl` has a value `abc+xyz`, `(abc+xyz)*(abc+xyz)`, `f((abc+xyz),(abc+xyz))`, and so on.
2	When reading the code, it is not immediately obvious whether `foo` has a value 0 or 1 after the statement.
3	When reading the code, it is not immediately obvious what value is assigned to `var`.

Case	Reason for noncompliance
4	When reading the code, it is not immediately obvious which values control the `for` loop.
5	When reading the code, it is not immediately obvious whether the `if` statement depends on `abc`, `xyz`, or both.

Compliant Cases

Case	Reason for compliance
1	Using commas to call functions with variables is allowed.
2	Comma operator is not used.
3 & 4	When using the comma for initialization, the variables and their values are immediately obvious.

Check Information

Group: Expressions
Category: Advisory
AGC Category: Advisory

See Also

MISRA C:2012 Rule 12.1|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 12.4

Evaluation of constant expressions should not lead to unsigned integer wrap-around

Description

Rule Definition

Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Rationale

Unsigned integer expressions do not strictly overflow, but instead wraparound. Although there may be good reasons to use modulo arithmetic at run time, intentional use at compile time is less likely.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Advisory
AGC Category: Advisory

See Also
MISRA C:2012 Rule 12.1|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 12.5

The `sizeof` operator shall not have an operand which is a function parameter declared as "array of type"

Description

Rule Definition

The `sizeof` operator shall not have an operand which is a function parameter declared as "array of type".

This rule comes from MISRA C: 2012 Amendment 1.

Rationale

The `sizeof` operator acting on an array normally returns the array size in bytes. For instance, in the following code, `sizeof(arr)` returns the size of `arr` in bytes.

```
int32_t arr[4];
size_t numberOfElements = sizeof (arr) / sizeof(arr[0]);
```

However, when the array is a function parameter, it degenerates to a pointer. The `sizeof` operator acting on the array returns the corresponding pointer size and not the array size.

The use of `sizeof` operator on an array that is a function parameter typically indicates an unintended programming error.

Additional Message in Report

The `sizeof` operator shall not have an operand which is a function parameter declared as "array of type".

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Incorrect Use of `sizeof` Operator

```
#include <stdint.h>
int32_t glbA[] = { 1, 2, 3, 4, 5 };
void f (int32_t A[4])
{
     uint32_t numElements = sizeof(A) / sizeof(int32_t);   /* Non-compliant */
    uint32_t numElements_glbA = sizeof(glbA) / sizeof(glbA[0]);  /* Compliant */
}
```

In this example, the variable `numElements` always has the same value of 1, irrespective of the number of members that appear to be in the array (4 in this case), because A has type `int32_t *` and not `int32_t[4]`.

The variable `numElements_glbA` has the expected vale of 5 because the `sizeof` operator acts on the global array `glbA`.

Check Information
Group: Expressions
Category: Mandatory
AGC Category: Mandatory

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2017a

MISRA C:2012 Rule 13.1

Initializer lists shall not contain persistent side effects

Description

Rule Definition

Initializer lists shall not contain persistent side effects.

Rationale

C99 permits initializer lists with expressions that can be evaluated only at run-time. However, the order in which elements of the list are evaluated is not defined. If one element of the list modifies the value of a variable which is used in another element, the ambiguity in order of evaluation causes undefined values. Therefore, this rule requires that expressions occurring in an initializer list cannot modify the variables used in them.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Initializers with Persistent Side Effect

```
volatile int v;
int x;
int y;

void f(void) {
    int arr[2] = {x+y,x-y};   /* Compliant */
    int arr2[2] = {v,0};      /* Non-compliant */
    int arr3[2] = {x++,y};    /* Non-compliant */
}
```

In this example, the rule is not violated in the first initialization because the initializer does not modify either x or y. The rule is violated in the other initializations.

- In the second initialization, because v is volatile, the initializer can modify v.

- In the third initialization, the initializer modifies the variable x.

Check Information
Group: Side Effects
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 13.2|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 13.2

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

Description

Rule Definition

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.

Rationale

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

Polyspace Implementation

Polyspace raises a violation if an expression satisfies any of these conditions:

- The same variable is modified more than once in the expression or it is both read and written.
- The expression allows more than one order of evaluation.
- The expression contains a single `volatile` object that occurs multiple times.
- The expression contains more than one `volatile` object.

Because `volatile` objects can change their value at anytime, an expression containing multiple `volatile` variables or multiple instances of the same `volatile` variable might have different results depending on the order of evaluation.

Additional Message in Report

The value of 'XX' depends on the order of evaluation. The value of volatile 'XX' depends on the order of evaluation because of multiple accesses.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])/* Noncompliant  */

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);          /* Compliant */
    COPY_ELEMENT (i++);}
```

In this example, the rule is violated by the statement COPY_ELEMENT(i++) because i++ occurs twice and the order of evaluation of the two expressions is unspecified.

Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
    f ( i++, i );                    /* Non-compliant */
}
```

In this example, the rule is violated because it is unspecified whether the operation i++ occurs before or after the second argument is passed to f. The call f(i++,i) can translate to either f(0,0) or f(0,1).

Multiple volatile Variables in Expression

```
struct {
    volatile float x;
    volatile float y;
} volData;

float xCopy;
float yCopy;
float res, res2;

void function4(void) {
    res = volData.x + volData.y;        //Noncompliant
    res = volData.x * volData.x;      //Noncompliant
    xCopy = volData.x;
    yCopy = volData.y;
    res = xCopy + yCopy;   //Compliant
}
```

In this example, the expression volData.x + volData.y is noncompliant because the expression involves multiple volatile objects. The expression consists of three operations: accessing the value of volData.x, accessing the value of volData.y, and the addition. The values of the volatile fields x and y in the volData structure might change at any time. The value of res might vary depending on which variable is read first. Because the C standard does not specify the order in which the variables are read, the value of res might depend on the hardware and software that you use. Polyspace flags one of the volatile objects in the expression. Similarly, Polyspace flags one of the volatile objects in the expression volData.x * volData.x.

To avoid the violation, assign the volatile variables to nonvolatile temporary variables and use these temporary variables in the expression.

Check Information
Group: Side Effects
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Dir 4.9|MISRA C:2012 Rule 13.1|MISRA C:2012 Rule 13.3|MISRA C:2012 Rule 13.4|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 13.3

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator

Description

Rule Definition

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.

Rationale

The rule is violated if the following happens in the same line of code:

- The increment or decrement operator acts on a variable.
- Another read or write operation is performed on the variable.

For example, the line y=x++ violates this rule. The ++ and = operator both act on x.

Although the operator precedence rules determine the order of evaluation, placing the ++ and another operator in the same line can reduce the readability of the code.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Increment Operator Used in Expression with Other Side Effects

```
int input(void);
int choice(void);
int operation(int, int);

int func() {
    int x = input(), y = input(), res;
    int ch = choice();
    if (choice == -1)
        return(x++);            /* Non-compliant */
    if (choice == 0) {
        res = x++ + y++;            /* Non-compliant */
        return(res);
    }
    else if (choice == 1) {
        x++;                    /* Compliant */
        y++;                    /* Compliant */
        return (x+y);
    }
    else {
        res = operation(x++,y);            /* Non-compliant */
        return(res);
```

```
    }
}
```

In this example, the rule is violated when the expressions containing the ++ operator have side effects other than that caused by the operator. For example, in the expression `return(x++)`, the other side-effect is the `return` operation.

Check Information
Group: Side Effects
Category: Advisory
AGC Category: Readability

See Also
`MISRA C:2012 Rule 13.2` | `Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 13.4

The result of an assignment operator should not be used

Description

Rule Definition

The result of an assignment operator should not be used.

Rationale

The rule is violated if the following happens in the same line of code:

- The assignment operator acts on a variable.
- Another read or operation is performed on the result of the assignment.

For example, the line `a[x]=a[x=y];` violates this rule. The `[]` operator acts on the result of the assignment `x=y`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Result of Assignment Used

```
int x, y, b, c, d;
int a[10];
unsigned int bool_var, false=0, true=1;

int foo(void) {

    x = y;              /* Compliant - x is not used */

    a[x] = a[x = y];  /* Non-compliant - Value of x=y is used */

    if ( bool_var = false )/* Non-compliant - bool_var=false is used */
{}

    if ( bool_var == false ) {}   /* Compliant */

    if ( ( 0u == 0u ) || ( bool_var = true ) )/* Non-compliant */
                    /*- even though (bool_var=true) is not evaluated */
 {}

    if ( ( x = f () ) != 0 )/* Non-compliant - value of x=f() is used */
 {}
    a[b += c] = a[b];/* Non-compliant - value of b += c is used */
    b = c = d = 0; /* Non-compliant - value of d=0 and c=d=0 are used */
```

```
}
```

In this example, the rule is violated when the result of an assignment is used.

Check Information
Group: Side Effects
Category: Advisory
AGC Category: Advisory

See Also
`MISRA C:2012 Rule 13.2`|`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 13.5

The right hand operand of a logical && or || operator shall not contain persistent side effects

Description

Rule Definition

The right hand operand of a logical && or || operator shall not contain persistent side effects.

Rationale

The right operand of an || operator is not evaluated if the left operand is true. The right operand of an && operator is not evaluated if the left operand is false. In these cases, if the right operand modifies the value of a variable, the modification does not take place. Following the operation, if you expect a modified value of the variable, the modification might not always happen.

Polyspace Implementation

- For this rule, Polyspace considers that a function call does not have a persistent side effect if the function body is not present in the same file as the function call.

  If a call to a pure function is flagged, before ignoring this rule violation, make sure that the function has no side effects. For instance, floating-point functions such as abs() seem to only return a value and have no other side effect. However, these functions make use of the FPU Register Stack and can have side-effects in certain architectures, for instance, certain Intel® architectures.

- If the right operand is a volatile variable, Polyspace does not flag this as a rule violation.

Additional Message in Report

The right hand operand of a && operator shall not contain side effects. The right hand operand of a || operator shall not contain side effects.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Right Operand of Logical Operator with Persistent Side Effects

```
int check (int arg) {
    static int count;
    if(arg > 0) {
        count++;                        /* Persistent side effect */
        return 1;
    }
    else
        return 0;
}
```

```
int getSwitch(void);
int getVal(void);

void main(void) {
    int val = getVal();
    int mySwitch = getSwitch();
    int checkResult;

    if(mySwitch && check(val)) {   /* Non-compliant */
    }

    checkResult = check(val);
    if(checkResult && mySwitch) {  /* Compliant */
    }

    if(check(val) && mySwitch) {   /* Compliant */
    }
}
```

In this example, the rule is violated when the right operand of the && operation contains a function call. The function call has a persistent side effect because the static variable `count` is modified in the function body. Depending on `mySwitch`, this modification might or might not happen.

The rule is not violated when the left operand contains a function call. Alternatively, to avoid the rule violation, assign the result of the function call to a variable. Use this variable in the logical operation in place of the function call.

In this example, the function call has the side effect of modifying a `static` variable. Polyspace flags all function calls when used on the right-hand side of a logical && or || operator, even when the function does not have a side effect. Manually inspect your function body to see if it has side effects. If the function does not have side effects, add a comment and justification in your Polyspace result explaining why you retained your code.

Check Information
Group: Side Effects
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 13.6

The operand of the sizeof operator shall not contain any expression which has potential side effects

Description

Rule Definition

The operand of the sizeof operator shall not contain any expression which has potential side effects.

Rationale

The argument of a `sizeof` operator is usually not evaluated at run time. If the argument is an expression, you might wrongly expect that the expression is evaluated.

Polyspace Implementation

The rule is not violated if the argument is a `volatile` variable.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Expressions in `sizeof` Operator

```
#include <stddef.h>
int x;
int y[40];
struct S {
    int a;
    int b;
};
struct S myStruct;

void main() {
    size_t sizeOfType;
    sizeOfType = sizeof(x);         /* Compliant */
    sizeOfType = sizeof(y);         /* Compliant */
    sizeOfType = sizeof(myStruct);  /* Compliant */
    sizeOfType = sizeof(x++);       /* Non-compliant */
}
```

In this example, the rule is violated when the expression `x++` is used as argument of `sizeof` operator.

Check Information
Group: Side Effects
Category: Mandatory
AGC Category: Mandatory

See Also

`MISRA C:2012 Rule 18.8` | `Check MISRA C:2012 (-misra3)`

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 14.1

A loop counter shall not have essentially floating type

Description

Rule Definition

A loop counter shall not have essentially floating type.

Rationale

When using a floating-point loop counter, accumulation of rounding errors can result in a mismatch between the expected and actual number of iterations. This rounding error can happen when a loop step that is not a power of the floating point radix is rounded to a value that can be represented by a float.

Even if a loop with a floating-point loop counter appears to behave correctly on one implementation, it can give a different number of iteration on another implementation.

Polyspace Implementation

If the `for` index is a variable symbol, Polyspace checks that it is not a float.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

`for` Loop Counters

```
int main(void){
    unsigned int counter = 0u;
    int result = 0;
    float foo;

    // Float loop counters
    for(float foo = 0.0f; foo < 1.0f; foo +=0.001f){/* Non-compliant*/
        ++counter;
    }

    float fff = 0.0f;
    for(fff = 0.0f; fff <12.0f; fff += 1.0f){/* Non-compliant*/
        result++;
    }

    // Integer loop count
    for(unsigned int count = 0u; count < 1000u; ++count){/* Compliant */
        foo = (float) count * 0.001f;
    }
}
```

In this example, the three `for` loops show three different loop counters. The first and second `for` loops use float variables as loop counters, and therefore are not compliant. The third loop uses the integer `count` as the loop counter. Even though `count` is used as a float inside the loop, the variable remains an integer when acting as the loop index. Therefore, this `for` loop is compliant.

`while` Loop Counters

```
int main(void){
    unsigned int u32a;
    float foo;

    foo = 0.0f;
    while (foo < 1.0f){/* Non-compliant - foo used as a loop counter */
        foo += 0.001f;
    }

    foo = read_float32();
    do{
        u32a = read_u32();
    }while( ((float)u32a - foo) > 10.0f );
                        /* Compliant - foo doesn't change in the loop */
                        /*  so cannot be a counter */
    return 1;
}
```

This example shows two `while` loops both of which use `foo` in the `while`-loop conditions.

The first `while` loop uses `foo` in the condition and inside the loop. Because `foo` changes, floating-point rounding errors can cause unexpected behavior.

The second `while` loop does not use `foo` inside the loop, but does use `foo` inside the `while`-condition. So `foo` is not the loop counter. The integer `u32a` is the loop counter because it changes inside the loop and is part of the while condition. Because `u32a` is an integer, the rounding error issue is not a concern, making this `while` loop compliant.

Check Information
Group: Control Statement Expressions
Category: Required
AGC Category: Advisory

See Also
MISRA C:2012 Rule 14.2|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 14.2

A for loop shall be well-formed

Description

Rule Definition

A for loop shall be well-formed.

Rationale

The `for` loop provides a flexible looping facility. You can perform other operations besides the loop counter initialization, termination, and increment in the control statement, and increment the loop counter anywhere inside the loop body. However, using a restricted loop format makes your code easier to review and to analyze.

Polyspace Implementation

A for loop consists of a control statement with three clauses and a loop body. The checker raises a violation if:

- The first clause does not contain an initialization (except for when the clause is empty). The checker considers the last assigned variable of the first `for`-loop clause as the loop counter. If the first clause is empty, the checker considers the variable incremented or decremented in the third clause as the loop counter.
- The second clause does not contain a comparison operation involving the loop counter.
- The third clause contains an operation other than incrementing or decrementing the loop counter (separated by a comma from the increment or decrement).
- The loop counter has a data type that is not an integer or a pointer type.
- The loop counter is incremented inside the loop body.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Altering the Loop Counter Inside the Loop

```
void foo(void){

    for(short index=0; index < 5; index++){  /* Non-compliant */
        index = index + 3;       /* Altering the loop counter */
    }
}
```

In this example, the loop counter `index` changes inside the `for` loop. It is hard to determine when the loop terminates.

Correction — Use Another Variable to Terminate Early

One possible correction is to use an extra flag to terminate the loop early.

In this correction, the second clause of the `for` loop depends on the counter value, `index < 5`, and upon an additional flag, `!flag`. With the additional flag, the for loop definition and counter remain readable, and you can escape the loop early.

```
#define FALSE 0
#define TRUE  1

void foo(void){

    int flag = FALSE;

    for(short index=0; (index < 5) && !flag; index++){ /* Compliant */
        if((index % 4) == 0){
            flag = TRUE;        /* allows early termination of loop */
        }
    }
}
```

`for` Loops With Empty Clauses

```
void foo(void){
    for(short index = 0; ; index++) {}   /* Non-compliant */

    for(short index = 0; index < 10;) {} /* Non-compliant */

    short index;
    for(; index < 10;) {}      /* Non-compliant */

    for(; index < 10; index++) {} /* Compliant */

    for(;;){}
        /* Compliant - Exception all three clauses can be empty */
}
```

This example shows `for` loops definitions with a variety of missing clauses. To be compliant, initialize the first clause variable before the `for` loop (line 9). However, you cannot have a `for` loop without the second or third clause.

The one exception is a `for` loop with all three clauses empty, so as to allow for infinite loops.

Check Information
Group: Control Statement Expressions
Category: Required
AGC Category: Readability

See Also
MISRA C:2012 Rule 14.1|MISRA C:2012 Rule 14.3|MISRA C:2012 Rule 14.4|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"

"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 14.3

Controlling expressions shall not be invariant

Description

Rule Definition

Controlling expressions shall not be invariant.

Rationale

If the controlling expression, for example an `if` condition, has a constant value, the non-changing value can point to a programming error.

Polyspace Implementation

The checker flags conditions in `if` or `while` statements or conditions that appear as the first operands of ternary operators (`?:`) if the conditions are invariant, for instance, evaluate always to true or false.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Polyspace Bug Finder flags some violations of MISRA C 14.3 through the `Dead code` and `Useless if` checkers.

Polyspace Code Prover does not use gray code to flag MISRA C 14.3 violations. In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. See "Check for Coding Standard Violations".

Additional Message in Report

- Boolean operations whose results are invariant shall not be permitted.
- Expression is always true.
- Expression is always false.
- Controlling expressions shall not be invariant.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Control Statement Expressions
Category: Required
AGC Category: Required

See Also
```
MISRA C:2012 Rule 2.1|MISRA C:2012 Rule 14.2|Check MISRA C:2012 (-misra3)
```

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 14.4

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Description

Rule Definition

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Rationale

Strong typing requires the controlling expression on an `if` statement or iteration statement to have *essentially Boolean* type.

Polyspace Implementation

Polyspace does not flag integer constants, for example `if(2)`.

The analysis recognizes the Boolean types, `bool` or `_Bool` (defined in `stdbool.h`)

You can also define types that are essentially Boolean using the option `Effective boolean types` (`-boolean-types`).

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Controlling Expression in `if`, `while`, and `for`

```
#include <stdbool.h>
#include <stdlib.h>

#define TRUE 1

typedef _Bool bool_t;
extern bool_t flag;

void foo(void){
    int *p = 1;
    int *q = 0;
    int i = 0;
    while(p){}          /* Non-compliant - p is a pointer */

    while(q != NULL){}  /* Compliant */

    while(TRUE){}       /* Compliant */

    while(flag){}       /* Compliant */
```

```
    if(i){}                 /* Non-compliant - int32_t is not boolean */

    if(i != 0){}        /* Compliant */

    for(int i=-10; i;i++){}   /* Non-compliant - int32_t is not boolean */

    for(int i=0; i<10;i++){}  /* Compliant */
}
```

This example shows various controlling expressions in `while`, `if`, and `for` statements.

The noncompliant statements (the first `while`, `if`, and `for` examples), use a single non-Boolean variable. If you use a single variable as the controlling statement, it must be essentially Boolean (lines 17 and 19). Boolean expressions are also compliant with MISRA.

Check Information
Group: Control Statement Expressions
Category: Required
AGC Category: Advisory

See Also
`MISRA C:2012 Rule 14.2` | `MISRA C:2012 Rule 20.8` | `Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 15.1

The goto statement should not be used

Description

Rule Definition

The goto statement should not be used.

Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of `goto` Statements

```
void foo(void) {
    int i = 0, result = 0;

label1:
    for ( i; i < 5; i++ ) {
        if (i > 2) goto label2;      /* Non-compliant */
    }

label2: {
        result++;
        goto label1;                 /* Non-compliant */
    }
}
```

In this example, the rule is violated when `goto` statements are used.

Check Information
Group: Control Flow
Category: Advisory
AGC Category: Advisory

See Also
MISRA C:2012 Rule 15.2 | MISRA C:2012 Rule 15.3 | MISRA C:2012 Rule 15.4 | Check
MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 15.2

The goto statement shall jump to a label declared later in the same function

Description

Rule Definition

The goto statement shall jump to a label declared later in the same function.

Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand. You can use a forward `goto` statement together with a backward one to implement iterations. Restricting backward `goto` statements ensures that you use only iteration statements provided by the language such as `for` or `while` to implement iterations. This restriction reduces visual complexity of the code.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of Backward goto Statements

```
void foo(void) {
    int i = 0, result = 0;

label1:
    for ( i; i < 5; i++ ) {
        if (i > 2) goto label2;    /* Compliant */
    }

label2: {
        result++;
        goto label1;                /* Non-compliant */
    }
}
```

In this example, the rule is violated when a `goto` statement causes a backward jump to `label1`.

The rule is not violated when a `goto` statement causes a forward jump to `label2`.

Check Information
Group: Control Flow
Category: Required
AGC Category: Advisory

See Also
MISRA C:2012 Rule 15.1|MISRA C:2012 Rule 15.3|MISRA C:2012 Rule 15.4|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 15.3

Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement

Description

Rule Definition

Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.

Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand. Restricting use of `goto` statements to jump between blocks or into nested blocks reduces visual code complexity.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

goto Statements Jump Inside Block

```
void f1(int a) {
    if(a <= 0) {
        goto L2;        /* Non-compliant - L2 in different block*/
    }

    goto L1;            /* Compliant - L1 in same block*/

    if(a == 0) {
        goto L1;        /* Compliant - L1 in outer block*/
    }

    goto L2;            /* Non-compliant - L2 in inner block*/

    L1: if(a > 0) {
            L2:;
    }
}
```

In this example, `goto` statements cause jumps to different labels. The rule is violated when:

- The label occurs in a block different from the block containing the `goto` statement.

  The block containing the label neither encloses nor is enclosed by the current block.

- The label occurs in a block enclosed by the block containing the `goto` statement.

The rule is not violated when:

- The label occurs in the same block as the block containing the `goto` statement..
- The label occurs in a block that encloses the block containing the `goto` statement..

`goto` Statements in `switch` Block

```
void f2 ( int x, int z ) {
    int y = 0;

    switch(x) {
    case 0:
        if(x == y) {
            goto L1;  /* Non-compliant - switch-clauses are treated as blocks */
        }
        break;
    case 1:
        y = x;
        L1: ++x;
        break;
    default:
        break;
    }

}
```

In this example, the label for the `goto` statement appears to occur in a block that encloses the block containing the `goto` statement. However, for the purposes of this rule, the software considers that each `case` statement begins a new block. Therefore, the `goto` statement violates the rule.

Check Information
Group: Control Flow
Category: Required
AGC Category: Advisory

See Also
MISRA C:2012 Rule 15.1|MISRA C:2012 Rule 15.2|MISRA C:2012 Rule 15.4|MISRA C:2012 Rule 16.1|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 15.4

There should be no more than one break or goto statement used to terminate any iteration statement

Description

Rule Definition

There should be no more than one break or goto statement used to terminate any iteration statement.

Rationale

If you use one `break` or `goto` statement in your loop, you have one secondary exit point from the loop. Restricting number of exits from a loop in this way reduces visual complexity of your code.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

break Statements in Inner and Outer Loops

```
volatile int stop;

int func(int *arr, int size, int sat) {
    int i,j;
    int sum = 0;
    for (i=0; i< size; i++) {    /* Compliant  */
        if(sum >= sat)
            break;
        for (j=0; j< i; j++) {  /* Compliant */
            if(stop)
                break;
            sum += arr[j];
        }
    }
}
```

In this example, the rule is not violated in both the inner and outer loop because both loops have one `break` statement each.

break and goto Statements in Loop

```
volatile int stop;

void displayStopMessage();

int func(int *arr, int size, int sat) {
    int i;
    int sum = 0;
    for (i=0; i< size; i++) {
        if(sum >= sat)
            break;
```

```
        if(stop)
            goto L1;    /* Non-compliant  */
        sum += arr[i];
    }

    L1: displayStopMessage();
}
```

In this example, the rule is violated because the `for` loop has one `break` statement and one `goto` statement.

goto Statement in Inner Loop and break Statement in Outer Loop

```
volatile int stop;

void displayMessage();

int func(int *arr, int size, int sat) {
    int i,j;
    int sum = 0;
    for (i=0; i< size; i++) {
        if(sum >= sat)
            break;
        for (j=0; j< i; j++) { /* Compliant */
            if(stop)
                goto L1;  /* Non-compliant */
            sum += arr[i];
        }
    }

    L1: displayMessage();
}
```

In this example, the rule is not violated in the inner loop because you can exit the loop only through the one `goto` statement. However, the rule is violated in the outer loop because you can exit the loop through either the `break` statement or the `goto` statement in the inner loop.

Check Information
Group: Control Flow
Category: Advisory
AGC Category: Advisory

See Also
MISRA C:2012 Rule 15.1|MISRA C:2012 Rule 15.2|MISRA C:2012 Rule 15.3|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 15.5

A function should have a single point of exit at the end

Description

Rule Definition

A function should have a single point of exit at the end.

Rationale

This rule requires that a `return` statement must occur as the last statement in the function body. Otherwise, the following issues can occur:

- Code following a `return` statement can be unintentionally omitted.
- If a function that modifies some of its arguments has early `return` statements, when reading the code, it is not immediately clear which modifications actually occur.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

More Than One `return` Statement in Function

```
#define MAX ((unsigned int)2147483647)
#define NULL (void*)0

typedef unsigned int bool_t;
bool_t false = 0;
bool_t true = 1;

bool_t f1(unsigned short n, char *p) {          /* Non-compliant */
    if(n > MAX) {
        return false;
    }

    if(p == NULL) {
        return false;
    }

    return true;
}
```

In this example, the rule is violated because there are three `return` statements.

Correction — Use Variable to Store Return Value

One possible correction is to store the return value in a variable and return this variable just before the function ends.

```
#define MAX ((unsigned int)2147483647)
#define NULL (void*)0
```

```
typedef unsigned int bool_t;
bool_t false = 0;
bool_t true = 1;
bool_t return_value;

bool_t f2 (unsigned short n, char *p) {          /* Compliant */
    return_value = true;
    if(n > MAX) {
        return_value = false;
    }

    if(p == NULL) {
        return_value = false;
    }

    return return_value;
}
```

Check Information
Group: Control Flow
Category: Advisory
AGC Category: Advisory

See Also
MISRA C:2012 Rule 17.4|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 15.6

The body of an iteration-statement or a selection-statement shall be a compound statement

Description

Rule Definition

The body of an iteration-statement or a selection-statement shall be a compound- statement.

Rationale

If the block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

This checker enforces the practice of adding braces following a selection or iteration statement even for a single line in the body. Later, when more lines are added, the developer adding them does not need to note the absence of braces and include them.

Polyspace Implementation

The checker flags `for` loops where the first token following a `for` statement is not a left brace, for instance:

```
for (i=init_val; i > 0; i--)
   if (arr[i] < 0)
      arr[i] = 0;
```

Similar checks are performed for `if`, `else if`, `else`, `switch`, `for` and `do..while` statements.

The second line of the message on the **Result Details** pane indicates which statement is violating the rule. For instance, in the preceding example, there are two violations. The second line of the message points to the `for` loop for one violation and the `if` condition for another.

Additional Message in Report

- The else keyword shall be followed by either a compound statement, or another if statement.
- An if (expression) construct shall be followed by a compound statement.
- The statement forming the body of a while statement shall be a compound statement.
- The statement forming the body of a do ... while statement shall be a compound statement.
- The statement forming the body of a for statement shall be a compound statement.
- The statement forming the body of a switch statement shall be a compound statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Iteration Block

```
int data_available = 1;
void f1(void) {
    while(data_available)                /* Non-compliant */
        process_data();

    while(data_available) {              /* Compliant */
        process_data();
    }
}
```

In this example, the second `while` block is enclosed in braces and does not violate the rule.

Nested Selection Statements

```
#include<stdbool.h>
void f1(bool flag_1, bool flag_2) {
    if(flag_1)                           /* Non-compliant */
        if(flag_2)                       /* Non-compliant */
            action_1();
    else                                 /* Non-compliant */
            action_2();
}
```

In this example, the rule is violated because the `if` or `else` blocks are not enclosed in braces. Unless indented as above, it is easy to associate the `else` statement with the inner `if`.

Correction — Place Selection Statement Block in Braces

One possible correction is to enclose each block associated with an `if` or `else` statement in braces.

```
#include<stdbool.h>
void f1(bool flag_1, bool flag_2) {
    if(flag_1) {                         /* Compliant */
        if(flag_2) {                      /* Compliant */
            action_1();
        }
    }
    else {                               /* Compliant */
        action_2();
    }
}
```

Spurious Semicolon After Iteration Statement

```
#include<stdbool.h>
void f1(bool flag_1) {
    while(flag_1);                       /* Non-compliant */
    {
        flag_1 = action_1();
    }
}
```

In this example, the rule is violated even though the `while` statement is followed by a block in braces. The semicolon following the `while` statement causes the block to dissociated from the `while` statement.

The rule helps detect such spurious semicolons.

Check Information
Group: Control Flow
Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 15.7

All if ... else if constructs shall be terminated with an else statement

Description

Rule Definition

All if ... else if constructs shall be terminated with an else statement.

Rationale

Unless there is a terminating `else` statement in an `if...elseif...else` construct, during code review, it is difficult to tell if you considered all possible results for the `if` condition.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Missing else Block

```
#include<stdbool.h>
void action_1(void);
void action_2(void);

void f1(bool flag_1, bool flag_2) {
    if(flag_1) {
        action_1();
    }
    else if(flag_2) {/* Non-compliant */
        action_2();
    }
}
```

In this example, the rule is violated because the `if ... else if` construct does not have a terminating `else` block.

Correction — Add else Block

To avoid the rule violation, add a terminating `else` block. This `else` block can, for instance, handle exceptions or be empty.

```
#include<stdbool.h>
bool ERROR = 0;
void action_1(void);
void action_2(void);

void f1(bool flag_1, bool flag_2) {
    if(flag_1) {
        action_1();
    }
    else if(flag_2) {
```

```
        action_2();
    }else{
        // Can be empty
        ERROR = 1;
    }
}
```

Check Information
Group: Control Flow
Category: Required
AGC Category: Readability

See Also
MISRA C:2012 Rule 16.5|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 16.1

All switch statements shall be well-formed

Description

Rule Definition

All switch statements shall be well-formed

Rationale

The syntax for switch statements in C is not particularly rigorous and can allow complex, unstructured behavior. This rule and other rules impose a simple consistent structure on the `switch` statement.

Polyspace Implementation

Following the MISRA specifications, the coding rules checker also raises a violation of rule 16.1 if a `switch` statement violates one of these rules: 16.2, 16.3, 16.4, 16.5 or 16.6.

Additional Message in Report

All messages in report file begin with "MISRA-C switch statements syntax normative restriction."

- Initializers shall not be used in switch clauses.
- The child statement of a switch shall be a compound statement.
- All switch clauses shall appear at the same level.
- A switch clause shall only contain switch labels and switch clauses, and no other code.
- A switch statement shall only contain switch labels and switch clauses, and no other code.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also
MISRA C:2012 Rule 15.3 | MISRA C:2012 Rule 16.2 | MISRA C:2012 Rule 16.3 | MISRA C:2012 Rule 16.4 | MISRA C:2012 Rule 16.5 | MISRA C:2012 Rule 16.6 | Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 16.2

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Description

Rule Definition

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Rationale

The C Standard permits placing a switch label (for instance, `case` or `default`) before any statement contained in the body of a switch statement. This flexibility can lead to unstructured code. To prevent unstructured code, make sure a switch label appears only at the outermost level of the body of a switch statement.

Additional Message in Report

All messages in report file begin with "MISRA-C switch statements syntax normative restriction."

- Initializers shall not be used in switch clauses.
- The child statement of a switch shall be a compound statement.
- All switch clauses shall appear at the same level.
- A switch clause shall only contain switch labels and switch clauses, and no other code.
- A switch statement shall only contain switch labels and switch clauses, and no other code.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also
MISRA C:2012 Rule 16.1|Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 16.3

An unconditional break statement shall terminate every switch-clause

Description

Rule Definition

An unconditional break statement shall terminate every switch-clause

Rationale

A *switch-clause* is a case containing at least one statement. Two consecutive labels without an intervening statement is compliant with MISRA.

If you fail to end your switch-clauses with a break statement, then control flow "falls" into the next statement. This next statement can be another switch-clause, or the end of the switch. This behavior is sometimes intentional, but more often it is an error. If you add additional cases later, an unterminated switch-clause can cause problems.

Polyspace Implementation

Polyspace raises a warning for each noncompliant `case` clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also
MISRA C:2012 Rule 16.1 | Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 16.4

Every switch statement shall have a default label

Description

Rule Definition

Every switch statement shall have a default label

Rationale

The requirement for a `default` label is defensive programming. Even if your switch covers all possible values, there is no guarantee that the input takes one of these values. Statements following the `default` label take some appropriate action. If the `default` label requires no action, use comments to describe why there are no specific actions.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Switch Statement Without `default`

```
short func1(short xyz){

    switch(xyz){        /* Non-compliant - default label is required */
        case 0:
            ++xyz;
            break;
        case 1:
        case 2:
            break;
    }
    return xyz;
}
```

In this example, the switch statement does not include a `default` label, and is therefore noncompliant.

Correction — Add `default` With Error Flag

One possible correction is to use the `default` label to flag input errors. If your switch-clauses cover all expected input, then the default cases flags any input errors.

```
short func1(short xyz){
int errorflag = 0;
    switch(xyz){        /* Compliant */
        case 0:
            ++xyz;
            break;
        case 1:
        case 2:
```

```
            break;
        default:
            errorflag = 1;
            break;
    }
    if (errorflag == 1)
        return errorflag;
    else
        return xyz;
}
```

Switch Statement for Enumerated Inputs

```
enum Colors{
    RED, GREEN, BLUE
};

enum Colors func2(enum Colors color){
    enum Colors next;

    switch(color){      /* Non-compliant - default label is required */
        case RED:
            next = GREEN;
            break;
        case GREEN:
            next = BLUE;
            break;
        case BLUE:
            next = RED;
            break;
    }
    return next;
}
```

In this example, the switch statement does not include a `default` label, and is therefore noncompliant. Even though this switch statement handles all values of the enumeration, there is no guarantee that color takes one of the those values.

Correction — Add default

To be compliant, add the `default` label to the end of your switch. You can use this case to flag unexpected inputs.

```
enum Colors{
    RED, GREEN, BLUE, ERROR
};

enum Colors func2(enum Colors color){
    enum Colors next;

    switch(color){      /* Compliant */
        case RED:
            next = GREEN;
            break;
        case GREEN:
            next = BLUE;
            break;
        case BLUE:
```

```
            next = RED;
            break;
        default:
            next = ERROR;
            break;
    }

    return next;
}
```

Check Information
Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also
MISRA C:2012 Rule 2.1|MISRA C:2012 Rule 16.1|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 16.5

A default label shall appear as either the first or the last switch label of a switch statement

Description

Rule Definition

A default label shall appear as either the first or the last switch label of a switch statement.

Rationale

Using this rule, you can easily locate the `default` label within a `switch` statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Default Case in `switch` Statements

```
void foo(int var){

    switch(var){
        default:   /* Compliant - default is the first label */
        case 0:
            ++var;
            break;
        case 1:
        case 2:
            break;
    }

    switch(var){
        case 0:
            ++var;
            break;
        default:    /* Non-compliant - default is mixed with the case labels */
        case 1:
        case 2:
            break;
    }

    switch(var){
        case 0:
            ++var;
            break;
        case 1:
        case 2:
        default:    /* Compliant - default is the last label */
            break;
    }
```

```
    switch(var){
        case 0:
            ++var;
            break;
        case 1:
        case 2:
            break;
        default:        /* Compliant - default is the last label */
            var = 0;
            break;
    }
}
```

This example shows the same switch statement several times, each with `default` in a different place. As the first, third, and fourth switch statements show, `default` must be the first or last label. `default` can be part of a compound switch-clause (for instance, the third `switch` example), but it must be the last listed.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 15.7`|`MISRA C:2012 Rule 16.1`|`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 16.6

Every switch statement shall have at least two switch-clauses

Description

Rule Definition

Every switch statement shall have at least two switch-clauses.

Rationale

A switch statement with a single path is redundant and can indicate a programming error.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also
MISRA C:2012 Rule 16.1|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 16.7

A switch-expression shall not have essentially Boolean type

Description

Rule Definition

A switch-expression shall not have essentially Boolean type

Rationale

The C Standard requires the controlling expression to a `switch` statement to have an integer type. Because C implements Boolean values with integer types, it is possible to have a Boolean expression control a `switch` statement. For controlling flow with Boolean types, an `if-else` construction is more appropriate.

Polyspace Implementation

The analysis recognizes the Boolean types, `bool` or `_Bool` (defined in `stdbool.h`)

You can also define types that are essentially Boolean using the option `Effective boolean types` (`-boolean-types`).

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Switch Statements
Category: Required
AGC Category: Advisory

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 17.1

The features of <stdarg.h> shall not be used

Description

Rule Definition

The features of <stdarg.h> shall not be used..

Rationale

The rule forbids use of `va_list`, `va_arg`, `va_start`, `va_end`, and `va_copy`.

You can use these features in ways where the behavior is not defined in the Standard. For instance:

- You invoke `va_start` in a function but do not invoke the corresponding `va_end` before the function block ends.
- You invoke `va_arg` in different functions on the same variable of type `va_list`.
- `va_arg` has the syntax `type va_arg (va_list ap, type)`.

  You invoke `va_arg` with a `type` that is incompatible with the actual type of the argument retrieved from `ap`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of `va_start`, `va_list`, `va_arg`, and `va_end`

```
#include<stdarg.h>
void f2(int n, ...) {
    int i;
    double val;
    va_list vl;                         /* Non-compliant */

    va_start(vl, n);                    /* Non-compliant */

    for(i = 0; i < n; i++)
    {
        val = va_arg(vl, double);        /* Non-compliant */
    }

    va_end(vl);                         /* Non-compliant */
}
```

In this example, the rule is violated because `va_start`, `va_list`, `va_arg` and `va_end` are used.

Undefined Behavior of `va_arg`

```
#include <stdarg.h>
```

```
void h(va_list ap) {                        /* Non-compliant */
    double y;

    y = va_arg(ap, double );                 /* Non-compliant */
}

void g(unsigned short n, ...) {
    unsigned int x;
    va_list ap;                             /* Non-compliant */

    va_start(ap, n);                        /* Non-compliant */
    x = va_arg(ap, unsigned int);        /* Non-compliant */

    h(ap);

    /* Undefined - ap is indeterminate because va_arg used in h () */
    x = va_arg(ap, unsigned int);        /* Non-compliant */

}

void f(void) {
    /* undefined - uint32_t:double type mismatch when g uses va_arg () */
    g(1, 2.0, 3.0);
}
```

In this example, `va_arg` is used on the same variable `ap` of type `va_list` in both functions `g` and `h`. In `g`, the second argument is `unsigned int` and in `h`, the second argument is `double`. This type mismatch causes undefined behavior.

Check Information
Group: Function
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 17.2

Functions shall not call themselves, either directly or indirectly

Description

Rule Definition

Functions shall not call themselves, either directly or indirectly.

Rationale

Variables local to a function are stored in the call stack. If a function calls itself directly or indirectly several times, the available stack space can be exceeded, causing serious failure. Unless the recursion is tightly controlled, it is difficult to determine the maximum stack space required.

Polyspace Implementation

The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.

You can calculate the total number of recursion cycles using the code complexity metric `Number of Recursions`.

Additional Message in Report

Message in Report: Function XX is called indirectly by YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Direct and Indirect Recursion

```
void foo1( void ) {      /* Non-compliant - Indirect recursion foo1->foo2->foo1... */
    foo2();
    foo1();               /* Non-compliant - Direct recursion */
}

void foo2( void ) { /* Non-compliant - Indirect recursion foo2->foo1->foo2... */
    foo1();
}
```

In this example, the rule is violated because of:

- Direct recursion `foo1` → `foo1`.
- Indirect recursion `foo1` → `foo2` → `foo1`.
- Indirect recursion `foo2` → `foo1` → `foo2`.

Check Information
Group: Function

Category: Required
AGC Category: Required

See Also
`Number of Recursions`|`Number of Direct Recursions`|`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 17.3

A function shall not be declared implicitly

Description

Rule Definition

A function shall not be declared implicitly.

Rationale

An implicit declaration occurs when you call a function before declaring or defining it. When you declare a function explicitly before calling it, the compiler can match the argument and return types with the parameter types in the declaration. If an implicit declaration occurs, the compiler makes assumptions about the argument and return types. For instance, it assumes a return type of `int`. The assumptions might not agree with what you expect and cause undesired type conversions.

Additional Message in Report

Function 'XX' has no complete visible prototype at call.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Function Not Declared Before Call

```
#include <math.h>

extern double power3 (double val, int exponent);
int getChoice(void);

double func() {
    double res;
    int ch = getChoice();
    if(ch == 0) {
        res = power(2.0, 10);    /* Non-compliant */
    }
    else if( ch==1) {
        res = power2(2.0, 10);   /* Non-compliant */
    }
    else {
        res = power3(2.0, 10);   /* Compliant */
        return res;
    }
}

double power2 (double val, int exponent) {
    return (pow(val, exponent));
}
```

In this example, the rule is violated when a function that is not declared is called in the code. Even if a function definition exists later in the code, the rule violation occurs.

The rule is not violated when the function is declared before it is called in the code. If the function definition exists in another file and is available only during the link phase, you can declare the function in one of the following ways:

- Declare the function with the `extern` keyword in the current file.
- Declare the function in a header file and include the header file in the current file.

Check Information
Group: Function
Category: Mandatory
AGC Category: Mandatory

See Also
MISRA C:2012 Rule 8.2|MISRA C:2012 Rule 8.4|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 17.4

All exit paths from a function with non-void return type shall have an explicit return statement with an expression

Description

Rule Definition

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Rationale

If a non-`void` function does not explicitly return a value but the calling function uses the return value, the behavior is undefined. To prevent this behavior:

1 You must provide `return` statements with an explicit expression.

2 You must ensure that during run time, at least one `return` statement executes.

Additional Message in Report

Missing return value for non-void function 'XX'.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Missing Return Statement Along Certain Execution Paths

```
int absolute(int v) {
    if(v < 0) {
        return v;
    }
} // Non-compliant
```

In this example, the rule is violated because a `return` statement does not exist on all execution paths. If `v >= 0`, then the control returns to the calling function without an explicit return value.

Return Statement Without Explicit Expression

```
#define SIZE 10
int table[SIZE];

unsigned short lookup(unsigned short v) {
    if((v < 0) || (v > SIZE)) {
        return; // Non-compliant
    }
    return table[v];
}
```

In this example, the rule is violated because the `return` statement in the `if` block does not have an explicit expression.

Check Information

Group: Function
Category: Mandatory
AGC Category: Mandatory

See Also

MISRA C:2012 Rule 15.5 | Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 17.5

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements

Description

Rule Definition

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

Rationale

If you use an array declarator for a function parameter instead of a pointer, the function interface is clearer because you can state the minimum expected array size. If you do not state a size, the expectation is that the function can handle an array of any size. In such cases, the size value is typically another parameter of the function, or the array is terminated with a sentinel value.

However, it is legal in C to specify an array size but pass an array of smaller size. This rule prevents you from passing an array of size smaller than the size you declared.

Additional Message in Report

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

The argument type has *actual_size* elements whereas the parameter type expects *expected_size* elements.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Incorrect Array Size Passed to Function

```
void func(int arr[4]);

int main() {
    int arrSmall[3] = {1,2,3};
    int arr[4] = {1,2,3,4};
    int arrLarge[5] ={1,2,3,4,5};

    func(arrSmall);      /* Non-compliant */
    func(arr);           /* Compliant */
    func(arrLarge);      /* Compliant */

    return 0;
}
```

In this example, the rule is violated when `arrSmall`, which has size 3, is passed to `func`, which expects at least 4 elements.

Check Information

Group: Functions
Category: Advisory
AGC Category: Readability

See Also

`Check MISRA C:2012 (-misra3)|MISRA C:2012 Rule 17.6`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2015b

MISRA C:2012 Rule 17.6

The declaration of an array parameter shall not contain the static keyword between the [ ]

Description

Rule Definition

The declaration of an array parameter shall not contain the static keyword between the [ ].

Rationale

If you use the `static` keyword within `[]` for an array parameter of a function, you can inform a C99 compiler that the array contains a minimum number of elements. The compiler can use this information to generate efficient code for certain processors. However, in your function call, if you provide less than the specified minimum number, the behavior is not defined.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of `static` Keyword Within [] in Array Parameter

```
extern int arr1[20];
extern int arr2[10];


unsigned int total (unsigned int n,
                    unsigned int arr[static 20]) { // Non-compliant

    unsigned int i;
    unsigned int sum = 0;

    for (i=0U; i < n; i++) {
        sum+= arr[i];
    }

    return sum;
}

void func (void) {
    int res, res2;
    res = total (10U, arr1); //Undefined behavior
    res2 = total (20U, arr2);
}
```

In this example, the rule is violated when the `static` keyword is used within `[]` in the array parameter of function `total`. Even if you call `total` with array arguments where the behavior is well-defined, the rule violation occurs.

Check Information
Group: Function
Category: Mandatory
AGC Category: Mandatory

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 17.7

The value returned by a function having non-void return type shall be used

Description

Rule Definition

The value returned by a function having non-void return type shall be used.

Rationale

You can unintentionally call a function with a non-`void` return type but not use the return value. Because the compiler allows the call, you might not catch the omission. This rule forbids calls to a non-`void` function where the return value is not used. If you do not intend to use the return value of a function, explicitly cast the return value to `void`.

Polyspace Implementation

The checker flags functions with non-`void` return if the return value is not used or not explicitly cast to a `void` type.

The checker does not flag the functions `memcpy`, `memset`, `memmove`, `strcpy`, `strncpy`, `strcat`, `strncat` because these functions simply return a pointer to their first arguments.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Used and Unused Return Values

```
unsigned int cutOff(unsigned int val) {
    if (val > 10 && val < 100) {
        return val;
    }
    else {
        return 0;
    }
}

unsigned int getVal(void);

void func2(void) {
    unsigned int val = getVal(), res;
    cutOff(val);            /* Non-compliant */
    res = cutOff(val);      /* Compliant */
    (void)cutOff(val);      /* Compliant */
}
```

In this example, the rule is violated when the return value of `cutOff` is not used subsequently.

The rule is not violated when the return value is:

- Assigned to another variable.
- Explicitly cast to `void`.

Check Information
Group: Function
Category: Required
AGC Category: Readability

See Also
`MISRA C:2012 Rule 2.2|Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 17.8

A function parameter should not be modified

Description

Rule Definition

A function parameter should not be modified.

Rationale

When you modify a parameter, the function argument corresponding to the parameter is not modified. However, you or another programmer unfamiliar with C can expect by mistake that the argument is also modified when you modify the parameter.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Function Parameter Modified

```c
int input(void);

void func(int param1, int* param2) {

    param1 = input();   /* Non-compliant */
    *param2 = input();  /* Compliant */
}
```

In this example, the rule is violated when the parameter `param1` is modified.

The rule is not violated when the parameter is a pointer `param2` and `*param2` is modified.

Check Information
Group: Functions
Category: Advisory
AGC Category: Readability

See Also
Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2015b

MISRA C:2012 Rule 18.1

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

Description

Rule Definition

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

Rationale

Using an invalid array subscript can lead to erroneous behavior of the program. Run-time derived array subscripts are especially troublesome because they cannot be easily checked by manual review or static analysis.

The C Standard defines the creation of a pointer to one beyond the end of the array. The rule permits the C Standard. Dereferencing a pointer to one beyond the end of an array causes undefined behavior and is noncompliant.

Polyspace Implementation

Polyspace flags this rule during the analysis as:

- Bug Finder — `Array access out-of-bounds` and `Pointer access out-of-bounds`.
- Code Prover — `Illegally dereferenced pointer` and `Out of bounds array index`.

Bug Finder and Code Prover check this rule differently and can show different results for this rule. In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. See "Check for Coding Standard Violations".

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Pointers and Arrays
Category: Required
AGC Category: Required

See Also
`MISRA C:2012 Dir 4.1`|`MISRA C:2012 Rule 18.4`|`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 18.2

Subtraction between pointers shall only be applied to pointers that address elements of the same array

Description

Rule Definition

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Rationale

This rule applies to expressions of the form `pointer_expression1 - pointer_expression2`. The behavior is undefined if `pointer_expression1` and `pointer_expression2`:

- Do not point to elements of the same array,

- Or do not point to the element one beyond the end of the array.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Subtracting Pointers

```
#include <stddef.h>
#include <stdint.h>

void f1 (int32_t *ptr)
{
    int32_t a1[10];
    int32_t a2[10];
    int32_t *p1 = &a1[ 1];
    int32_t *p2 = &a2[10];
    ptrdiff_t diff1, diff2, diff3;

    diff1 =  p1 - a1;   // Compliant
    diff2 =  p2 - a2;   // Compliant
    diff3 =  p1 - p2;   // Non-compliant
}
```

In this example, the three subtraction expressions show the difference between compliant and noncompliant pointer subtractions. The `diff1` and `diff2` subtractions are compliant because the pointers point to the same array. The `diff3` subtraction is not compliant because `p1` and `p2` point to different arrays.

Check Information

Group: Pointers and Arrays

Category: Required
AGC Category: Required

See Also
MISRA C:2012 Dir 4.1 | MISRA C:2012 Rule 18.4 | Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 18.3

The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object

Description

Rule Definition

The relational operators >, >=, <, and <= shall not be applied to objects of pointer type except where they point into the same object.

Rationale

If two pointers do not point to the same object, comparisons between the pointers produces undefined behavior.

You can address the element beyond the end of an array, but you cannot access this element.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Pointer and Array Comparisons

```
void f1(void){
    int arr1[10];
    int arr2[10];
    int *ptr1 = arr1;

    if(ptr1 < arr2){}    /* Non-compliant */
    if(ptr1 < arr1){}    /* Compliant */
}
```

In this example, `ptr1` is a pointer to `arr1`. To be compliant with rule 18.3, you can compare only `ptr1` with `arr1`. Therefore, the comparison between `ptr1` and `arr2` is noncompliant.

Structure Comparisons

```
struct limits{
  int lower_bound;
  int upper_bound;
};

void func2(void){
    struct limits lim_1 = { 2, 5 };
    struct limits lim_2 = { 10, 5 };

    if(&lim_1.lower_bound <= &lim_2.upper_bound){}  /* Non-compliant *
    if(&lim_1.lower_bound <= &lim_1.upper_bound){}  /* Compliant */
}
```

This example defines two `limits` structures, `lim1` and `lim2`, and compares the elements. To be compliant with rule 18.3, you can compare only the structure elements within a structure. The first comparison compares the `lower_bound` of `lim1` and the `upper_bound` of `lim2`. This comparison is noncompliant because the `lim_1.lower_bound` and `lim_2.upper_bound` are elements of two different structures.

Check Information
Group: Pointers and Arrays
Category: Required
AGC Category: Required

See Also
`MISRA C:2012 Dir 4.1|Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 18.4

The +, -, += and -= operators should not be applied to an expression of pointer type

Description

Rule Definition

The +, -, += and -= operators should not be applied to an expression of pointer type.

Rationale

The preferred form of pointer arithmetic is using the array subscript syntax `ptr[expr]`. This syntax is clear and less prone to error than pointer manipulation. With pointer manipulation, any explicitly calculated pointer value has the potential to access unintended or invalid memory addresses. Array indexing can also access unintended or invalid memory, but it is easier to review.

To a new C programmer, the expression `ptr+1` can be mistakenly interpreted as one plus the address of `ptr`. However, the new memory address depends on the size, in bytes, of the pointer's target. This confusion can lead to unexpected behavior.

When used with caution, pointer manipulation using ++ can be more natural (for instance, sequentially accessing locations during a memory test).

Polyspace Implementation

Polyspace flags operations on pointers, for example, `Pointer + Integer`, `Integer + Pointer`, `Pointer - Integer`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Pointers and Array Expressions

```
void fun1(void){
    unsigned char arr[10];
    unsigned char *ptr;
    unsigned char index = 0U;

    index = index + 1U;    /* Compliant - rule only applies to pointers */

    arr[index] = 0U;       /* Compliant */
    ptr = &arr[5];         /* Compliant */
    ptr = arr;
    ptr++;                 /* Compliant - increment operator not + */
    *(ptr + 5) = 0U;       /* Non-compliant */
    ptr[5] = 0U;           /* Compliant */
}
```

This example shows various operations with pointers and arrays. The only operation in this example that is noncompliant is using the + operator directly with a pointer (line 12).

Adding Array Elements Inside a `for` Loop

```
void fun2(void){
    unsigned char array_2_2[2][2] = {{1U, 2U}, {4U, 5U}};
    unsigned char i = 0U;
    unsigned char j = 0U;
    unsigned char sum = 0U;

    for(i = 0u; i < 2U; i++){
        unsigned char *row = array_2_2[ i ];

        for(j = 0u; j < 2U; j++){
            sum += row[ j ];                    /* Compliant */
        }
    }
}
```

In this example, the second `for` loop uses the array pointer `row` in an arithmetic expression. However, this usage is compliant because it uses the array index form.

Pointers and Array Expressions

```
void fun3(unsigned char *ptr1, unsigned char ptr2[ ]){
    ptr1++;                 /* Compliant */
    ptr1 = ptr1 - 5;        /* Non-compliant */
    ptr1 -= 5;              /* Non-compliant */
    ptr1[2] = 0U;           /* Compliant */

    ptr2++;                 /* Compliant */
    ptr2 = ptr2 + 3;        /* Non-compliant */
    ptr2 += 3;              /* Non-compliant */
    ptr2[3] = 0U;           /* Compliant */
}
```

This example shows the offending operators used on pointers and arrays. Notice that the same types of expressions are compliant and noncompliant for both pointers and arrays.

If `ptr1` does not point to an array with at least six elements, and `ptr2` does not point to an array with at least 4 elements, this example violates rule 18.1.

Check Information
Group: Pointers and Arrays
Category: Advisory
AGC Category: Advisory

See Also
MISRA C:2012 Rule 18.1|MISRA C:2012 Rule 18.2|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 18.5

Declarations should contain no more than two levels of pointer nesting

Description

Rule Definition

Declarations should contain no more than two levels of pointer nesting.

Rationale

The use of more than two levels of pointer nesting can seriously impair the ability to understand the behavior of the code. Avoid this usage.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Pointer Nesting

```
typedef char *INTPTR;

void function(char ** arrPar[ ])    /* Non-compliant - 3 levels */
{
    char   **  obj2;            /* Compliant */
    char   *** obj3;            /* Non-compliant */
    INTPTR *   obj4;            /* Compliant */
    INTPTR * const * const obj5;    /* Non-compliant */
    char   ** arr[10];         /* Compliant */
    char   ** (*parr)[10];     /* Compliant */
    char   *  (**pparr)[10];      /* Compliant */
}

struct s{
    char *   s1;               /* Compliant */
    char **  s2;               /* Compliant */
    char *** s3;               /* Non-compliant */
};

struct s *   ps1;          /* Compliant */
struct s **  ps2;          /* Compliant */
struct s *** ps3;          /* Non-compliant */

char **  (  *pfunc1)(void);      /* Compliant */
char **  ( **pfunc2)(void);      /* Compliant */
char **  (***pfunc3)(void);      /* Non-compliant */
char *** ( **pfunc4)(void);      /* Non-compliant */
```

This example shows various pointer declarations and nesting levels. Any pointer with more than two levels of nesting is considered noncompliant.

Check Information
Group: Pointers and Arrays
Category: Advisory
AGC Category: Readability

See Also
```
Check MISRA C:2012 (-misra3)
```

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 18.6

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist

Description

Rule Definition

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.

Rationale

The address of an object becomes indeterminate when the lifetime of that object expires. Any use of an indeterminate address results in undefined behavior.

Polyspace Implementation

Polyspace flags a violation when assigning an address to a global variable, returning a local variable address, or returning a parameter address.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Address of Local Variables

```
char *func(void){
    char local_auto;
    return &local_auto ; /* Non-compliant
                         * &local_auto is indeterminate */
}
```

In this example, because `local_auto` is a local variable, after the function returns, the address of `local_auto` is indeterminate.

Copying Pointer Addresses to Local Variables

```
char *sp;

void f(unsigned short u){
    g(&u);
}

void h(void){
    static unsigned short *q;

    unsigned short x =0u;
    q = &x;  /* Non-compliant -
             * &x stored in object with greater lifetime */
}
```

In this example, the function h stores the address of a local variable x in the a static variable q. The lifetime of the static variable q persists after the lifetime of the local variable x ends. Copying x into q is noncompliant with this rule and Polyspace flags the variable x.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required

See Also

Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 18.7

Flexible array members shall not be declared

Description

Rule Definition

Flexible array members shall not be declared.

Rationale

Flexible array members are usually used with dynamic memory allocation. Dynamic memory allocation is banned by Directive 4.12 and Rule 21.3 on page 6-267.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Pointers and Arrays
Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 21.3|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 18.8

Variable-length array types shall not be used

Description

Rule Definition

Variable-length array types shall not be used.

Rationale

When the size of an array declared in a block or function prototype is not an integer constant expression, you specify variable array types. Variable array types are typically implemented as a variable size object stored on the stack. Using variable type arrays can make it impossible to determine statistically the amount of memory for the stack requires.

If the size of a variable-length array is negative or zero, the behavior is undefined.

If a variable-length array must be compatible with another array type, then the size of the array types must be identical and positive integers. If your array does not meet these requirements, the behavior is undefined.

If you use a variable-length array type in a `sizeof`, it is uncertain if the array size is evaluated or not.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Pointers and Arrays
Category: Required
AGC Category: Required

See Also
`MISRA C:2012 Rule 13.6|Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 19.1

An object shall not be assigned or copied to an overlapping object

Description

Rule Definition

An object shall not be assigned or copied to an overlapping object.

Rationale

When you assign an object to another object with overlapping memory, the behavior is undefined. The exceptions are:

- You assign an object to another object with exactly overlapping memory and compatible type.
- You copy one object to another using `memmove`.

Additional Message in Report

- An object shall not be assigned or copied to an overlapping object.
- Destination and source of XX overlap, the behavior is undefined.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Assignment of Union Members

```
void func (void) {
    union {
        short i;
        int j;
    } a = {0}, b = {1};

    a.j = a.i;   /* Non-compliant */
    a = b;       /* Compliant */
}
```

In this example, the rule is violated when `a.i` is assigned to `a.j` because the two variables have overlapping regions of memory.

Assignment of Array Segments

```
#include <string.h>

int arr[10];

void func(void) {
    memcpy (&arr[5], &arr[4], 2u * sizeof(arr[0]));    /* Non-compliant */
    memcpy (&arr[5], &arr[4], sizeof(arr[0]));         /* Compliant */
```

```
    memcpy (&arr[1], &arr[4], 2u * sizeof(arr[0]));    /* Compliant */
}
```

In this example, memory equal to twice `sizeof(arr[0])` is the memory space taken up by two array elements. If that memory space begins from `&a[4]` and `&a[5]`, the two memory regions overlap. The rule is violated when the `memcpy` function is used to copy the contents of these two overlapping memory regions.

Check Information
Group: Overlapping Storage
Category: Mandatory
AGC Category: Mandatory

See Also
`MISRA C:2012 Rule 19.2`|`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 19.2

The union keyword should not be used

Description

Rule Definition

The union keyword should not be used.

Rationale

If you write to a union member and read the same union member, the behavior is well-defined. But if you read a different member, the behavior depends on the relative sizes of the members. For instance:

- If you read a union member with wider memory size, the value you read is unspecified.
- Otherwise, the value is implementation-dependent.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Possible Problems with union Keyword

```
unsigned int zext(unsigned int s)
{
    union                   /* Non-compliant */
    {
        unsigned int ul;
        unsigned short us;
    } tmp;

    tmp.us = s;
    return tmp.ul;          /* Unspecified value */
}
```

In this example, the 16-bit `short` field `tmp.us` is written but the wider 32-bit `int` field `tmp.ul` is read. Using the `union` keyword can cause such unspecified behavior. Therefore, the rule forbids using the `union` keyword.

Check Information

Group: Overlapping Storage
Category: Advisory
AGC Category: Advisory

See Also

MISRA C:2012 Rule 19.1|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 20.1

#include directives should only be preceded by preprocessor directives or comments

Description

Rule Definition

#include directives should only be preceded by preprocessor directives or comments.

Rationale

For better code readability, group all `#include` directives in a file at the top of the file. Undefined behavior can occur if you use `#include` to include a standard header file within a declaration or definition, or if you use part of the Standard Library before including the related standard header files.

Polyspace Implementation

Polyspace flags text that precedes a `#include` directive. Polyspace ignores preprocessor directives, comments, spaces, or "new lines".

Additional Message in Report

#include directives should only be preceded by preprocessor directives or comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Advisory
AGC Category: Advisory

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.2

The ', " or \ characters and the /* or // character sequences shall not occur in a header file name

Description

Rule Definition

The ', " or \ characters and the / or // character sequences shall not occur in a header file name.*

Rationale

The program's behavior is undefined if:

- You use ', ", \, /* or // between < > delimiters in a header name preprocessing token.
- You use ', \, /* or // between " delimiters in a header name preprocessing token.

Although \ results in undefined behavior, many implementations accept / in its place.

Polyspace Implementation

Polyspace flags the characters ', ", \, /* or // between < and > in `#include <filename>`.

Polyspace flags the characters ', \, /* or // between " and " in `#include "filename"`.

Additional Message in Report

The ', "or \ characters and the /* or // character sequences shall not occur in a header file name.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.3

The #include directive shall be followed by either a <filename> or "filename" sequence

Description

Rule Definition

The #include directive shall be followed by either a <filename> or "filename" sequence.

Rationale

This rule applies only after macro replacement.

The behavior is undefined if an `#include` directive does not use one of the following forms:

- `#include <filename>`
- `#include "filename"`

Additional Message in Report

- '#include' expects "FILENAME" or <FILENAME>
- '#include_next' expects "FILENAME" or <FILENAME>
- '#include' does not expect string concatenation.
- '#include_next' does not expect string concatenation.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.4

A macro shall not be defined with the same name as a keyword

Description

Rule Definition

A macro shall not be defined with the same name as a keyword.

Rationale

Using macros to change the meaning of keywords can be confusing. The behavior is undefined if you include a standard header while a macro is defined with the same name as a keyword.

Additional Message in Report

- The macro *macro_name* shall not be redefined.
- The macro *macro_name* shall not be undefined.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Redefining `int` keyword

```
#include <stdlib.h>
#define int some_other_type /* Non-compliant - int keyword behavior altered */

//...
```

In this example, the `#define` violates Rule 20.4 because it alters the behavior of the `int` keyword. The inclusion of the standard header results in undefined behavior.

Correction — Rename keyword

One possible correction is to use a different keyword:

```
#include <stdlib.h>
#define int_mine some_other_type

//...
```

Redefining keywords versus statements

```
#define while(E) for ( ; (E) ; )  /* Non-compliant - while redefined*/
#define unless(E) if ( !(E) )     /* Compliant*/

#define seq(S1, S2) do{ S1; S2;} while(false)  /* Compliant*/
#define compound(S) {S;}                        /* Compliant*/
//...
```

In this example, it is noncompliant to redefine the keyword `while`, but it is compliant to define a macro that expands to statements.

Redefining keywords in different standards

```
#define inline // Non-compliant
```

In this example, redefining `inline` is compliant in C90, but not in C99 because `inline` is not a keyword in C90.

Check Information
Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.5

#undef should not be used

Description

Rule Definition

#undef should not be used.

Rationale

`#undef` can make the software unclear which macros exist at a particular point within a translation unit.

Additional Message in Report

#undef shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Advisory
AGC Category: Readability

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.6

Tokens that look like a preprocessing directive shall not occur within a macro argument

Description

Rule Definition

Tokens that look like a preprocessing directive shall not occur within a macro argument.

Rationale

An argument containing sequences of tokens that otherwise act as preprocessing directives leads to undefined behavior.

Polyspace Implementation

Polyspace looks for the # character in a macro arguments (outside a string or character constant).

Additional Message in Report

Macro argument shall not look like a preprocessing directive.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Macro Expansion Causing Non-Compliance

```
#define M( A ) printf ( #A )

#include <stdio.h>

void foo(void){
    M(
#ifdef SW          /* Non-compliant */
    "Message 1"
#else
    "Message 2"   /* Compliant - SW not defined */
#endif            /* Non-compliant */
    );
}
```

This example shows a macro definition and the macro usage. `#ifdef SW` and `#endif` are noncompliant because they look like a preprocessing directive. Polyspace does not flag `#else` `"Message 2"` because after macro expansion, Polyspace knows `SW` is not defined. The expanded macro is `printf ("\"Message 2\"");`

Check Information

Group: Preprocessing Directives

Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.7

Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses

Description

Rule Definition

Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.

Rationale

If you do not use parentheses, then it is possible that operator precedence does not give the results that you want when macro substitution occurs.

If you are not using a macro parameter as an expression, then the parentheses are not necessary because no operators are involved in the macro.

Additional Message in Report

Expanded macro parameter *param* shall be enclosed in parentheses.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Macro Expressions

```
#define mac1(x, y) (x * y)
#define mac2(x, y) ((x) * (y))

void foo(void){
    int r;

    r = mac1(1 + 2, 3 + 4);      /* Non-compliant */
    r = mac1((1 + 2), (3 + 4));  /* Compliant */

    r = mac2(1 + 2, 3 + 4);      /* Compliant */
}
```

In this example, `mac1` and `mac2` are two defined macro expressions. The definition of `mac1` does not enclose the arguments in parentheses. In line 7, the macro expands to `r = (1 + 2 * 3 + 4);` This expression can be `(1 + (2 * 3) + 4)` or `(1 + 2) * (3 + 4)`. However, without parentheses, the program does not know the intended expression. Line 8 uses parentheses, so the line expands to `(1 + 2) * (3 + 4)`. This macro expression is compliant.

The definition of `mac2` does enclose the argument in parentheses. Line 10 (the same macro arguments in line 7) expands to `(1 + 2) * (3 + 4)`. This macro and macro expression are compliant.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.9 | Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.8

The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1

Description

Rule Definition

The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.

Rationale

Strong typing requires that conditional inclusion preprocessing directives, `#if` or `#elif`, have a controlling expression that evaluates to a Boolean value.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Advisory

See Also

`MISRA C:2012 Rule 14.4|Check MISRA C:2012 (-misra3)`

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.9

All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation

Description

Rule Definition

All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation.

Rationale

If attempt to use a macro identifier in a preprocessing directive, and you have not defined that identifier, then the preprocessor assumes that it has a value of zero. This value might not meet developer expectations.

Additional Message in Report

Identifier is not defined.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Macro Identifiers

```
#if M == 0                  /* Non-compliant - Not defined */
#endif

#if defined (M)             /* Compliant - M is not evaluate */
#if M == 0                  /* Compliant - M is known to be defined */
#endif
#endif

#if defined (M) && (M == 0)  /* Compliant
                              * if M defined, M evaluated in ( M == 0 ) */
#endif
```

This example shows various uses of M in preprocessing directives. The second and third `#if` clauses check to see if the software defines M before evaluating M. The first `#if` clause does not check to see if M is defined, and because M is not defined, the statement is noncompliant.

Check Information
Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Dir 4.9` | `Check MISRA C:2012 (-misra3)`

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.10

The # and ## preprocessor operators should not be used

Description

Rule Definition

The # and ## preprocessor operators should not be used.

Rationale

The order of evaluation associated with multiple #, multiple ##, or a mix of # and ## preprocessor operators is unspecified. In some cases, it is therefore not possible to predict the result of macro expansion.

The use of ## can result in obscured code.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Advisory
AGC Category: Advisory

See Also
MISRA C:2012 Rule 1.3|MISRA C:2012 Rule 20.11|Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.11

A macro parameter immediately following a # operator shall not immediately be followed by a ##
operator

Description

Rule Definition

*A macro parameter immediately following a # operator shall not immediately be followed by a ##
operator.*

Rationale

The order of evaluation associated with multiple #, multiple ##, or a mix of # and ## preprocessor
operators, is unspecified. Rule 20.10 discourages the use of # and ##. The result of a # operator is a
string literal. It is extremely unlikely that pasting this result to any other preprocessing token results
in a valid token.

Additional Message in Report

The ## preprocessor operator shall not follow a macro parameter following a # preprocessor
operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of # and

```
#define A( x )    #x            /* Compliant */
#define B( x, y ) x ## y        /* Compliant */
#define C( x, y ) #x ## y     /* Non-compliant */
```

In this example, you can see three uses of the # and ## operators. You can use these preprocessing
operators alone (line 1 and line 2), but using # then ## is noncompliant (line 3).

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

MISRA C:2012 Rule 20.10|Check MISRA C:2012 (-misra3)

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"

"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.12

A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators

Description

Rule Definition

A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.

Rationale

The parameter to # or ## is not expanded prior to being used. The same parameter appearing elsewhere in the replacement text is expanded. If the macro parameter is itself subject to macro replacement, its use in mixed contexts within a macro replacement might not meet developer expectations.

Additional Message in Report

Expanded macro parameter *param1* is also an operand of *op* operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.13

A line whose first token is # shall be a valid preprocessing directive

Description

Rule Definition

A line whose first token is # shall be a valid preprocessing directive

Rationale

You typically use a preprocessing directive to conditionally exclude source code until a corresponding `#else`, `#elif`, or `#endif` directive is encountered. If your compiler does not detect a preprocessing directive because it is malformed or invalid, you can end up excluding more code than you intended.

If all preprocessing directives are syntactically valid, even in excluded code, this unintended code exclusion cannot happen.

Additional Message in Report

Directive is not syntactically meaningful.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 20.14

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related

Description

Rule Definition

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.

Rationale

When conditional compilation directives include or exclude blocks of code and are spread over multiple files, confusion arises. If you terminate an `#if` directive within the same file, you reduce the visual complexity of the code and the chances of an error.

If you terminate `#if` directives within the same file, you can use `#if` directives in included files

Additional Message in Report

- '#else' not within a conditional.
- '#elseif' not within a conditional.
- '#endif' not within a conditional.

    Unterminated conditional directive.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

MISRA C:2012 Rule 21.1

#define and #undef shall not be used on a reserved identifier or reserved macro name

Description

Rule Definition

#define and #undef shall not be used on a reserved identifier or reserved macro name.

Rationale

Reserved identifiers and reserved macro names are intended for use by the implementation. Removing or changing the meaning of a reserved macro can result in undefined behavior. This rule applies to the following:

- Identifiers or macro names beginning with an underscore
- Identifiers in file scope described in the C Standard Library
- Macro names described in the C Standard Library as being defined in a standard header

The rule checker can flag different identifiers or macros depending on the version of the C standard used in the analysis. See `C standard version (-c-version)`. For instance, if you run a C99 analysis, the reserved identifiers and macros are defined in the ISO/IEC 9899:1999 standard, Section 7, "Library".

Additional Message in Report

- The macro *macro_name* shall not be redefined.
- The macro *macro_name* shall not be undefined.
- The macro *macro_name* shall not be defined.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Defining or Undefining Reserved Identifiers

```
#undef __LINE__             /* Non-compliant - begins with _ */
#define _Guard_H 1          /* Non-compliant - begins with _ */
#undef _BUILTIN_sqrt        /* Non-compliant - implementation may
                             * use _BUILTIN_sqrt for other purposes,
                             * e.g. generating a sqrt instruction */
#define defined             /* Non-compliant - reserved identifier */
#define errno my_errno      /* Non-compliant - library identifier */
#define isneg(x) ( (x) < 0 )  /* Compliant - rule doesn't include
                             * future library directions    */
```

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`MISRA C:2012 Rule 20.4|Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 21.2

A reserved identifier or reserved macro name shall not be declared

Description

Rule Definition

A reserved identifier or reserved macro name shall not be declared.

Rationale

The Standard allows implementations to treat reserved identifiers specially. If you reuse reserved identifiers, you can cause undefined behavior.

Polyspace Implementation

- If you define a macro name that corresponds to a standard library macro, object, or function, rule 21.1 is violated.
- The rule considers tentative definitions as definitions.

Additional Message in Report

Identifier 'XX' shall not be reused.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Standard Libraries
Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 21.3

The memory allocation and deallocation functions of `<stdlib.h>` shall not be used

Description

Rule Definition

The memory allocation and deallocation functions of `<stdlib.h>` shall not be used.

Rationale

Using memory allocation and deallocation routines can cause undefined behavior. For instance:

- You free memory that you had not allocated dynamically.
- You use a pointer that points to a freed memory location.

Polyspace Implementation

The checker flags uses of the `calloc`, `malloc`, `realloc`, `aligned_alloc` and `free` functions.

If you define macros with the same names as these dynamic heap memory allocation functions, and you expand the macros in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro <name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of `malloc`, `calloc`, `realloc` and `free`

```
#include <stdlib.h>

static int foo(void);

typedef struct struct_1 {
    int a;
    char c;
} S_1;

static int foo(void) {

    S_1 * ad_1;
    int  * ad_2;
    int  * ad_3;

    ad_1 = (S_1*)calloc(100U, sizeof(S_1));        /* Non-compliant */
```

```
    ad_2 = malloc(100U * sizeof(int));          /* Non-compliant */
    ad_3 = realloc(ad_3, 60U * sizeof(long));   /* Non-compliant */

    free(ad_1);                                 /* Non-compliant */
    free(ad_2);                                 /* Non-compliant */
    free(ad_3);                                 /* Non-compliant */

    return 1;
}
```

In this example, the rule is violated when the functions `malloc`, `calloc`, `realloc` and `free` are used.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

MISRA C:2012 Rule 18.7 | Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 21.4

The standard header file <setjmp.h> shall not be used

Description

Rule Definition

The standard header file <setjmp.h> shall not be used.

Rationale

Using `setjmp` and `longjmp`, you can bypass normal function call mechanisms and cause undefined behavior.

Polyspace Implementation

If the `longjmp` function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

Check MISRA C:2012 (`-misra3`)

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 21.5

The standard header file <signal.h> shall not be used

Description

Rule Definition

The standard header file <signal.h> shall not be used.

Rationale

Using signal handling functions can cause implementation-defined and undefined behavior.

Polyspace Implementation

If the signal function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

Check MISRA C:2012 (`-misra3`)

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 21.6

The Standard Library input/output functions shall not be used

Description

Rule Definition

The Standard Library input/output functions shall not be used.

Rationale

This rule applies to the functions that are provided by `<stdio.h>` and in C99, their character-wide equivalents provided by `<wchar.h>`. Using these functions can cause unspecified, undefined and implementation-defined behavior.

Polyspace Implementation

If the Standard Library function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Standard Libraries
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 21.7

The Standard Library functions `atof`, `atoi`, `atol`, and `atoll` functions of `<stdlib.h>` shall not be used

Description

Rule Definition

The Standard Library functions `atof, atoi, atol, and atoll` functions of `<stdlib.h>` shall not be used.

Rationale

When a string cannot be converted, the behavior of these functions can be undefined.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Standard Libraries
Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 21.8

The Standard Library functions of `abort`, `exit`, `getnenv` and `system` of `<stdlib.h>` shall not be used

Description

Rule Definition

The Standard Library functions of `abort, exit, getnenv` and `system` of `<stdlib.h>` shall not be used.

Rationale

Using these functions can cause undefined and implementation-defined behaviors.

Polyspace Implementation

Polyspace flags the use of the `abort`, `exit`, `_Exit`, or `quick_exit` functions that are defined in `<stdlib.h>`.

If these functions are user defined, Polyspace does not flag them.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Standard Libraries
Category: Required
AGC Category: Required

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 21.9

The Standard Library library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used

Description

Rule Definition

The library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used.

Rationale

The comparison function in these library functions can behave inconsistently when the elements being compared are equal. Also, the implementation of `qsort` can be recursive and place unknown demands on the call stack.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Standard Libraries
Category: Required
AGC Category: Required

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 21.10

The Standard Library time and date functions shall not be used

Description

Rule Definition

The Standard Library time and date functions shall not be used.

Rationale

Using these functions can cause unspecified, undefined and implementation-defined behavior.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

Check MISRA C:2012 (`-misra3`)

Topics

"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 21.11

The standard header file <tgmath.h> shall not be used

Description

Rule Definition

The standard header file <tgmath.h> shall not be used.

Rationale

Using the facilities of this header file can cause undefined behavior.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Additional Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of Function in `tgmath.h`

```
#include <tgmath.h>

float f1,res;


void func(void) {
    res = sqrt(f1); /* Non-compliant */
}
```

In this example, the rule is violated when the `sqrt` macro defined in `tgmath.h` is used.

Correction — Use Appropriate Function in `math.h`

For this example, one possible correction is to use the function `sqrtf` defined in `math.h` for `float` arguments.

```
#include <math.h>

float f1, res;


void func(void) {
```

```
 res = sqrtf(f1);
}
```

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

See Also

`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 21.12

The exception handling features of `<fenv.h>` should not be used

Description

Rule Definition

The exception handling features of `<fenv.h>` should not be used.

Rationale

In some cases, the values of the floating-point status flags are unspecified. Attempts to access them can cause undefined behavior.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of Features in `<fenv.h>`

```
#include <fenv.h>

void func(float x, float y) {
    float z;

    feclearexcept(FE_DIVBYZERO);              /* Non-compliant */
    z = x/y;

    if(fetestexcept (FE_DIVBYZERO))  {        /* Non-compliant */
    }
    else {
#pragma STDC FENV_ACCESS ON
        z=x*y;
        if(z>x) {
#pragma STDC FENV_ACCESS OFF
            if(fetestexcept (FE_OVERFLOW)) {  /* Non-compliant */
            }
        }
    }
}
```

In this example, the rule is violated when the identifiers `feclearexcept` and `fetestexcept`, and the macros FE_DIVBYZERO and FE_OVERFLOW are used.

Check Information
Group: Standard libraries
Category: Advisory
AGC Category: Advisory

See Also
Check MISRA C:2012 (`-misra3`)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2015b

MISRA C:2012 Rule 21.15

The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types

Description

Rule Definition

The pointer arguments to the Standard Library functions `memcpy, memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types.

This rule comes from MISRA C: 2012 Amendment 1.

Rationale

The functions

```
memcpy( arg1, arg2, num_bytes );
memmove( arg1, arg2, num_bytes );
memcmp( arg1, arg2, num_bytes );
```

perform a byte-by-byte copy, move or comparison between the memory locations that `arg1` and `arg2` point to. A byte-by-byte copy, move or comparison is meaningful only if `arg1` and `arg2` have compatible types.

Using pointers to different data types for `arg1` and `arg2` typically indicates a coding error.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Incompatible Argument Types for `memcpy`

```
#include <stdint.h>

void f ( uint8_t s1[ 8 ], uint16_t s2[ 8 ] )
{
    ( void ) memcpy ( s1, s2, 8 ); /* Non-compliant */
}
```

In this example, `s1` and `s2` are pointers to different data types. The `memcpy` statement copies eight bytes from one buffer to another.

Eight bytes represent the entire span of the buffer that `s1` points to, but only part of the buffer that `s2` points to. Therefore, the `memcpy` statement copies only part of `s2` to `s1`, which might be unintended.

Check Information
Group: Standard libraries

Category: Required
AGC Category: Required

See Also
MISRA C:2012 Rule 21.16 | Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2017a

MISRA C:2012 Rule 21.16

The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type

Description

Rule Definition

The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type.

This rule comes from MISRA C: 2012 Amendment 1.

Rationale

The Standard Library function

```
memcmp ( lhs, rhs, num );
```

performs a byte-by-byte comparison of the first num bytes of the two objects that lhs and rhs point to.

Do not use memcmp for a byte-by-byte comparison of the following.

Type	Rationale
Structures	If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. The content of these extra padding bytes is meaningless. If you perform a byte-by-byte comparison of structures with memcmp, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.
Objects with essentially floating type	The same floating point value can be stored using different representations. If you perform a byte-by-byte comparison of two variables with memcmp, you can reach the false conclusion that the variables are unequal even when they have the same value. The reason is that the values are stored using two different representations.
Essentially char arrays	Essentially char arrays are typically used to store strings. In strings, the content in bytes after the null terminator is meaningless. If you perform a byte-by-byte comparison of two strings with memcmp, you might reach the false conclusion that two strings are not equal, even if the bytes before the null terminator store the same value.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Using memcmp for Comparison of Structures, Unions, and *essentially char* Arrays

```
#include <stdbool.h>
#include <stdint.h>

struct S {
//...
};

bool f1(struct S* s1, struct S* s2)
{
    return (memcmp(s1, s2, sizeof(struct S)) != 0); /* Non-compliant */
}

union U {
    uint32_t range;
    uint32_t height;
};
bool f2(union U* u1, union U* u2)
{
    return (memcmp(u1, u2, sizeof(union U)) != 0); /* Non-compliant */
}

const char a[ 6 ] = "task";
bool f3(const char b[ 6 ])
{
    return (memcmp(a, b, 6) != 0); /* Non-compliant */
}
```

In this example:

- Structures `s1` and `s2` are compared in the `bool_t f1` function. The return value of this function might indicate that `s1` and `s2` are different due to padding. This comparison is noncompliant.

- Unions `u1` and `u2` are compared in the `bool_t f2` function. The return value of this function might indicate that `u1` and `u2` are the same due to unintentional comparison of `u1.range` and `u2.height`, or `u1.height` and `u2.range`. This comparison is noncompliant.

- Essentially char arrays `a` and `b` are compared in the `bool_t f3` function. The return value of this function might incorrectly indicate that the strings are different because the length of `a` (four) is less than the number of bytes compared (six). This comparison is noncompliant.

Check Information

Group: Standard libraries
Category: Required
AGC Category: Required

See Also

MISRA C:2012 Rule 21.15 | Check MISRA C:2012 (-misra3)

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"
"Software Quality Objective Subsets (C:2012)"

Introduced in R2017a

MISRA C:2012 Rule 22.5

A pointer to a `FILE` object shall not be dereferenced

Description

Rule Definition

A pointer to a FILE object shall not be dereferenced.

Rationale

The Standard states that the address of a `FILE` object used to control a stream can be significant. Copying that object might not give the same behavior. This rule ensures that you cannot perform such a copy.

Directly manipulating a `FILE` object might be incompatible with its use as a stream designator.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

FILE* Pointer Dereferenced

```
#include <stdio.h>

void func(void) {
    FILE *pf1;
    FILE *pf2;
    FILE f3;

    pf2 = pf1;        /* Compliant */
    f3 = *pf2;        /* Non-compliant */
    pf2->_flags=0;    /* Non-compliant */
 }
```

In this example, the rule is violated when the FILE* pointer `pf2` is dereferenced.

Check Information
Group: Resources
Category: Mandatory
AGC Category: Mandatory

See Also
`Check MISRA C:2012 (-misra3)`

Topics
"Check for Coding Standard Violations"
"Polyspace MISRA C:2012 Checkers"

"Software Quality Objective Subsets (C:2012)"

Introduced in R2015b

MISRA C++: 2008

MISRA C++:2008 Rule 0-1-1

A project shall not contain unreachable code

Description

Rule Definition

A project shall not contain unreachable code.

Rationale

This rule flags situations where a group of statements is unreachable because of syntactic reasons. For instance, code following a `return` statement are always unreachable.

Unreachable code involve unnecessary maintenance and can often indicate programming errors.

Polyspace Implementation

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Unreachable statements

```
int func(int arg) {
 int temp = 0;
 switch(arg) {
     temp = arg; // Noncompliant
     case 1:
     {
         break;
     }
     default:
     {
         break;
     }
 }
 return arg;
 arg++; // Noncompliant
}
```

These statements are unreachable:

- Statements inside a `switch` statement that do not belong to a `case` or `default` block.
- Statements after a `return` statement.

Check Information
Group: Language Independent Issues
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 0-1-2

A project shall not contain infeasible paths

Description

Rule Definition

A project shall not contain infeasible paths.

Rationale

This rule flags situations where a group of statements is redundant because of nonsyntactic reasons. For instance, an `if` condition is always true or false. Code that is unreachable from syntactic reasons are flagged by rule 0-1-1.

Unreachable or redundant code involve unnecessary maintenance and can often indicate programming errors.

Polyspace Implementation

Bug Finder and Code Prover check this rule differently. The analysis can produce different results.

- Bug Finder checks for this rule through the `Dead code` and `Useless if` checkers..
- Code Prover does not use run-time checks to detect violations of this rule. Instead, Code Prover detects the violations at compile time.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Boolean Operations with Invariant Results

```
void func (unsigned int arg) {
 if (arg >= 0U) //Noncompliant
      arg  = 1U;
 if (arg < 0U) //Noncompliant
      arg = 1U;
}
```

An `unsigned int` variable is nonnegative. Both `if` conditions involving the variable are always true or always false and are therefore redundant.

Check Information
Group: Language Independent Issues
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 0-1-3

A project shall not contain unused variables

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

A project shall not contain unused variables.

Polyspace Implementation

The checker flags local or global variables that are declared or defined but not used anywhere in the source files. This specification also applies to members of structures and classes.

Additional Message in Report

A project shall not contain unused variables.

Variable is never used or used only in unreachable code.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of Named Bit Field for Padding

```
#include <iostream>
struct S {
    unsigned char b1 : 3;
    unsigned char pad: 1;  //Noncompliant
    unsigned char b2 : 4;
};
void init(struct S S_obj)
{
    S_obj.b1 = 0;
    S_obj.b2 = 0;
}
```

In this example, the bit field `pad` is used for padding the structure. Therefore, the field is never read or written and causes a violation of this rule. To avoid the violation, use an unnamed field for padding.

```
#include <iostream>
struct S {
    unsigned char b1 : 3;
    unsigned char : 1;  //Compliant
    unsigned char b2 : 4;
};
```

```
void init(struct S S_obj)
{
    S_obj.b1 = 0;
    S_obj.b2 = 0;
}
```

Check Information

Group: Language Independent Issues
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2018a

MISRA C++:2008 Rule 0-1-4

A project shall not contain non-volatile POD variables having only one use

Description

Rule Definition

A project shall not contain non-volatile POD variables having only one use.

Rationale

If you use a non-volatile variable with a Plain Old Data type (`int`, `double`, etc.) *only once*, you can replace the variable with a constant literal. Your use of a variable indicates that you intended more than one use for that variable and might have a programming error in the code. You might have omitted the other uses of the non-volatile variable or incorrectly used other variables at intended points of use.

Polyspace Implementation

The checker flags local and static variables that have a function scope (locally static) and file scope, which are used only once. The checker considers `const`-qualified global variables without the `extern` specifier as static variables with file scope.

The checker counts these use cases as one use of the non-volatile variable:

- An explicit initialization using a constant literal or the return value of a function
- An assignment
- A reference to the variable such as a read operation
- An assignment of the variable address to a pointer

If the variable address is assigned to a pointer, the checker assumes that the pointer might be dereferenced later and does not flag the variable.

Some objects are designed to be used only once by their semantics. Polyspace does not flag a single use of these objects:

- `lock_guard`
- `scoped_lock`
- `shared_lock`
- `unique_lock`
- `thread`
- `future`
- `shared_future`

If you use nonstandard objects that provide similar functionality as the objects in the preceding list, Polyspace might flag single uses of the nonstandard objects. Justify their single uses by using comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Non-volatile Variable Used Only Once

```
#include <mutex>
int readStatus1();
int readStatus2();
extern std::mutex m;
void foo()
{
    // Initiating lock 'lk'
    std::lock_guard<std::mutex> lk{m};
    int checkEngineStatus1 = readStatus1();
    int checkEngineStatus2 = readStatus2();//Noncompliant

    if(checkEngineStatus1) {
        //Perform some actions if both statuses are valid
    }
    // Release lock when 'lk' is deleted at exit point of scope
}
```

In this example, the variable `checkEngineStatus2` is used only once. The single use of this variable might indicate a programming error. For instance, you might have intended to check both `checkEngineStatus1` and `checkEngineStatus2` in the `if` condition, but omitted the second check. The `lock_guard` object `lk` is also used only once. Because the semantics of a `lock_guard` object justifies its single use, Polyspace does not flag it.

Check Information
Group: Language Independent Issues
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2020b

MISRA C++:2008 Rule 0-1-5

A project shall not contain unused type declarations

Description

Rule Definition

A project shall not contain unused type declarations.

Rationale

If a type is declared but not used, when reviewing the code later, it is unclear if the type is redundant or left unused by mistake.

Unused types can indicate coding errors. For instance, you declared a enumerated data type for some specialized data but used an integer type for the data.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Unused enum Declaration

```
enum switchValue {low, medium, high}; //Noncompliant

void operate(int userInput) {
    switch(userInput) {
        case 0: // Turn on low setting
                break;
        case 1: // Turn on medium setting
                break;
        case 2: // Turn on high setting
                break;
        default: // Return error
    }
}
```

In this example, the enumerated type `switchValue` is not used. Perhaps the intention was to use the type as `switch` input like this.

```
enum switchValue {low, medium, high}; //Compliant

void operate(switchValue userInput) {
    switch(userInput) {
        case low: // Turn on low setting
                    break;
        case medium: // Turn on medium setting
                        break;
        case high: // Turn on high setting
                    break;
        default: // Return error
```

```
        }
}
```

Check Information
Group: Language Independent Issues
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2018a

MISRA C++:2008 Rule 0-1-7

The value returned by a function having a non- void return type that is not an overloaded operator shall always be used

Description

Rule Definition

The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.

Rationale

The unused return value might indicate a coding error or oversight.

Overloaded operators are excluded from this rule because their usage must emulate built-in operators which might not use their return value.

Polyspace Implementation

The checker flags functions with non-`void` return if the return value is not used or not explicitly cast to a `void` type.

The checker does not flag the functions `memcpy`, `memset`, `memmove`, `strcpy`, `strncpy`, `strcat`, `strncat` because these functions simply return a pointer to their first arguments.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Return Value Not Used

```
#include <iostream>
#include <new>

int assignMemory(int * ptr){
    int res = 1;
    ptr = new (std::nothrow) int;
    if(ptr==NULL) {
        res = 0;
    }
    return res;
}

void main() {
    int val;
    int status;

    assignMemory(&val);    //Noncompliant
    status = assignMemory(&val); //Compliant
```

```
    (void)assignMemory(&val); //Compliant

}
```

The first call to the function `assignMemory` is noncompliant because the return value is not used. The second and third calls use the return value. The return value from the second call is assigned to a local variable.

The return value from the third call is cast to `void`. Casting to `void` indicates deliberate non-use of the return value and cannot be a coding oversight.

Check Information
Group: Language Independent Issues
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 0-1-9

There shall be no dead code

Description

Rule Definition

There shall be no dead code.

Rationale

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code. For instance, suppose that a variable is never read following a write operation. The write operation is redundant.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Redundant Operations

```
#define ULIM 10000

int func(int arg) {
    int res;
    res = arg*arg + arg;
    if (res > ULIM)
        res = 0; //Noncompliant
    return arg;
}
```

In this example, the operations involving `res` are redundant because the function `func` returns its argument `arg`. All operations involving `res` can be removed without changing the effect of the function.

The checker flags the last write operation on `res` because the variable is never read after that point. The dead code can indicate an unintended coding error. For instance, you intended to return the value of `res` instead of `arg`.

Check Information
Group: Language Independent Issues
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2016b

MISRA C++:2008 Rule 0-1-10

Every defined function shall be called at least once

Description

Rule Definition

Every defined function shall be called at least once.

Rationale

If a function with a definition is not called, it might indicate a serious coding error. For instance, the function call is unreachable or a different function is called unintentionally.

Polyspace Implementation

The checker detects situations where a static function is defined but not called at all in its translation unit.

Additional Message in Report

Every defined function shall be called at least once. The static function *funcName* is not called.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Uncalled Static Function

```
static void func1() {
}

static void func2() { //Noncompliant
}

void func3();

int main() {
    func1();
    return 0;
}
```

The `static` function `func2` is defined but not called.

The function `func3` is not called either, however, it is only declared and not defined. The absence of a call to `func3` does not violate the rule.

Check Information

Group: Language Independent Issues

Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 0-1-11

There shall be no unused parameters (named or unnamed) in nonvirtual functions

Description

Rule Definition

There shall be no unused parameters (named or unnamed) in nonvirtual functions.

Rationale

Unused parameters often indicate later design changes. You perhaps removed all uses of a specific parameter but forgot to remove the parameter from the parameter list.

Unused parameters constitute an unnecessary overhead. You can also inadvertently call the function with a different number of arguments causing a parameter mismatch.

Polyspace Implementation

The checker flags a function that has unused named parameters unless the function body is empty.

Additional Message in Report

Function *funcName* has unused parameters.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Unused Parameters

```
typedef int (*callbackFn) (int a, int b);

int callback_1 (int a, int b) { //Compliant
    return a+b;
}

int callback_2 (int a, int b) { //Noncompliant
    return a;
}

int callback_3 (int, int b) { //Compliant - flagged by Polyspace
    return b;
}

int getCallbackNumber();
int getInput();

void main() {
    callbackFn ptrFn;
    int n = getCallbackNumber();
```

```
    int x = getInput(), y = getInput();
    switch(n) {
        case 0: ptrFn = &callback_1; break;
        case 1: ptrFn = &callback_2; break;
        default: ptrFn = &callback_3; break;
    }

    (*ptrFn)(x,y);
}
```

In this example, the three functions `callback_1`, `callback_2` and `callback_3` are used as callback functions. One of the three functions is called via a function pointer depending on a value obtained at run time.

- Function `callback_1` uses all its parameters and does not violate the rule.
- Function `callback_2` does not use its parameter `a` and violates this rule.
- Function `callback_3` also does not use its first parameter but it does not violate the rule because the parameter is unnamed. However, Polyspace flags the unused parameter as a rule violation. If you see a violation of this kind, justify the violation with comments. See "Address Polyspace Results Through Bug Fixes or Justifications".

Check Information

Group: Language Independent Issues
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2016b

MISRA C++:2008 Rule 0-1-12

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it

Description

Rule Definition

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.

Rationale

Unused parameters often indicate later design changes. You perhaps removed all uses of a specific parameter but forgot to remove the parameter from the parameter list.

Unused parameters constitute an unnecessary overhead. You can also inadvertently call the function with a different number of arguments causing a parameter mismatch.

Polyspace Implementation

For each virtual function, the checker looks at all overrides of the function. If an override has a named parameter that is not used, the checker shows a violation on the original virtual function and lists the override as a supporting event.

Note that Polyspace checks for unused parameters in virtual functions within single translation units. For instance, if a base class contains a virtual method with an unused parameter but the derived class implementation of the method uses that parameter, the rule is not violated. However, if the base class and derived class are defined in different files, the checker, which operates file by file, flags a violation of this rule on the base class.

The checker does not flag unused parameters in functions with empty bodies.

Additional Message in Report

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.

Function *funcName* has unused parameters.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Unused Parameter in Virtual Function

```
class base {
    public:
        virtual void assignVal (int arg1, int arg2) = 0; //Noncompliant
```

```
        virtual void assignAnotherVal (int arg1, int arg2) = 0;
};

class derived1: public base {
    public:
        virtual void assignVal (int arg1, int arg2) {
            arg1 = 0;
        }
        virtual void assignAnotherVal (int arg1, int arg2) {
            arg1 = 1;
        }
};

class derived2: public base {
    public:
        virtual void assignVal (int arg1, int arg2) {
            arg1 = 0;
        }
        virtual void assignAnotherVal (int arg1, int arg2) {
            arg2 = 1;
        }
};
```

In this example, the second parameter of the virtual method `assignVal` is not used in any of the derived class implementations of the method.

On the other hand, the implementation of the virtual method `assignAnotherVal` in derived class `derived1` uses the first parameter of the method. The implementation in `derived2` uses the second parameter. Both parameters of `assignAnotherVal` are used and therefore the virtual method does not violate the rule.

Check Information
Group: Language Independent Issues
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2016b

MISRA C++:2008 Rule 0-2-1

An object shall not be assigned to an overlapping object

Description

Rule Definition

An object shall not be assigned to an overlapping object.

Rationale

When you assign an object to another object with overlapping memory, the behavior is undefined.

The exceptions are:

- You assign an object to another object with exactly overlapping memory and compatible type.
- You copy one object to another with `memmove`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Assignment of Union Members

```
void func (void) {
    union {
        short i;
        int j;
    } a = {0}, b = {1};

    a.j = a.i;   //Noncompliant
    a = b;       //Compliant
}
```

In this example, the rule is violated when `a.i` is assigned to `a.j` because the two variables have overlapping regions of memory.

Check Information
Group: Language Independent Issues
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2016b

MISRA C++:2008 Rule 0-3-2

If a function generates error information, then that error information shall be tested

Description

Rule Definition

If a function generates error information, then that error information shall be tested.

Rationale

If you do not check the return value of functions that indicate error information through their return values, your program can behave unexpectedly. Errors from these functions can propagate throughout the program causing incorrect output, security vulnerabilities, and possibly system failures.

For the `errno`-setting functions, to see if the function call completed without errors, check `errno` for error values. The return values of these `errno`-setting functions do not indicate errors. The return value can be one of the following:

- `void`
- Even if an error occurs, the return value can be the same as the value from a successful call. Such return values are called in-band error indicators. For instance, `strtol` converts a string to a long integer and returns the integer. If the result of conversion overflows, the function returns `LONG_MAX` and sets `errno` to `ERANGE`. However, the function can also return `LONG_MAX` from a successful conversion. Only by checking `errno` can you distinguish between an error and a successful conversion.

For the `errno`-setting functions, you can determine if an error occurred only by checking `errno`.

Polyspace Implementation

The checker raises a violation when:

- You call sensitive standard functions that return information about possible errors and you do one of the following:
  - Ignore the return value.

    You simply do not assign the return value to a variable, or explicitly cast the return value to `void`.
  - Use an output from the function (return value or argument passed by reference) without testing the return value for errors.

The checker considers a function as sensitive if the function call is prone to failure because of reasons such as:

- Exhausted system resources (for example, when allocating resources).
- Changed privileges or permissions.
- Tainted sources when reading, writing, or converting data from external sources.

- Unsupported features despite an existing API.

Some of these functions can perform critical tasks such as:

- Set privileges (for example, `setuid`)
- Create a jail (for example, `chroot`)
- Create a process (for example, `fork`)
- Create a thread (for example, `pthread_create`)
- Lock or unlock mutex (for example, `pthread_mutex_lock`)
- Lock or unlock memory segments (for example, `mlock`)

For functions that are not critical, the checker allows casting the function return value to `void`.

- You call a function that sets `errno` to indicate error conditions, but do not check `errno` after the call. For these functions, checking `errno` is the only reliable way to determine if an error occurred.

Functions that set `errno` on errors include:

- `fgetwc`, `strtol`, and `wcstol`.

For a comprehensive list of functions, see documentation about errno.

- POSIX `errno`-setting functions such as `encrypt` and `setkey`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Sensitive Function Return Ignored

```
#include <pthread.h>
#include <cstdlib>
#define fatal_error() abort()

void initialize_1() {
    pthread_attr_t attr;
    pthread_attr_init(&attr); //Noncompliant
}

void initialize_2() {
    pthread_attr_t attr;
    (void)pthread_attr_init(&attr); //Compliant
}

void initialize_3() {
    pthread_attr_t attr;
    int result;
    result = pthread_attr_init(&attr); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

This example shows a call to the sensitive function `pthread_attr_init`. The return value of `pthread_attr_init` is ignored, causing a rule violation.

To be compliant, you can explicitly cast the return value to `void` or test the return value of `pthread_attr_init` and check for errors.

Critical Function Return Ignored

```
#include <pthread.h>
#include <cstdlib>
#define fatal_error() abort()
extern void *start_routine(void *);

void returnnotchecked_1() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;

    (void)pthread_attr_init(&attr);
    (void)pthread_create(&thread_id, &attr, &start_routine, ((void *)0)); //Noncompliant
    pthread_join(thread_id,  &res); //Noncompliant
}

void returnnotchecked_2() {
    pthread_t thread_id;
    pthread_attr_t attr;
    void *res;
    int result;

    (void)pthread_attr_init(&attr);
    result = pthread_create(&thread_id, &attr, &start_routine, NULL); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }

    result = pthread_join(thread_id,  &res); //Compliant
    if (result != 0) {
        /* Handle error */
        fatal_error();
    }
}
```

In this example, two critical functions are called: `pthread_create` and `pthread_join`. The return value of the `pthread_create` is ignored by casting to void, but because `pthread_create` is a critical function (not just a sensitive function), the rule checker still raises a violation. The other critical function, `pthread_join`, returns a value that is ignored implicitly.

To be compliant, check the return value of these critical functions to verify the function performed as expected.

errno Not Checked After Call to `strtol`

```
#include<cstdlib>
#include<cerrno>
#include<climits>
#include<iostream>
```

```
int main(int argc, char *argv[]) {
    char *str, *endptr;
    int base;

    str = argv[1];
    base = 10;

    long val = strtol(str, &endptr, base); //Noncompliant
    std::cout<<"Return value of strtol() = %ld\n" << val;

    errno = 0;
    long val2 = strtol(str, &endptr, base); //Compliant
    if((val2 == LONG_MIN || val2 == LONG_MAX) && errno == ERANGE) {
        std::cout<<"strtol error";
        exit(EXIT_FAILURE);
    }
    std::cout<<"Return value of strtol() = %ld\n" << val2;
}
```

In the noncompliant example, the return value of `strtol` is used without checking `errno`.

To be compliant, before calling `strtol`, set `errno` to zero . After a call to `strtol`, check the return value for LONG_MIN or LONG_MAX and `errno` for ERANGE.

Check Information
Group: Language Independent Issues
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2020b

MISRA C++:2008 Rule 1-0-1

All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1"

Description

Rule Definition

All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".

Polyspace Implementation

The checker reports compilation errors as detected by a compiler that strictly adheres to the C++03 Standard (ISO/IEC 14882:2003).

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

The message has two parts:

- Rule statement:

  All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".

- Compilation error message such as:

  Expected a ;

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: General
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-3-1

Trigraphs shall not be used

Description

Rule Definition

Trigraphs shall not be used.

Rationale

You denote trigraphs with two question marks followed by a specific third character (for instance,`'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

For instance, the string

`"(Date should be in the form ??-??-??)"`

is transformed to

`"(Date should be in the form ~~]"`

but this transformation might not be intended.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-5-1

Digraphs should not be used

Description

Rule Definition

Digraphs should not be used.

Rationale

Digraphs are a sequence of two characters that are supposed to be treated as a single character. The checker flags use of these digraphs:

- <%, indicating [
- %>, indicating]
- <:, indicating {
- :>, indicating }
- %:, indicating #
- %:%:

When developing or reviewing code with digraphs, the developer or reviewer can incorrectly consider the digraph as a sequence of separate characters.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Lexical Conventions
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-7-1

The character sequence /* shall not be used within a C-style comment

Description

Rule Definition

The character sequence / shall not be used within a C-style comment.*

Rationale

If your code contains a /* in a /* */ comment, it typically means that you have inadvertently commented out code. See the example that follows.

Polyspace Implementation

You cannot justify a violation of this rule using source code annotations.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of /* in /* */ Comment

```
void setup(void);
void foo() {
    /* Initializer functions
     setup();
    /* Step functions */  //Noncompliant
}
```

In this example, the call to `setup()` is commented out because the ending */ is omitted, perhaps inadvertently. The checker flags this issue by highlighting the /* in the /* */ comment.

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-7-2

Sections of code shall not be "commented out" using C-style comments

Description

Rule Definition

Sections of code shall not be "commented out" using C-style comments.

Rationale

C-style comments enclosed in /* */ do not support nesting. A comment beginning with /* ends at the first */ even when the */ is intended as the end of a later nested comment. If a section of code that is commented out already contains comments, you can encounter compilation errors (or at least comment out less code than you intend).

Commenting out code is not a good practice. The commented out code can remain out of sync with the surrounding code without causing compilation errors. Later, if you uncomment the code, you can encounter unexpected issues.

Use comments only to explain aspects of the code that are not apparent from the code itself.

Polyspace Implementation

The checker uses internal heuristics to detect commented out code. For instance, characters such as #, ;, { or } indicate comments that might potentially contain code. These comments are then evaluated against other metrics to determine the likelihood of code masquerading as comment. For instance, several successive words without a symbol in between reduces this likelihood.

The checker does not flag the following comments even if they contain code:

- Doxygen comments beginning with /** or /*!.

- Comments that repeat the same symbol several times, for instance, the symbol = here:

  ```
  /* ====================================
   * A comment
   * ====================================*/
  ```

- Comments on the first line of a file.

- Comments that mix the C style (/* */) and C++ style (//).

The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Code Commented Out With C-Style Comments

```
#include <iostream>
/* class randInt {//Noncompliant
    public:
       int getRandInt();
};
*/

int getRandInt();

/* Function to print random integers*/
void printInteger() {
    /* int val = getRandInt();//Noncompliant
     * val++;
     * std::cout << val;*/
    std::cout << getRandInt();
}
```

This example contains two blocks of commented out code, that constitutes two rule violations.

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2020b

MISRA C++:2008 Rule 2-7-3

Sections of code should not be "commented out" using C++-style comments

Description

Rule Definition

Sections of code should not be "commented out" using C++-style comments.

Rationale

Commenting out code is not a good practice. The commented out code can remain out of sync with the surrounding code without causing compilation errors. Later, if you uncomment the code, you can encounter unexpected issues.

Use comments only to explain aspects of the code that are not apparent from the code itself.

Polyspace Implementation

The checker uses internal heuristics to detect commented out code. For instance, characters such as #, ;, { or } indicate comments that might potentially contain code. These comments are then evaluated against other metrics to determine the likelihood of code masquerading as comment. For instance, several successive words without a symbol in between reduces this likelihood.

The checker does not flag the following comments even if they contain code:

- Doxygen comments beginning with /// or //!.
- Comments that repeat the same symbol several times, for instance, the symbol = here:

  ```
  // ====================================
  // A comment
  // ===================================*/
  ```
- Comments on the first line of a file.
- Comments that mix the C style (/* */) and C++ style (//).

The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Code Commented Out With C++-Style Comments

```
#include <iostream>
int getRandInt();

// Function to print random integers
void printInteger() {
```

```
        // int val = getRandInt();
        // val++;
        // std::cout << val;
        std::cout << getRandInt();
}
```

This example contains a block of commented out code that violates the rule.

Check Information
Group: Lexical Conventions
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2020b

MISRA C++:2008 Rule 2-10-1

Different identifiers shall be typographically unambiguous

Description

Rule Definition

Different identifiers shall be typographically unambiguous.

Rationale

When you use identifiers that are typographically close, you can confuse between them.

The identifiers should not differ by:

- The interchange of a lowercase letter with its uppercase equivalent.
- The presence or absence of the underscore character.
- The interchange of the letter O and the digit 0.
- The interchange of the letter I and the digit 1.
- The interchange of the letter I and the letter l.
- The interchange of the letter S and the digit 5.
- The interchange of the letter Z and the digit 2.
- The interchange of the letter n and the letter h.
- The interchange of the letter B and the digit 8.
- The interchange of the letters rn and the letter m.

Polyspace Implementation

The rule checker does not consider the fully qualified names of variables when checking this rule.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  //Non-compliant

    int id2_numval;
    int id2_numVal;    //Non-compliant
```

```
        int id3_lvalue;
        int id3_Ivalue;    //Non-compliant

        int id4_xyZ;
        int id4_xy2;        //Non-compliant

        int id5_zer0;
        int id5_zer0;       //Non-compliant

        int id6_rn;
        int id6_m;          //Non-compliant
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-10-2

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope

Description

Rule Definition

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Rationale

The rule flags situations where the same identifier name is used in two variable declarations, one in an outer scope and the other in an inner scope.

```
int var;
...
{
...
  int var;
...
}
```

All uses of the name in the inner scope refers to the variable declared in the inner scope. However, a developer or code reviewer can incorrectly assume that the usage refers to the variable declared in the outer scope.

Polyspace Implementation

The rule checker flags all cases of variable shadowing including when:

- The same identifier name is used in an outer and inner named namespace.
- The same name is used for a class data member and a variable outside the class.
- The same name is used for a method in a base and derived class.

To exclude these cases, switch to the more modern standard AUTOSAR C++14 and check for the rule `AUTOSAR C++14 Rule A2-10-1`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Local Variable Hiding Global Variable

```
int varInit = 1;

void doSomething(void);

void step(void) {
    int varInit = 0; //Noncompliant
    if(varInit)
```

```
        doSomething();
}
```

In this example, `varInit` defined in `func` hides the global variable `varInit`. The `if` condition refers to the local `varInit` and the block is unreachable, but you might expect otherwise.

Loop Index Hiding Variable Outside Loop

```
void runSomeCheck(int);

void checkMatrix(int dim1, int dim2) {
  for(int index = 0; index < dim1; index++) {
      for(int index = 0; index < dim2; index++) { // Noncompliant
          runSomeCheck(index);
      }
  }
}
```

In this example, the variable `index` defined in the inner `for` loop hides the variable with the same name in the outer loop.

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-10-3

A typedef name (including qualification, if any) shall be a unique identifier

Description

Rule Definition

A typedef name (including qualification, if any) shall be a unique identifier.

Rationale

The rule flags identifier declarations where the identifier name is the same as a previously declared typedef name. When you use identifiers that are identical, you can confuse between them.

Polyspace Implementation

The checker does not flag situations where the conflicting names occur in different namespaces.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

A typedef name (including qualification, if any) shall be a unique identifier.

Identifier *typeName* should not be reused.

Already used as typedef name (*fileName lineNumber*).

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Typedef Name Conflicting with Other Identifiers

```
namespace NS1 {
    typedef int WIDTH;
}

namespace NS2 {
    float WIDTH; //Compliant
}

void f1() {
    typedef int TYPE;
}

void f2() {
    float TYPE; //Noncompliant
}
```

In this example, the declaration of `TYPE` in `f2()` conflicts with a typedef declaration in `f1()`.

The checker does not flag the redeclaration of `WIDTH` because the two declarations belong to different namespaces.

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-10-4

A class, union or enum name (including qualification, if any) shall be a unique identifier

Description

Rule Definition

A class, union or enum name (including qualification, if any) shall be a unique identifier.

Rationale

The rule flags identifier declarations where the identifier name is the same as a previously declared class, union or typedef name. When you use identifiers that are identical, you can confuse between them.

Polyspace Implementation

The checker does not flag situations where the conflicting names occur in different namespaces.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

A class, union or enum name (including qualification, if any) shall be a unique identifier.

Identifier *tagName* should not be reused.

Already used as tag name (*fileName lineNumber*).

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Typedef Name Conflicting with Other Identifiers

```
void f1() {
    class floatVar {};
}

void f2() {
    float floatVar; //Noncompliant
}
```

In this example, the declaration of `floatVar` in `f2()` conflicts with a class declaration in `f1()`.

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-10-5

The identifier name of a non-member object or function with static storage duration should not be reused

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

The identifier name of a non-member object or function with static storage duration should not be reused.

Rationale

The rule flags situations where the name of an identifier with static storage duration is reused. The rule applies even if the identifiers belong to different namespaces because the reuse leaves the chance that you mistake one identifier for the other.

Polyspace Implementation

The rule checker flags redefined functions only when there is a declaration.

The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions
- Uncalled and undefined local functions
- Unused types and variables

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Additional Message in Report

The identifier name of a non-member object or function with static storage duration should not be reused.

Identifier *name* should not be reused.

Already used as static identifier with static storage duration (*fileName lineNumber*).

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Reuse of Identifiers in Different Namespaces

```
namespace NS1 {
    static int WIDTH;
}

namespace NS2 {
    float WIDTH; //Noncompliant
}
```

In this example, the identifier name WIDTH is reused in the two namespaces NS1 and NS2.

Check Information
Group: Lexical Conventions
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-10-6

If an identifier refers to a type, it shall not also refer to an object or a function in the same scope

Description

Rule Definition

If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

Rationale

For compatibility with C, in C++, you are allowed to use the same name for a type and an object or function. However, the name reuse can cause confusion during development or code review.

Polyspace Implementation

If the identifier is a function and the function is both declared and defined, then the violation is reported only once.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Reuse of Name for Type and Object

```
struct vector{
    int x;
    int y;
    int z;
}vector; //Noncompliant
```

In this example, the name `vector` is used both for the structured data type and for an object of that type.

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-13-1

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used

Description

Rule Definition

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

Rationale

Escape sequences are certain special characters represented in string and character literals. They are written with a backslash (\) followed by a character.

The C++ Standard (ISO/IEC 14882:2003, Sec. 2.13.2) defines a list of escape sequences. See Escape Sequences. Use of escape sequences (backslash followed by character) outside that list leads to undefined behavior.

Additional Message in Report

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

char is not an ISO/IEC escape sequence.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Incorrect Escape Sequences

```
void func () {
  const char a[2] = "\k"; //Noncompliant
  const char b[2] = "\b"; //Compliant
}
```

In this example, \k is not a recognized escape sequence.

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-13-2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used

Description

Rule Definition

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Rationale

Octal constants are denoted by a leading zero. A developer or code reviewer can mistake an octal constant as a decimal constant with a redundant leading zero.

Octal escape sequences beginning with \ can also cause confusion. Inadvertently introducing an 8 or 9 in the digit sequence after \ breaks the escape sequence and introduces a new digit. A developer or code reviewer can ignore this issue and continue to treat the escape sequence as one digit.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of Octal Constants and Octal Escape Sequences

```
void func(void) {
  int busData[6];

  busData[0] = 100;
  busData[1] = 108;
  busData[2] = 052;      //Noncompliant
  busData[3] = 071;      //Noncompliant
  busData[4] = '\109';   //Noncompliant
  busData[5] = '\100';   //Noncompliant

}
```

The checker flags all octal constants (other than zero) and all octal escape sequences (other than \0).

In this example:

*   The octal escape sequence contains the digit 9, which is not an octal digit. This escape sequence has implementation-defined behavior.
*   The octal escape sequence \100 represents the number 64, but the rule checker forbids this use.

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-13-3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type

Description

Rule Definition

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Rationale

The signedness of a constant is determined from:

- Value of the constant.
- Base of the constant: octal, decimal or hexadecimal.
- Size of the various types.
- Any suffixes used.

Unless you use a suffix u or U, another developer looking at your code cannot determine easily whether a constant is signed or unsigned.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-13-4

Literal suffixes shall be upper case

Description

Rule Definition

Literal suffixes shall be upper case.

Rationale

Literal constants can end with the letter l (el). Enforcing literal suffixes to be upper case removes potential confusion between the letter l and the digit 1.

For consistency, use upper case constants for other suffixes such as U (unsigned) and F (float).

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of Literal Constants with Lower Case Suffix

```
const int a = 0l; //Noncompliant
const int b = 0L; //Compliant
```

In this example, both a and b are assigned the same literal constant. However, from a quick glance, one can mistakenly assume that a is assigned the value 01 (octal one).

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 2-13-5

Narrow and wide string literals shall not be concatenated

Description

Rule Definition

Narrow and wide string literals shall not be concatenated.

Rationale

Narrow string literals are enclosed in double quotes without a prefix. Wide string literals are enclosed in double quotes with a prefix L outside the quotes. See string literals.

Concatenation of narrow and wide string literals can lead to undefined behavior.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Concatenation of Narrow and Wide String Literals

```
char array[] = "Hello" "World";
wchar_t w_array[] = L"Hello" L"World";
wchar_t mixed[] = "Hello" L"World"; //Noncompliant
```

In this example, in the initialization of the array `mixed`, the narrow string literal `"Hello"` is concatenated with the wide string literal `L"World"`.

Check Information
Group: Lexical Conventions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-1-1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule

Description

Rule Definition

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Rationale

If a header file with variable or function definitions appears in multiple inclusion paths, the header file violates the One Definition Rule possibly leading to unpredictable behavior. For instance, a source file includes the header file `include.h` and another header file, which also includes `include.h`.

Polyspace Implementation

The rule checker flags variable and function definitions in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-1-2

Functions shall not be declared at block scope

Description

Rule Definition

Functions shall not be declared at block scope.

Rationale

It is a good practice to place all declarations at the namespace level.

Additionally, if you declare a function at block scope, it is often not clear if the statement is a function declaration or an object declaration with a call to the constructor.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Function Declarations at Block Scope

```
class A {
};

void b1() {
    void func(); //Noncompliant
    A a();   //Noncompliant
}
```

In this example, the declarations of `func` and `a` are in the block scope of `b1`.

The second function declaration can cause confusion because it is not clear if `a` is a function that returns an object of type `A` or `a` is itself an object of type `A`.

Check Information
Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-1-3

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization

Description

Rule Definition

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Rationale

Though you can declare an incomplete array type and later complete the type, specifying the array size during the first declaration makes the subsequent array access less error-prone.

Additional Message in Report

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Size of array *arrayName* should be explicitly stated.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Array Size Unspecified During Declaration

```
int array[10];
extern int array2[]; //Noncompliant
int array3[]= {0,1,2};
extern int array4[10];
```

In the declaration of `array2`, the array size is unspecified.

Check Information
Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-2-1

All declarations of an object or function shall have compatible types

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

All declarations of an object or function shall have compatible types.

Rationale

If the declarations of an object or function in two different translation units have incompatible types, the behavior is undefined.

Polyspace Implementation

Polyspace considers two types to be compatible if they have the same size and signedness in the environment that you use. The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions
- Uncalled and undefined local functions
- Unused types and variables

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Compatible and Incompatible Definitions in Two Files

file1.cpp

```
typedef          char       char_t;
typedef signed   short      int16_t;
typedef signed   long       int64_t;

namespace bar {
    int64_t a;
    int16_t c;

};
```

file2.cpp

```
typedef          char          char_t;
typedef signed   int           int32_t;

namespace bar {
    extern char_t c;// Noncompliant
    extern int32_t a;
    void foo(void){
        ++a;
        ++c;
    }
};
```

In this example, the variable `bar::c` is defined as a `char` in `file2.cpp` and as a `signed short` in `file1.cpp`. In the target processor i386, the size of these types are not equal. Polyspace flags the definition of `bar::c`.

The variable `bar::a` is defined as a `long` in `file1.cpp` and as an `int` in `file2.cpp`. In the target processor i386, both `int` and `long` has a size of 32 bits. Because the definitions of `bar::a` is compatible in both files, Polyspace does not raise a flag.

Check Information
Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-2-2

The One Definition Rule shall not be violated

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

The One Definition Rule shall not be violated.

Rationale

Violations of the One Definition Rule leads to undefined behavior.

Polyspace Implementation

The checker flags situations where the same function or object has multiple definitions and the definitions differ by some token. The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static` or `extern` functions
- Uncalled and undefined local functions
- Unused types and variables

Additional Message in Report

The One Definition Rule shall not be violated.

Declaration of class *className* violates the One Definition Rule:

it conflicts with other declaration (*fileName lineNumber*).

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Different Tokens in Same Type Definition

This example uses two files:

- `file1.cpp`:

```
typedef struct S //Noncompliant
{
    int x;
    int y;
```

```
    }S;
    void foo(S& s){
    //...
    }
```

- `file2.cpp`:

```
    typedef struct S
    {
        int y;
        int x;
    }S ;
    void bar(S& s){
    //...
    }
```

In this example, both `file1.cpp` and `file2.cpp` define the structure S. However, the definitions switch the order of the structure fields.

Check Information

Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-2-3

A type, object or function that is used in multiple translation units shall be declared in one and only one file

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Rationale

If you declare an identifier in a header file, you can include the header file in any translation unit where the identifier is defined or used. In this way, you ensure consistency between:

- The declaration and the definition.
- The declarations in different translation units.

The rule enforces the practice of declaring external objects or functions in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-2-4

An identifier with external linkage shall have exactly one definition

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

An identifier with external linkage shall have exactly one definition.

Rationale

If an identifier has multiple definitions or no definitions, it can lead to undefined behavior.

Polyspace Implementation

The checker is not raised on unused code such as

- Noninstantiated templates
- Uncalled `static extern` or functions
- Uncalled and undefined local functions
- Unused types and variables

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Multiple Definitions of Identifier

This example uses two files:

- `file1.cpp`:

```
typedef signed    int         int32_t;

namespace NS {
    extern int32_t a;

    void foo(){
        a = 0;

    }
};
```
- `file2.cpp`:

```
typedef signed    int         int32_t;
```

```
typedef signed   long long          int64_t;

namespace NS {
    extern int64_t a; //Noncompliant
    void bar(){
        ++a;

    }
};
```

The same identifier a is defined in both files.

Check Information
Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-3-1

Objects or functions with external linkage shall be declared in a header file

Description

Rule Definition

Objects or functions with external linkage shall be declared in a header file.

Rationale

If you declare a function or object in a header file, it is clear that the function or object is meant to be accessed in multiple translation units. If you intend to access the function or object from a single translation unit, declare it `static` or in an unnamed namespace.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Declaration in Header File Missing

This example uses two files:

- `decls.h`:

  ```
  extern int x;
  ```
- `file.cpp`:

  ```
  #include "decls.h"

  int x = 0;
  int y = 0; //Noncompliant
  static int z = 0;
  ```

In this example, the variable `x` is declared in a header file but the variable `y` is not. The variable `z` is also not declared in a header file but it is declared with the `static` specifier and does not have external linkage.

Check Information
Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-3-2

If a function has internal linkage then all re-declarations shall include the static storage class specifier

Description

Rule Definition

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Rationale

If a function declaration has the `static` storage class specifier, it has internal linkage. Subsequent redeclarations of the function have internal linkage even without the `static` specifier.

However, if you do not specify the `static` keyword explicitly, it is not immediately clear from a declaration whether the function has internal linkage.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Missing `static` Specifier from Redeclaration

```
static void func1 ();
static void func2 ();

void func1() {}  //Noncompliant
static void func2() {}
```

In this example, the function `func1` is declared `static` but defined without the `static` specifier.

Check Information
Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-4-1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility

Description

Rule Definition

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Rationale

Defining variables with the minimum possible block scope reduces the possibility that they might later be accessed unintentionally.

For instance, if an object is meant to be accessed in one function only, declare the object local to the function.

Polyspace Implementation

The rule checker determines if an object is used in one block only. If the object is used in one block but defined outside the block, the checker raises a violation.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of Global Variable in Single Function

```
static int countReset; //Noncompliant

volatile int check;

void increaseCount() {
    int count = countReset;
    while(check%2) {
     count++;
    }
}
```

In this example, the variable `countReset` is declared global used in one function only. A compliant solution declares the variable local to the function to reduce its visibility.

Check Information
Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-9-1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations

Description

Rule Definition

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Rationale

If a redeclaration is not token-for-token identical to the previous declaration, it is not clear from visual inspection which object or function is being redeclared.

Polyspace Implementation

The rule checker compares the current declaration with the last seen declaration.

Additional Message in Report

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Variable *varName* is not compatible with previous declaration.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Identical Declarations That Do Not Match Token for Token

```
typedef int* intptr;

int* map;
extern intptr map; //Noncompliant

intptr table;
extern intptr table; //Compliant
```

In this example, the variable `map` is declared twice. The second declaration uses a `typedef` which resolves to the type of the first declaration. Because of the `typedef`, the second declaration is not token-for-token identical to the first.

Check Information
Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-9-2

typedefs that indicate size and signedness should be used in place of the basic numerical types

Description

Rule Definition

typedefs that indicate size and signedness should be used in place of the basic numerical types.

Rationale

When the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

Polyspace Implementation

The rule checker does not raise violations in templates that are not instantiated.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Direct Use of Basic Numerical Types

```
typedef unsigned int uint32_t;

unsigned int x = 0;        //Noncompliant
uint32_t y = 0;  //Compliant
```

In this example, the declaration of x is noncompliant because it uses the basic type int directly.

Check Information
Group: Basic Concepts
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 3-9-3

The underlying bit representations of floating-point values shall not be used

Description

Rule Definition

The underlying bit representations of floating-point values shall not be used.

Rationale

The underlying bit representations of floating point values vary across compilers. If you directly use the underlying representation of floating point values, your program is not portable across implementations.

Polyspace Implementation

The rule checker flags conversions from pointers to floating point types into pointers to integer types, and vice versa.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Using Underlying Representation of Floating-Point Values

```
float fabs2(float f) {
    unsigned int* ptr = reinterpret_cast <unsigned int*> (&f); //Noncompliant
    *(ptr + 3) &= 0x7f;
    return f;
}
```

In this example, the `reinterpret_cast` attempts to cast a floating-point value to an integer and access the underlying bit representation of the floating point value.

Check Information
Group: Basic Concepts
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 4-5-1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator

Description

Rule Definition

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Rationale

Operators other than the ones mentioned in the rule do not produce meaningful results with `bool` operands. Use of `bool` operands with these operators can indicate programming errors. For instance, you intended to use the logical operator || but used the bitwise operator | instead.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Compliant and Noncompliant Uses of bool Operands

```
void boolOperations() {
    bool lhs = true;
    bool rhs = false;

    int res;

    if(lhs & rhs) {} //Noncompliant
    if(lhs < rhs) {} //Noncompliant
    if(~rhs) {}      //Noncompliant
    if(lhs ^ rhs) {} //Noncompliant
    if(lhs == rhs) {} //Compliant
    if(!rhs) {}       //Compliant
    res = lhs? -1:1;  //Compliant
}
```

In this example, `bool` operands do not violate the rule when used with the ==, ! and the ? operators.

Check Information
Group: Standard Conversions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 4-5-2

Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=

Description

Rule Definition

Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Standard Conversions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 4-5-3

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator N

Description

Rule Definition

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N

Rationale

The C++03 Standard only requires that the characters '0' to '9' have consecutive values. Other characters do not have well-defined values. If you use these characters in operations other than the ones mentioned in the rule, you implicitly use their underlying values and might see unexpected results.

Additional Message in Report

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Compliant and Noncompliant Uses of Character Operands

```
void charManipulations (char ch) {

    char initChar = 'a'; //Compliant
    char finalChar = 'z'; //Compliant

    if(ch == initChar) {} //Compliant
    if( (ch >= initChar) && (ch <= finalChar)) {} //Noncompliant
    else if( (ch >= '0') && (ch <= '9') ) {} //Compliant by exception
}
```

In this example, character operands do not violate the rule when used with the = and == operators. Character operands can also be used with relational operators as long as the comparison is performed with the digits '0' to '9'.

Check Information
Group: Standard Conversions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 4-10-1

NULL shall not be used as an integer value

Description

Rule Definition

NULL shall not be used as an integer value.

Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of NULL to null pointer constants. `MISRA C++:2008 Rule 4-10-2` restricts the use of the literal 0 to integers.

Polyspace Implementation

The checker flags assignment of NULL to an integer variable or binary operations involving NULL and an integer. Assignments can be direct or indirect such as passing NULL as integer argument to a function.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Compliant and Noncompliant Uses of NULL

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(NULL); //Noncompliant
    checkPointer(NULL); //Compliant
}
```

In this example, the use of `NULL` as argument to the `checkInteger` function is noncompliant because the function expects an `int` argument.

Check Information
Group: Standard Conversions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2018a

MISRA C++:2008 Rule 4-10-2

Literal zero (0) shall not be used as the null-pointer-constant

Description

Rule Definition

Literal zero (0) shall not be used as the null-pointer-constant.

Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of the literal 0 to integers. MISRA C++:2008 Rule 4-10-1 restricts the use of NULL to null pointer constants.

Polyspace Implementation

The checker flags assignment of 0 to a pointer variable or binary operations involving 0 and a pointer. Assignments can be direct or indirect such as passing 0 as pointer argument to a function.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Compliant and Noncompliant Uses of Literal 0

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(0); //Compliant
    checkPointer(0); //Noncompliant
}
```

In this example, the use of 0 as argument to the checkPointer function is noncompliant because the function expects an int * argument.

Check Information
Group: Standard Conversions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2018a

MISRA C++:2008 Rule 5-0-1

The value of an expression shall be the same under any order of evaluation that the standard permits

Description

Rule Definition

The value of an expression shall be the same under any order of evaluation that the standard permits.

Rationale

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

Polyspace Implementation

Polyspace raises a violation if an expression satisfies any of these conditions:

- The same variable is modified more than once in the expression or it is both read and written.
- The expression allows more than one order of evaluation.
- The expression contains a single `volatile` object that occurs multiple times.
- The expression contains more than one `volatile` object.

Because `volatile` objects can change their value at anytime, an expression containing multiple `volatile` variables or multiple instances of the same `volatile` variable might have different results depending on the order of evaluation.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)]) // Non-compliant

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         // Compliant
    COPY_ELEMENT (i++);    // Violation happens on this line but macro definition is flagged.
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified. Because `COPY_ELEMENT` is a macro, Polyspace flags the macro definition and highlights the line where the violation occurs.

Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
```

```
    unsigned int i=0;
    f ( i++, i );                     // Noncompliant
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

Multiple volatile Objects in an Expression

```
volatile int a, b;
int mathOp(int x, int y);

int foo(void){
    int temp = mathOp(5,a) + mathOp(6,b);//Noncompliant
    return temp * mathOp(a,a);//Noncompliant
}
```

In this example, this rule is violated twice.

• The declaration of `temp` uses two `volatile` objects in the expression. Because the value of `volatile` objects might change at any time, the expression might evaluate to different values depending on the order of evaluation. Polyspace flags the second `volatile` object in the expression.

• The `return` statement uses the same `volatile` object twice. Because the expression might have different results depending on the order of evaluation, Polyspace raises this defect.

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-2

Limited dependence should be placed on C++ operator precedence rules in expressions

Description

Rule Definition

Limited dependence should be placed on C++ operator precedence rules in expressions.

Rationale

Use parentheses to clearly indicate the order of evaluation.

Depending on operator precedence can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:
  - In the operation *p++, it is possible that you expect the dereferenced value to be incremented. However, the pointer p is incremented before the dereference.
  - In the operation (x == y | z), it is possible that you expect x to be compared with y | z. However, the == operation happens before the | operation.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Evaluation Order Dependent on Operator Precedence Rules

```
#include <cstdio>

void showbits(unsigned int x) {
    for(int i = (sizeof(int) * 8) - 1; i >= 0; i--) {
        (x & 1u << i) ? putchar('1') : putchar('0'); // Noncompliant
    }
    printf("\n");
}
```

In this example, the checker flags the operation x & 1u << i because the statement relies on operator precedence rules for the << operation to happen before the & operation. If this is the intended order, the operation can be rewritten as x & (1u << i).

Check Information
Group: Expressions
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-3

A cvalue expression shall not be implicitly converted to a different underlying type

Description

Rule Definition

A cvalue expression shall not be implicitly converted to a different underlying type.

Rationale

This rule ensures that the result of the expression does not overflow when converted to a different type.

Polyspace Implementation

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

The underlying data type of a cvalue expression is the widest of operand data types in the expression. For instance, if you add two variables, one of type `int8_t` (`typedef` for `char`) and another of type `int32_t` (`typedef` for `int`), the addition has underlying type `int32_t`. If you assign the sum to a variable of type `int8_t`, the rule is violated.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Implicit Conversion of Cvalue Expression

```
#include<cstdint>

void func ( )
  {
    int32_t s32;
    int8_t s8;
    s32 = s8 + s8; //Noncompliant
    s32 = s32 + s8; //Compliant
  }
```

In this example, the rule is violated when two variables of type `int8_t` are added and the result is assigned to a variable of type `int32_t`. The underlying type of the addition does not take into account the integer promotion involved and is simply the widest of operand data types, in this case, `int8_t`.

The rule is not violated if one of the operands has type `int32_t` and the result is assigned to a variable of type `int32_t`. In this case, the underlying data type of the addition is the same as the type of the variable to which the result is assigned.

Check Information

Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-4

An implicit integral conversion shall not change the signedness of the underlying type

Description

Rule Definition

An implicit integral conversion shall not change the signedness of the underlying type.

Rationale

Some conversions from signed to unsigned data types can lead to implementation-defined behavior. You can see unexpected results from the conversion.

Polyspace Implementation

The checker flags implicit conversions from a signed to an unsigned integer data type or vice versa.

The checker assumes that `ptrdiff_t` is a signed integer.

Additional Message in Report

An implicit integral conversion shall not change the signedness of the underlying type.

Implicit conversion of one of the binary + operands whose underlying types are *typename_1* and *typename_2*.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Implicit Conversions that Change Signedness

```
typedef char int8_t;
typedef unsigned char uint8_t;

void func()
  {
    int8_t s8;
    uint8_t u8;

    s8 = u8; //Noncompliant
    u8 = s8 + u8; //Noncompliant
    u8 = static_cast< uint8_t > ( s8 ) + u8; //Compliant
}
```

In this example, the rule is violated when a variable with a variable with signed data type is implicitly converted to a variable with unsigned data type or vice versa. If the conversion is explicit, as in the preceding example, the rule violation does not occur.

Check Information

Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-5

There shall be no implicit floating-integral conversions

Description

Rule Definition

There shall be no implicit floating-integral conversions.

Rationale

If you convert from a floating point to an integer type, you lose information. Unless you explicitly cast from floating point to an integer type, it is not clear whether the loss of information is intended. Additionally, if the floating-point value cannot be represented in the integer type, the behavior is undefined.

Conversion from an integer to floating-point type can result in an inexact representation of the value. The error from conversion can accumulate over later operations and lead to unexpected results.

Polyspace Implementation

The checker flags implicit conversions between floating-point types (`float` and `double`) and integer types (`short`, `int`, etc.).

This rule takes precedence over 5-0-4 and 5-0-6 if they apply at the same time.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Conversion Between Floating Point and Integer Types

```
typedef signed int int32_t;
typedef float float32_t;

void func ( )
  {
    float32_t f32;
    int32_t   s32;
    s32 = f32;   //Noncompliant
    f32 = s32;   //Noncompliant
    f32 = static_cast< float32_t > ( s32 ); //Compliant
  }
```

In this example, the rule is violated when a floating-point type is *implicitly* converted to an integer type. The violation does not occur if the conversion is explicit.

Check Information
Group: Expressions

Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type

Description

Rule Definition

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Rationale

A conversion that reduces the size of the underlying type can result in loss of information. Unless you explicitly cast to the narrower type, it is not clear whether the loss of information is intended.

Polyspace Implementation

The checker flags implicit conversions that reduce the size of a type.

If the conversion is to a narrower integer with a different sign, then rule 5-0-4 takes precedence over rule 5-0-6. Only rule 5-0-4 is shown.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Conversion That Reduces Size of Type

```
typedef signed short int16_t;
typedef signed int int32_t;

void func ( )
  {
    int16_t    s16;;
    int32_t    s32;
    s16 = s32;    //Noncompliant
    s16 = static_cast< int16_t > ( s32 ); //Compliant
  }
```

In this example, the rule is violated when a type is *implicitly* converted to a narrower type. The violation does not occur if the conversion is explicit.

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-7

There shall be no explicit floating-integral conversions of a cvalue expression

Description

Rule Definition

There shall be no explicit floating-integral conversions of a cvalue expression.

Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression). For instance, in this example, the result of an integer division is then cast to a floating-point type.

```
short num;
short den;
float res;
res= static_cast<float> (num/den);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a floating-point division because of the later cast.

Additional Message in Report

There shall be no explicit floating-integral conversions of a cvalue expression.

Complex expression of underlying type *typeBeforeConversion* may only be cast to narrower integer type of same signedness, however the destination type is *typeAfterconversion*.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Conversion of Division Result from Integer to Floating Point

```
void func() {
    short num;
    short den;
    short res_short;
    float res_float;

    res_float = static_cast<float> (num/den); //Noncompliant

    res_short = num/den;
    res_float = static_cast<float> (res_short); //Compliant

}
```

In this example, the first cast on the division result violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the expression is evaluated with an underlying type `float`.
- The second cast makes it clear that the expression is evaluated with the underlying type `short`. The result is then cast to the type `float`.

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression

Description

Rule Definition

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression). For instance, in this example, the sum of two `short` operands is cast to the wider type `int`.

```
short op1;
short op2;
int res;
res= static_cast<int> (op1 + op2);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a sum with the underlying type `int` because of the later cast.

Additional Message in Report

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Complex expression of underlying type *typeBeforeConversion* may only be cast to narrower integer type of same signedness, however the destination type is *typeAfterconversion*.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Conversion of Sum to Wider Integer Type

```
void func() {
    short op1;
    short op2;
    int res;

    res = static_cast<int> (op1 + op2); //Noncompliant
    res = static_cast<int> (op1) + op2; //Compliant
```

```
}
```

In this example, the first cast on the sum violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the sum is evaluated with an underlying type `int`.
- The second cast first converts one of the operands to `int` so that the sum is actually evaluated with the underlying type `int`.

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression

Description

Rule Definition

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression).. For instance, in this example, the sum of two `unsigned int` operands is cast to the type `int`.

```
unsigned int op1;
unsigned int op2;
int res;
res= static_cast<int> (op1 + op2);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a sum with the underlying type `int` because of the later cast.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Conversion of Sum to Wider Integer Type

```
typedef int int32_t;
typedef unsigned int uint32_t;

void func() {
    uint32_t op1;
    uint32_t op2;
    int32_t res;

    res = static_cast<int32_t> (op1 + op2); //Noncompliant
    res = static_cast<int32_t> (op1) +
          static_cast<int32_t> (op2); //Compliant

}
```

In this example, the first cast on the sum violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the sum is evaluated with an underlying type `int32_t`.
- The second cast first converts each of the operands to `int32_t` so that the sum is actually evaluated with the underlying type `int32_t`.

Check Information

Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-10

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand

Description

Rule Definition

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-11

The plain char type shall only be used for the storage and use of character values

Description

Rule Definition

The plain char type shall only be used for the storage and use of character values.

Polyspace Implementation

The checker raises a violation when a value of signed or unsigned integer type is implicitly converted to the plain char type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2015a

MISRA C++:2008 Rule 5-0-12

Signed char and unsigned char type shall only be used for the storage and use of numeric values

Description

Rule Definition

Signed char and unsigned char type shall only be used for the storage and use of numeric values.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2015a

MISRA C++:2008 Rule 5-0-13

The condition of an if-statement and the condition of an iteration- statement shall have type bool

Description

Rule Definition

The condition of an if-statement and the condition of an iteration- statement shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-14

The first operand of a conditional-operator shall have type bool

Description

Rule Definition

The first operand of a conditional-operator shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-15

Array indexing shall be the only form of pointer arithmetic

Description

Rule Definition

Array indexing shall be the only form of pointer arithmetic.

Rationale

You can traverse an array in two ways:

- Increment or decrement an array index, and then use the array index to access an element.
- Increment or decrement a pointer to the array and then dereference the pointer.

The first method is clearer and less error-prone.

All other forms of explicit pointer arithmetic introduce the risk of accessing unintended memory locations.

Polyspace Implementation

The checker flags:

- Arithmetic operations on all pointers, for instance `p+I`, `I+p` and `p-I`, where `p` is a pointer and `I` an integer..
- Array indexing on nonarray pointers.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-16

A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Description

Rule Definition

A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Rationale

It is undefined behavior when the result of a pointer arithmetic operation that uses a pointer to an array element does not point to either:

- An element of the array.
- One past the last element of the array. For instance:

```
int arr[3];
int* res;
res = arr+3; // res points to one beyond arr
```

The rule applies to these operations. `ptr` is a pointer to an array element and `int_exp` is an integer expression.

- `ptr + int_exp`
- `int_exp + ptr`
- `ptr - int_exp`
- `ptr + +`
- `++ptr`
- `--ptr`
- `ptr--`
- `ptr[ int_exp ]`

Polyspace Implementation

- Single objects that are not part of an array are considered arrays of one element. For instance, in this code example, `arr_one` is equivalent to an array of one element. Polyspace does not flag the increment of pointer `ptr_to_one` because it points to one past the last element of `arr_one`.

```
void f_incr(int* x){
    int* ptr_to_one = x;
    ++ptr_to_one;  // Compliant
}

void func(){
    int arr_one=1; // Equivalent to array of one element
    f_incr(&arr_one);
}
```

- Polyspace does not flag the use of pointer parameters in pointer arithmetic operations when those pointers point to arrays. For instance, in this code snippet, the use of `&a1[2]` in `f1` is compliant when you pass an array to `f1`.

```
void f1( int* const a1){
     int* b= &a1[2]; // Compliant
}
void f2(){
    int arr[3] {};
    f1(arr);
}
```

- In structures with multiple elements, Polyspace does not flag the result of a pointer arithmetic operation on an element that results in a pointer that points to a different element if the pointer points within the allocated memory of the structure or to one past the last element of the structure.

  For instance, in this code snippet, the assignment to `ptr_to_struct` is compliant because it remains inside `myStruct`, even if it points outside `myStruct.elem1`. Using an index larger than the element dimension to access the content of that element is not compliant, even if the resulting address is within the allocated memory of the structure.

```
void func(){
    struct {
        char elem1[10];
        char elem2[10];
    } myStruct;

    char* ptr_to_struct = &myStruct.elem1[11]; //Compliant
      // Address of myStruct.elem1[11] is inside myStruct
        char val_to_struct = myStruct.elem1[11]; // Non-compliant
}
```

- In multidimensional arrays, Polyspace flags any use of indices that are larger than a subarray dimension to access an element of that subarray. Polyspace does not flag the assignment of the address of that same subarray element if the address is inside the allocated memory of the top-level array.

  For example, in this code snippet, the assignment to pointer `ptr_to_arr` is compliant because the pointer points to an address that is within the allocated memory of `multi_arr`. The assignment to variable `arr_val` is not compliant because the index used to access the subarray element (3) is larger than the dimension of the subarray (2).

```
void func(){
    int multi_arr[5][2];

        // Assigned memory is inside top level array
    int* ptr_to_arr = &multi_arr[2][3]; //Compliant

    // Use of index 3 with subarray of size 2
    int arr_val = multi_arr[2][3]; // Non-compliant
}
```

- Polyspace flags the dereference of a pointer when that pointer points to one past the last element of an array. For instance, in this code snippet, the assignment of `ptr` is compliant, but the dereference of `ptr` is not. `tab+3` is one past the last element of tab.

```
void derefPtr(){
    int tab[3] {};
```

```
        int* ptr = tab+3; //Compliant
        int res = *(tab+3); // Non-compliant
    }
```

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Pointer Arithmetic by Using Pointers to Array Elements

```
void f_incr(int* x)
{
    int* ptr_to_one = x;
    ++ptr_to_one;  // Compliant
}

void f1(int* const a1)
{
    int* b = &a1[2]; // Compliant
}

int main()
{

    int arr_one = 1; // Equivalent to array of one element
    f_incr(&arr_one);

    int arr[3] {};
    f1(arr);

    struct {
        char elem1[10];
        char elem2[10];
    } myStruct;

    char* ptr_to_struct = &myStruct.elem1[11]; // Compliant
    ptr_to_struct = &myStruct.elem2[11]; //Non-compliant

    int tab[3] {1, 2, 3};
    int* ptr =  &tab[2];
    int res = tab[2];
    ++ptr; // Compliant
    res = *ptr; //Non-compliant

    return 0;
}
```

In this example:

- The increment of `ptr_to_one` inside `f_incr()` is compliant because the operation results in a pointer that points to one past the last element of array `x`. The integer that is passed to `f_incr()` is equivalent to an array of one element.
- The operation on pointer parameter `a1` inside `f1()` is compliant because the pointer points to array `arr`.

- The first assignment of `ptr_to_struct` is compliant because `elem1[11]` is still inside `myStruct`. The second assignment of `ptr_to_struct` is not compliant because the result of the operation does not point to either inside `myStruct` or to one past the last element of `myStruct`.

- The increment of `ptr` is compliant because the result of the operation points to one past the last element of `tab`. The dereference of `ptr` on the next line is not compliant.

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2021a

MISRA C++:2008 Rule 5-0-17

Subtraction between pointers shall only be applied to pointers that address elements of the same array

Description

Rule Definition

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Polyspace Implementation

Use Bug Finder for this checker. The rule checker performs the same checks as `Subtraction or comparison between pointers to different arrays`. Code Prover can fail to detect some violations.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-18

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array

Description

Rule Definition

>, >=, <, <= *shall not be applied to objects of pointer type, except where they point to the same array.*

Polyspace Implementation

Use Bug Finder for this checker. The rule checker performs the same checks as `Subtraction or comparison between pointers to different arrays`. Code Prover can fail to detect some violations.

The checker ignores casts when showing the violation on relational operator use with pointers types.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-19

The declaration of objects shall contain no more than two levels of pointer indirection

Description

Rule Definition

The declaration of objects shall contain no more than two levels of pointer indirection.

Rationale

If you use pointers with more than two levels of indirection, a developer reading the code might find it difficult to understand the behavior of the code.

Polyspace Implementation

Polyspace flags all declarations of objects that contain more than two levels of pointer indirection.

*   If you use type aliases, the checker includes pointer indirections from the alias in the evaluation of the level of indirection. For instance, in this code snippet, the declaration of `var` is non-compliant. The type of `var` is `const` pointer to a `const` pointer to a pointer to `char`, which is three levels of pointer indirection. The declaration of `var2` has two levels of pointer indirection and is compliant.

    ```
    using ptrToChar = char*;

    void func()
    {
        ptrToChar* const* const var = nullptr; //Non-compliant, 3 levels of indirection
        char* const* const var2 = nullptr; //Compliant, 2 levels of indirection
        //...
    }
    ```

*   If you pass an array to a function, the conversion of the array to a pointer to the first element of the array is included in the evaluation of the level of indirection. For instance, in this code snippet, parameter `arrParam` is non-compliant. The type of `arrParam` is a pointer to a pointer to a pointer to `char` (three levels of pointer indirection). The declaration of `arrVar` is compliant because `arrVar` has type array of pointer to pointer to char (two levels of pointer indirection).

    ```
    void func(char** arrParam[])  //Non-compliant
    {
        //...
        char** arrVar[5]; //Compliant
    }
    ```

This checker does not flag the use of objects with more than two levels of indirection. For instance, in this code snippet, the declaration of `var` is non-compliant, but the evaluation of the size of `var` is compliant.

```
#include<iostream>


using charToPtr = char*;

void func()
{
    charToPtr* const* const var = nullptr; //Non-compliant
```

```
    std::cout << sizeof(var) << std::endl; //Compliant

}
```

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-20

Non-constant operands to a binary bitwise operator shall have the same underlying type

Description

Rule Definition

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-0-21

Bitwise operators shall only be applied to operands of unsigned underlying type

Description

Rule Definition

Bitwise operators shall only be applied to operands of unsigned underlying type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-1

Each operand of a logical && or || shall be a postfix-expression

Description

Rule Definition

Each operand of a logical && or || shall be a postfix-expression.

Rationale

This rule effectively requires that operands of a logical && or || operation be appropriately parenthesized. For instance, instead of `a + b || c`, the rule requires `(a + b) || c` or `a + (b || c)`. In both compliant cases, the left operand of `||`, that is `(a + b)` or `b`, is a primary expression and therefore also a postfix expression. For more information on postfix expressions, see the C++03 Standard (Section 5.2).

Enclosing operands in parentheses improves readability of code and makes sure that the operations occur in the order that the developer intends.

Polyspace Implementation

The checker raises a violation if a logical && or || operand is not a postfix expression.

A postfix expression can be a primary expression such as a simple identifier or a combination of identifiers enclosed in parentheses, but also one of the following:

- Function call such as `func()`.
- Array element access such as `arr[]`.
- Structure member access such as `aStructVar.aMember`.

For the complete list of postfix expressions, see the C++03 Standard (Section 5.2).

The checker allows exceptions on associative chains such as `(a && b && c)` or `(a || b || c)`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Noncompliant and Compliant Expressions Involving Logical Operations

```
bool Operations(bool a, bool b, bool c, bool priority) {
    bool res;
    if(priority) {
        res = a && b || c;  //Noncompliant
    }
    else {
        res = a && (b || c); //Compliant
    }
```

```
    return res;
}
```

In this example, the expression a && b || c violates the rule because the right operand of && and the left operand of || are not postfix expressions.

Check Information

Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast

Description

Rule Definition

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-3

Casts from a base class to a derived class should not be performed on polymorphic types

Description

Rule Definition

Casts from a base class to a derived class should not be performed on polymorphic types.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-4

C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used

Description

Rule Definition

C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-5

A cast shall not remove any const or volatile qualification from the type of a pointer or reference

Description

Rule Definition

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type

Description

Rule Definition

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-7

An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly

Description

Rule Definition

An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.

Polyspace Implementation

The checker flags all pointer conversions including between a pointer to a `struct` object and a pointer to the first member of the same `struct` type.

Indirect conversions from a pointer to non-pointer type are not detected.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-8

An object with integer type or pointer to void type shall not be converted to an object with pointer type

Description

Rule Definition

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Polyspace Implementation

The checker allows an exception on zero constants.

Objects with pointer type include objects with pointer-to-function type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-9

A cast should not convert a pointer type to an integral type

Description

Rule Definition

A cast should not convert a pointer type to an integral type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-10

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression

Description

Rule Definition

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Expressions
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-11

The comma operator, && operator and the || operator shall not be overloaded

Description

Rule Definition

The comma operator, && operator and the || operator shall not be overloaded.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-2-12

An identifier with array type passed as a function argument shall not decay to a pointer

Description

Rule Definition

An identifier with array type passed as a function argument shall not decay to a pointer.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-3-1

Each operand of the ! operator, the logical && or the logical || operators shall have type bool

Description

Rule Definition

Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-3-2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned

Description

Rule Definition

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-3-3

The unary & operator shall not be overloaded

Description

Rule Definition

The unary & operator shall not be overloaded.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-3-4

Evaluation of the operand to the sizeof operator shall not contain side effects

Description

Rule Definition

Evaluation of the operand to the sizeof operator shall not contain side effects.

Polyspace Implementation

The checker does not show a warning on volatile accesses and function calls

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-8-1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand

Description

Rule Definition

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-14-1

The right hand operand of a logical && or || operator shall not contain side effects

Description

Rule Definition

The right hand operand of a logical && or || operator shall not contain side effects.

Rationale

When evaluated, an expression with side effect modifies at least one of the variables in the expression. For instance, n++ is an expression with side effect.

The right-hand operand of a:

- Logical && operator is evaluated only if the left-hand operand evaluates to true.
- Logical || operator is evaluated only if the left-hand operand evaluates to false.

In other cases, the right-hand operands are not evaluated, so side effects of the expression do not take place. If your program relies on the side effects, you might see unexpected results in those cases.

Polyspace Implementation

The checker flags logical && or || operators whose right-hand operands are expressions with side effects.

The checker does not consider volatile accesses and function calls as potential side effects.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-18-1

The comma operator shall not be used

Description

Rule Definition

The comma operator shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 5-19-1

Evaluation of constant unsigned integer expressions should not lead to wrap-around

Description

Rule Definition

Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Expressions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-2-1

Assignment operators shall not be used in sub-expressions

Description

Rule Definition

Assignment operators shall not be used in sub-expressions.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-2-2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality

Description

Rule Definition

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Polyspace Implementation

The checker detects the use of == or != with floating-point variables or expressions. The checker does not detect indirectly testing of equality, for instance, using the <= operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-2-3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character

Description

Rule Definition

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.

Polyspace Implementation

The checker considers a null statement as a line where the first character excluding comments is a semicolon. The checker flags situations where:

- Comments appear before the semicolon.

  For instance:

  ```
  /* wait for pin */ ;
  ```

- Comments appear immediately after the semicolon without a white space in between.

  For instance:

  ```
  ;// wait for pin
  ```

The checker also shows a violation when a second statement appears on the same line following the null statement.

For instance:

```
; count++;
```

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-3-1

The statement forming the body of a switch, while, do while or for statement shall be a compound statement

Description

Rule Definition

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Rationale

A compound statement is included in braces.

If a block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

This checker enforces the practice of adding braces following a selection or iteration statement even for a single line in the body. Later, when more lines are added, the developer adding them does not need to note the absence of braces and include them.

Polyspace Implementation

The checker flags `for` loops where the first token following a `for` statement is not a left brace, for instance:

```
for (i=init_val; i > 0; i--)
    if (arr[i] < 0)
        arr[i] = 0;
```

Similar checks are performed for `switch`, `for` and `do..while` statements.

The second line of the message on the **Result Details** pane indicates which statement is violating the rule. For instance, in the preceding example, the second line of the message states that the `for` loop is violating the rule.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-4-1

An if (condition) construct shall be followed by a compound statement The else keyword shall be followed by either a compound statement, or another if statement

Description

Rule Definition

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Additional Message in Report

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-4-2

All if else if constructs shall be terminated with an else clause

Description

Rule Definition

All if ... else if constructs shall be terminated with an else clause.

Additional Message in Report

All if ... else if constructs shall be terminated with an else clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-4-3

A switch statement shall be a well-formed switch statement

Description

Rule Definition

A switch statement shall be a well-formed switch statement.

Polyspace Implementation

The checker flags these situations:

- A statement occurs between the `switch` statement and the first `case` statement.

  For instance:

  ```
  switch(ch) {
    int temp;
    case 1:
      break;
    default:
      break;
  }
  ```

- A label or a jump statement such as `goto` or `return` occurs in the `switch` block.

- A variable is declared in a `case` statement (outside any block).

  For instance:

  ```
  switch(ch) {
    case 1:
      int temp;
      break;
    default:
      break;
  }
  ```

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-4-4

A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Description

Rule Definition

A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-4-5

An unconditional throw or break statement shall terminate every non - empty switch-clause

Description

Rule Definition

An unconditional throw or break statement shall terminate every non - empty switch-clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-4-6

The final clause of a switch statement shall be the default-clause

Description

Rule Definition

The final clause of a switch statement shall be the default-clause.

Polyspace Implementation

The checker detects `switch` statements that do not have a final `default` clause.

The checker does not raise a violation if the `switch` variable is an `enum` with finite number of values and you have a `case` clause for each value. For instance:

```
enum Colours { RED, BLUE, GREEN } colour;

switch ( colour ) {
      case RED:
        break;
      case BLUE:
        break;
      case GREEN:
        break;
}
```

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-4-7

The condition of a switch statement shall not have bool type

Description

Rule Definition

The condition of a switch statement shall not have bool type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-4-8

Every switch statement shall have at least one case-clause

Description

Rule Definition

Every switch statement shall have at least one case-clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-5-1

A for loop shall contain a single loop-counter which shall not have floating type

Description

Rule Definition

A for loop shall contain a single loop-counter which shall not have floating type.

Polyspace Implementation

The checker flags these situations:

- The `for` loop index has a floating point type.

- More than one loop counter is incremented in the `for` loop increment statement.

  For instance:

  `for(i=0, j=0; i<10 && j < 10;i++, j++) {}`

- A loop counter is not incremented in the `for` loop increment statement.

  For instance:

  `for(i=0; i<10;) {}`

  Even if you increment the loop counter in the loop body, the checker still raises a violation. According to the MISRA C++ specifications, a loop counter is one that is initialized in or prior to the loop expression, acts as an operand to a relational operator in the loop expression and *is modified in the loop expression*. If the increment statement in the loop expression is missing, the checker cannot find the loop counter modification and considers as if a loop counter is not present.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-5-2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=

Description

Rule Definition

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-5-3

The loop-counter shall not be modified within condition or statement

Description

Rule Definition

The loop-counter shall not be modified within condition or statement.

Rationale

The `for` loop has a specific syntax for modifying the loop counter. A code reviewer expects modification using that syntax. Modifying the loop counter elsewhere can make the code harder to review.

Polyspace Implementation

The checker flags modification of a `for` loop counter in the loop body or the loop condition (the condition that is checked to see if the loop must be terminated).

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-5-4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop

Description

Rule Definition

The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-5-5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression

Description

Rule Definition

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-5-6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool

Description

Rule Definition

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-6-1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement

Description

Rule Definition

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-6-2

The goto statement shall jump to a label declared later in the same function body

Description

Rule Definition

The goto statement shall jump to a label declared later in the same function body.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-6-3

The continue statement shall only be used within a well-formed for loop

Description

Rule Definition

The continue statement shall only be used within a well-formed for loop.

Polyspace Implementation

The checker flags the use of `continue` statements in:

- `for` loops that are not well-formed, that is, loops that violate rules 6-5-x.
- `while` loops.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-6-4

For any iteration statement there shall be no more than one break or goto statement used for loop termination

Description

Rule Definition

For any iteration statement there shall be no more than one break or goto statement used for loop termination.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 6-6-5

A function shall have a single point of exit at the end of the function

Description

Rule Definition

A function shall have a single point of exit at the end of the function.

Rationale

This rule requires that a `return` statement must occur as the last statement in the function body. Otherwise, the following issues can occur:

- Code following a `return` statement can be unintentionally omitted.
- If a function that modifies some of its arguments has early `return` statements, when reading the code, it is not immediately clear which modifications actually occur.

Polyspace Implementation

The checker flags these situations:

- A function has more than one `return` statement.
- A non-`void` function has one `return` statement only but the `return` statement is not the last statement in the function.

A `void` function need not have a `return` statement. If a `return` statement exists, it need not be the last statement in the function.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Statements
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-1-1

A variable which is not modified shall be const qualified

Description

Rule Definition

A variable which is not modified shall be const qualified.

Rationale

Declaring a variable `const` reduces the chances that you modify the variable by accident.

Polyspace Implementation

The checker flags function parameters or local variables that are not const-qualified but never modified in the function body. Function parameters of integer, float, enum and boolean types are not flagged.

If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. These variables are not flagged.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Unmodified Local Variable

```
#include <string.h>

char returnNthCharacter (int n) {
    char* pwd = "aXeWdf10fg" ; //Noncompliant
    char nthCharacter;

    for(int i=0; i < strlen(pwd); i++) {
        if(i==n)
            nthCharacter = pwd[i];
    }
    return nthCharacter;
}
```

In this example, the pointer `pwd` is not `const`-qualified. However, beyond initialization with a constant, it is not reassigned anywhere in the `returnNthCharacter` function.

Check Information
Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2018a

MISRA C++:2008 Rule 7-1-2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified

Description

Rule Definition

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Polyspace Implementation

The checker flags pointers where the underlying object is not const-qualified but never modified in the function body.

If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. Pointers that point to these variables are not flagged.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2018a

MISRA C++:2008 Rule 7-3-1

The global namespace shall only contain main, namespace declarations and extern "C" declarations

Description

Rule Definition

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Rationale

The rule makes sure that all names found at global scope are part of a namespace. Adhering to this rule avoids name clashes and ensures that developers do not reuse a variable name, resulting in compilation/linking errors, or shadow a variable name, resulting in possibly unexpected issues later.

Polyspace Implementation

Other than the `main` function, the checker flags all names used at global scope that are not part of a namespace.

The checker does not flag names at global scope if they are declared in `extern "C"` blocks (C code included within C++ code). However, if you use the option `Ignore link errors (-no-extern-c)`, these names are also flagged.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-3-2

The identifier main shall not be used for a function other than the global function main

Description

Rule Definition

The identifier main shall not be used for a function other than the global function main.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-3-3

There shall be no unnamed namespaces in header files

Description

Rule Definition

There shall be no unnamed namespaces in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-3-4

using-directives shall not be used

Description

Rule Definition

using-directives shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-3-5

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier

Description

Rule Definition

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-3-6

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files

Description

Rule Definition

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-4-2

Assembler instructions shall only be introduced using the asm declaration

Description

Rule Definition

Assembler instructions shall only be introduced using the asm declaration.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-4-3

Assembly language shall be encapsulated and isolated

Description

Rule Definition

Assembly language shall be encapsulated and isolated.

Polyspace Implementation

The checker flags `asm` statements unless they are encapsulated in a function call.

For instance, the noncompliant `asm` statement below is in regular C code while the compliant `asm` statement is encapsulated in a call to the function `Delay`.

```
void Delay ( void )
  {
    asm( "NOP");//Compliant
  }
void fn (void)
  {
    DoSomething();
    Delay();// Assembler is encapsulated
    DoSomething();
    asm("NOP"); //Noncompliant
    DoSomething();
  }
```

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-5-1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function

Description

Rule Definition

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-5-2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist

Description

Rule Definition

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Rationale

If an object continues to point to another object *after* the latter object ceases to exist, dereferencing the first object leads to undefined behavior.

Polyspace Implementation

The checker flags situations where the address of a local variable is assigned to a pointer defined at global scope.

The checker does not raise violations of this rule if:

- A function returns the address of a local variable. `MISRA C++:2008 Rule 7-5-1` covers this situation.

- The address of a variable defined at block scope is assigned to a pointer that is defined with greater scope, but not global scope.

  For instance:

  ```
  void foobar ( void )
   {
     char * ptr;
     {
       char var;
       ptr = &var;
     }
   }
  ```

  Only if the pointer is defined at global scope is a rule violation raised. For instance, the rule checker flags the assignment here:

  ```
  char * ptr;
  void foobar ( void )
    {
        char var;
        ptr = &var;
    }
  ```

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Address of Local Variable Assigned to Global Pointer

```
char * ptr;

void foo (void) {
    char varInFoo;
    ptr = &varInFoo; //Noncompliant
}

void bar (void) {
    char varInBar = *ptr;
}

void main() {
    foo();
    bar();
}
```

The assignment `ptr = &varInFoo` is noncompliant because the global pointer `ptr` might be dereferenced outside the function `foo`, where the variable `varInFoo` is no longer in scope. For instance, in this example, `ptr` is dereferenced in the function `bar`, which is called after `foo` completes execution.

Check Information

Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-5-3

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference

Description

Rule Definition

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Declarations
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 7-5-4

Functions should not call themselves, either directly or indirectly

Description

Rule Definition

Functions should not call themselves, either directly or indirectly.

Polyspace Implementation

The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.

You can calculate the total number of recursion cycles using the code complexity metric `Number of Recursions`. Note that unlike the checker, the metric also considers implicit calls, for instance, to compiler-generated constructors during object creation.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarations
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 8-0-1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively

Description

Rule Definition

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Declarators
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 8-3-1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments

Description

Rule Definition

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Declarators
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 8-4-1

Functions shall not be defined using the ellipsis notation

Description

Rule Definition

Functions shall not be defined using the ellipsis notation.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarators
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 8-4-2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration

Description

Rule Definition

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Polyspace Implementation

The checker detects mismatch in parameter names between:

- A function declaration and the corresponding definition.
- Two declarations of a function, provided they occur in the same file.

  If the declarations occur in different files, the checker does not raise a violation for mismatch in parameter names. Redeclarations in different files are forbidden by `MISRA C++:2008 Rule 3-2-3`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Declarators
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 8-4-3

All exit paths from a function with non- void return type shall have an explicit return statement with an expression

Description

Rule Definition

All exit paths from a function with non- void return type shall have an explicit return statement with an expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarators
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 8-4-4

A function identifier shall either be used to call the function or it shall be preceded by &

Description

Rule Definition

A function identifier shall either be used to call the function or it shall be preceded by &.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarators
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 8-5-1

All variables shall have a defined value before they are used

Description

Rule Definition

All variables shall have a defined value before they are used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Declarators
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 8-5-2

Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures

Description

Rule Definition

Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.

Rationale

The use of nested braces in initializer lists to match the structures of nested objects in arrays, unions, and structs encourages you to consider the order of initialization of complex data types and makes your code more readable. For example, the use of nested braces in the initialization of `ex1` makes it easier to see how the nested arrays `arr1` and `arr2` in struct `ex1` are initialized.

```
struct Example
{
    int num;
    int arr1[2];
    int arr2[3];
};

//....
struct Example ex1 {1, {2, 3}, {4, 5, 6}}; //Compliant
```

The rule does not require the use of nested braces if you zero initialize an array, a union, or a struct with nested structures are the top-level, for instance:

```
struct Example ex1 {}; //Compliant
```

Polyspace Implementation

If you non-zero initialize an array, union, or struct that contains nested structures and you do not use nested braces to reflect the nested structure, Polyspace flags the first element of the first nested structure in the initializer list. For instance, in this code snippet, Polyspace flags the number 2 because it corresponds to the first element of nested structure `arr1` inside struct `ex1`.

```
struct Example
{
    int num;
    int arr1[2];
    int arr2[3];
};

//....
struct Example ex1 {1, 2, 3, 4, 5, 6}; // Non-compliant
```

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Missing Nested Braces in Initializer of Two-Dimensional Arrays

```
char arr1[2][3] {'a', 'b', 'c', 'd', 'e', 'f'}; //Non-compliant
char arr2[2][3] {{'a', 'b', 'c'}, {'d', 'e', 'f'}}; //Compliant
char arr_top_level[2][3] { }; //Compliant
char arr_sub_level[2][3] { {}, {'d', 'e', 'f'}}; //Non-compliant
```

In this example, two-dimensional array `arr1` is non-compliant because the initializer list does not reflect the nested structure of this array (two arrays of three elements each). The initialization of `arr2` uses nested braces to reflect the nested structure of the array and is compliant. Similarly, the initialization of `arr_top_level` is compliant because it zero initializes the array at the top level. Note that the initialization of `arr_sub_level` is non-compliant because zero-initializes only the first sub-array while explicitly initializing all the elements of the other sub-array.

Check Information

Group: Declarators
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 8-5-3

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized

Description

Rule Definition

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Declarators
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 9-3-1

const member functions shall not return non-const pointers or references to class-data

Description

Rule Definition

const member functions shall not return non-const pointers or references to class-data.

Polyspace Implementation

The checker flags a rule violation only if a `const` member function returns a non-`const` pointer or reference to a nonstatic data member. The rule does not apply to static data members.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 9-3-2

Member functions shall not return non-const handles to class-data

Description

Rule Definition

Member functions shall not return non-const handles to class-data.

Polyspace Implementation

The checker flags a rule violation only if a member function returns a non-`const` pointer or reference to a nonstatic data member. The rule does not apply to static data members.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 9-3-3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const

Description

Rule Definition

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Rationale

`const` member functions cannot modify the data members of the class. `static` member function cannot modify the nonstatic data members of the class. If a member function does not need to modify the nonstatic data members of the class, limit their access to data by declaring the member functions as `const` or `static`. Such declaration clearly expresses and enforces the design intent. That is, if you inadvertently attempt to modify a data member through a `const` member function, the compiler catches the error. Without the `const` declaration, this kind of inadvertent error might lead to bugs that are difficult to find or debug.

Polyspace Implementation

The checker performs these checks in this order:

1   The checker first checks if a class member function accesses a data member of the class. Functions that do not access data members can be declared static.

2   The checker then checks functions that access data members to determine if the function modifies any of the data members. Functions that do not modify data members can be declared const.

A violation on a const member function means that the function does not even access a data member of the class and can be declared static.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Explicitly Restrict Access for Member Functions that Do Not Modify Data Members

```
#include<cstdint>
void Connector(void);
class A
{
public:
    int16_t foo ( ) // Noncompliant
    {
        return m_i;
    }
```

```
    int16_t foo2 ( ) // Noncompliant
    {
        Connector();// Might have side-effect
        return m_i;
    }
    int16_t foo3 ( ) // Noncompliant
    {
        return m_s;
    }
    int16_t inc_m ( ) // Compliant
    {
        return ++m_i;
    }
    int16_t& getref()//Noncompliant
    {
        return m_i_ref;
    }
private:
    int16_t m_i;
    static int16_t m_s;
    int16_t& m_i_ref;
};
```

In this example, Polyspace flags the functions `foo`, `foo2`, `foo3`, and `getref` as noncompliant.

- The functions `foo` and `foo3` do not modify any nonstatic data members. Because their data access is not explicitly restricted by declaring them as `const`, Polyspace flags these functions. To fix these defects, declare `foo` and `foo3` as `const`.

- The function `foo2` does not explicitly modify any of the data members. Because it is not declared as `const`, Polyspace flags the function. `foo2` calls the global function `Connector`, which might have side effects. Do not declare `foo2` as a `const` function. In C++11 or later, `const` member functions are expected to be thread-safe, but `foo2` might not be thread-safe because of the side effects of `Connector`. To avoid data races, keep `foo2` as a nonconst function. Justify the defect by using review information or code comments.

- The function `getref` does not modify any data members. Because it is not declared as `const`, Polyspace flags it. Declaring `getref` as `const` resolves this defect, but that is not enough to restrict write access of `getref` because it returns a nonconst reference to `m_i_ref`. To restrict `getref` from modifying `m_i_ref`, the return type of `getref` must also be `const`.

Check Information
Group: Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2018a

MISRA C++:2008 Rule 9-5-1

Unions shall not be used

Description

Rule Definition

Unions shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 9-6-2

Bit-fields shall be either bool type or an explicitly unsigned or signed integral type

Description

Rule Definition

Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 9-6-3

Bit-fields shall not have enum type

Description

Rule Definition

Bit-fields shall not have enum type.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 9-6-4

Named bit-fields with signed integer type shall have a length of more than one bit

Description

Rule Definition

Named bit-fields with signed integer type shall have a length of more than one bit.

Rationale

Variables with signed integer bit-field types of length one might have values that do not meet developer expectations. For instance, signed integer types of fixed width such as `std16_t` (from `cstdint`) have a two's complement representation. In this representation, a single bit is just the sign bit and the value might be 0 or -1.

Polyspace Implementation

The checker flags declarations of named variables having signed integer bit field types of length equal to one.

Bit field types of length zero are not flagged.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Compliant and Noncompliant Bit-Field Types

```
#include <cstdint>

typedef struct
{
   std::uint16_t IOFlag :1;     //Compliant - unsigned type
   std::int16_t InterruptFlag :1; //Noncompliant
   std::int16_t Register1Flag :2; //Compliant - Length more than one bit
   std::int16_t : 1; //Compliant - Unnamed
   std::int16_t : 0; //Compliant - Unnamed
   std::uint16_t SetupFlag :1; //Compliant - unsigned type
} InterruptConfigbits_t;
```

In this example, only the second bit-field declaration is noncompliant. A named variable is declared with a signed type of length one bit.

Check Information
Group: Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 10-1-1

Classes should not be derived from virtual bases

Description

Rule Definition

Classes should not be derived from virtual bases.

Rationale

The use of virtual bases can lead to many confusing behaviors.

For instance, in an inheritance hierarchy involving a virtual base, the most derived class calls the constructor of the virtual base. Intermediate calls to the virtual base constructor are ignored.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of Virtual Bases

```
class Base {};
class Intermediate: public virtual Base {}; //Noncompliant
class Final: public Intermediate {};
```

In this example, the rule checker raises a violation when the `Intermediate` class is derived from the class `Base` with the `virtual` keyword.

The following behavior can be a potential source of confusion. When you create an object of type `Final`, the constructor of `Final` directly calls the constructor of `Base`. Any call to the `Base` constructor from the `Intermediate` constructor are ignored. You might see unexpected results if you do not take into account this behavior.

Check Information
Group: Derived Classes
Category: Advisory

See Also
MISRA C++:2008 Rule 10-1-2

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 10-1-2

A base class shall only be declared virtual if it is used in a diamond hierarchy

Description

Rule Definition

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Rationale

This rule is less restrictive than `MISRA C++:2008 Rule 10-1-1`. Rule 10-1-1 forbids the use of a virtual base anywhere in your code because a virtual base can lead to potentially confusing behavior.

Rule 10-1-2 allows the use of virtual bases in the one situation where they are useful, that is, as a common base class in diamond hierarchies.

For instance, the following diamond hierarchy violates rule 10-1-1 but not rule 10-1-2.

```
class Base {};
class Intermediate1: public virtual Base {};
class Intermediate2: public virtual Base {};
class Final: public Intermediate1, public Intermediate2 {};
```

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Derived Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 10-1-3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy

Description

Rule Definition

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Rationale

The checker flags situations where the same class is inherited as a virtual base class and a non-virtual base class in the same derived class. These situations defeat the purpose of virtual inheritance and causes multiple copies of the base class sub-object in the derived class object.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Base Class Both Virtual and Non-Virtual in Same Hierarchy

```
class Base {};
class Intermediate1: virtual public Base {};
class Intermediate2: virtual public Base {};
class Intermediate3: public Base {};
class Final: public Intermediate1, Intermediate2, Intermediate3 {}; //Noncompliant
```

In this example, the class `Base` is inherited in `Final` both as a virtual and non-virtual base class. The `Final` object contains at least two copies of a `Base` sub-object.

Check Information
Group: Derived Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 10-2-1

All accessible entity names within a multiple inheritance hierarchy should be unique

Description

Rule Definition

All accessible entity names within a multiple inheritance hierarchy should be unique.

Polyspace Implementation

The checker flags data members from different classes with conflicting names if the same class derives from these classes. For instance:

```
class B1
  {
    public:
      int count;
      void foo ( );
  };
class B2
  {
    public:
      int count;
      void foo ( );
  };

class D : public B1, public B2
  {
    public:
      void Bar ( )
      {
        ++B1::count;
        B1::foo ( );
      }
  };
```

If the data member access in the derived class is ambiguous, the analysis reports this issue as a compilation error and not a coding rule violation. For instance, a compilation error occurs in the preceding example if the class D is rewritten as:

```
class D : public B1, public B2
  {
    public:
      void Bar ( )
      {
        ++count;        // Is that B1::count or B2::count?
        foo ( );        // Is that B1::foo() or B2::foo()?
      }
  };
```

The checker does not check for conflicts between entities of different kinds, for instance, member functions against data members.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Derived Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 10-3-1

There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy

Description

Rule Definition

There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.

Rationale

The checker flags virtual member functions that have multiple definitions on the same path in an inheritance hierarchy. If a function is defined multiple times, it can be ambiguous which implementation is used in a given function call.

Polyspace Implementation

The checker also raises a violation if a base class member function is redefined in the derived class without the `virtual` keyword.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Virtual Function Redefined in Derived Class

```
class Base {
    public:
      virtual void foo() {
      }
};

class Intermediate1: public virtual Base {
    public:
      virtual void foo() { //Noncompliant
      }
};

class Intermediate2: public virtual Base {
    public:
       void bar() {
          foo();  // Calls Base::foo()
       }
};

class Final: public Intermediate1, public Intermediate2 {
};
```

```
void main() {
    Intermediate2 intermediate2Obj;
    intermediate2Obj.bar(); // Calls Base::foo()
    Final finalObj;
    finalObj.bar(); //Calls Intermediate1::foo()
                    //but you might expect Base::foo()
}
```

In this example, the `virtual` function `foo` is defined in the base class `Base` and also in the derived class `Intermediate1`.

A potential source of confusion can be the following. The class `Final` derives from `Intermediate1` and also derives from `Base` through another path using `Intermediate2`.

- When an `Intermediate2` object calls the function `bar` that calls the function `foo`, the implementation of `foo` in `Base` is called. An `Intermediate2` object does not know of the implementation in `Intermediate1`.

- However, when a `Final` object calls the same function `bar` that calls the function `foo`, the implementation of `foo` in `Intermediate1` is called because of dominance of the more derived class.

You might see unexpected results if you do not take this behavior into account.

To prevent this issue, declare a function as pure virtual in the base class. For instance, you can declare the class `Base` as follows:

```
class Base {
    public:
       virtual void foo()=0;
};

void Base::foo() {
      //You can still define Base::foo()
}
```

Check Information
Group: Derived Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 10-3-2

Each overriding virtual function shall be declared with the virtual keyword

Description

Rule Definition

Each overriding virtual function shall be declared with the virtual keyword.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Derived Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 10-3-3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual

Description

Rule Definition

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Derived Classes
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 11-0-1

Member data in non- POD class types shall be private

Description

Rule Definition

Member data in non- POD class types shall be private.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Member Access Control
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 12-1-1

An object's dynamic type shall not be used from the body of its constructor or destructor

Description

Rule Definition

An object's dynamic type shall not be used from the body of its constructor or destructor.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Special Member Functions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 12-1-2

All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes

Description

Rule Definition

All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Special Member Functions
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 12-1-3

All constructors that are callable with a single argument of fundamental type shall be declared explicit

Description

Rule Definition

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Special Member Functions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 12-8-1

A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member

Description

Rule Definition

A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Special Member Functions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 12-8-2

The copy assignment operator shall be declared protected or private in an abstract class

Description

Rule Definition

The copy assignment operator shall be declared protected or private in an abstract class.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Special Member Functions
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 14-5-1

A non-member generic function shall only be declared in a namespace that is not an associated namespace

Description

Rule Definition

A non-member generic function shall only be declared in a namespace that is not an associated namespace.

Rationale

This rule forbids placing generic functions in the same namespace as class (struct) type, enum type, or union type declarations. If the class, enum or union types are used as template parameters, the presence of generic functions in the same namespace can cause unexpected call resolutions. Place generic functions only in namespaces that cannot be associated with a class, enum or union type.

Consider the namespace NS that combines a class B and a generic form of `operator==`:

```
namespace NS {
    class B {};
    template <typename T> bool operator==(T, std::int32_t);
}
```

If you use class B as a template parameter for another generic class, such as this template class A:

```
template <typename T> class A {
    public:
        bool operator==(std::int64_t);
}

template class A<NS::B>;
```

the entire namespace NS is used for overload resolution when operators of class A are called. For instance, if you call `operator==` with an `int32_t` argument, the generic `operator==` in the namespace NS with an `int32_t` parameter is used instead of the `operator==` in the original template class A with an `int64_t` parameter. You or another developer or code reviewer might expect the operator call to resolve to the `operator==` in the original template class A.

Polyspace Implementation

For each generic function, the rule checker determines if the containing namespace also contains declarations of class types, enum types, or union types. If such a declaration is found, the checker flags a rule violation on the operator itself.

The checker also flags generic functions defined in the global namespace if the global namespace also has class, enum or union declarations.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Generic Operator in Same Namespace as Class Type

```
#include <cstdint>

template <typename T> class Pair {
    std::int32_t item1;
    std::int32_t item2;
    public:
      bool operator==(std::int64_t ItemToCompare);
      bool areItemsEqual(std::int32_t itemValue) {
          return (*this == itemValue);
      }
};

namespace Operations {
    class Data {};
    template <typename T> bool operator==(T, std::int32_t); //Noncompliant
}

namespace Checks {
    bool checkConsistency();
    template <typename T> bool operator==(T, std::int32_t); //Compliant
}

template class Pair<Operations::Data>;
```

In this example, the namespace `Operations` violates the rule because it contains the class type `Data` alongside the generic `operator==`. The namespace `Checks` does not violate the rule because the only other declaration in the namespace, besides the generic `operator==`, is a function declaration.

In the method `areItemsEqual` in `template class Pair<Operations::Data>`, the == operation invokes the generic `operator==` method in the `Operations` namespace. The invocation resolves to this `operator==` method based on the argument data type (`std_int32_t`). This method is a better match compared to the `operator==` method in the original template class `Pair`.

Check Information
Group: Templates
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2020b

MISRA C++:2008 Rule 14-5-2

A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter

Description

Rule Definition

A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Templates
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 14-5-3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter

Description

Rule Definition

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Templates
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 14-6-1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->

Description

Rule Definition

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Templates
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 14-6-2

The function chosen by overload resolution shall resolve to a function declared previously in the translation unit

Description

Rule Definition

The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.

Rationale

In general, you cannot call a function before it is declared, so you expect a function call to resolve to a previously declared function. However, in case of overload resolution of a function call inside a template, this expectation might not be satisfied. The resolution of this overload occurs at the point of template instantiation, not at the point of template definition. So, the call might resolve to a function that is declared *after* the template definition and lead to unexpected results. See examples below.

To satisfy the expectation that a function call *always* resolves to a previously declared function, declare the overloads of a function prior to calling it. Alternatively, use the scope resolution operator `::` or parenthesis to explicitly call a specific previously declared function and bypass the overload resolution mechanism.

Polyspace Implementation

The checker flags a call to a function or operator in a function template definition if the function or operator is declared *after* the template definition.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Function Call Resolves to Function Declared Later

```
void show (int);

namespace helpers {
     struct params {
        operator int () const;
     };
}

template <typename T> void displayParams(T const & arg) {
     show(arg);    //Non-compliant
     :show(arg);  //Compliant
     (show)(arg);  //Compliant
}

namespace helpers {
```

```
        void show (params const &);
}

void main() {
        helpers::params aParam;
        displayParams(aParam);
}
```

In this example, the call `show(arg)` in the template `displayParams` resolves to `helpers::show()`, but a developer or code reviewer might not expect this call resolution, since `helpers::show()` is declared later. Polyspace flags this call.

The calls `::show(arg)` and `(show)(arg)` explicitly indicate the previously declared function `show()` declared in the global namespace. Polyspace does not flag these calls.

Check Information
Group: Templates
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 14-7-3

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template

Description

Rule Definition

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Templates
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 14-8-1

Overloaded function templates shall not be explicitly specialized

Description

Rule Definition

Overloaded function templates shall not be explicitly specialized.

Polyspace Implementation

The checker first checks within file scope to find overloads. The checker later looks for call to a specialized template function later. As a result, the checker flags all specializations of overloaded templates even if overloading occurs after the call.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Templates
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 14-8-2

The viable function set for a function call should either contain no function specializations, or only contain function specializations

Description

Rule Definition

The viable function set for a function call should either contain no function specializations, or only contain function specializations.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Templates
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-0-2

An exception object should not have pointer type

Description

Rule Definition

An exception object should not have pointer type.

Polyspace Implementation

The checker raises a violation if a `throw` statement throws an exception of pointer type.

The checker does not raise a violation if a NULL pointer is thrown as exception. Throwing a NULL pointer is forbidden by `MISRA C++:2008 Rule 15-1-2`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Exception Handling
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-0-3

Control shall not be transferred into a try or catch block using a goto or a switch statement

Description

Rule Definition

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-1-1

The assignment-expression of a throw statement shall not itself cause an exception to be thrown

Description

Rule Definition

The assignment-expression of a throw statement shall not itself cause an exception to be thrown.

Rationale

In C++, you can use a `throw` statement to raise exceptions explicitly. The compiler executes such a `throw` statement in two steps:

- First, it creates the argument for the `throw` statement. The compiler might call a constructor or evaluate an assignment expression to create the argument object.
- Then, it raises the created object as an exception. The compiler tries to match the exception object to a compatible handler.

If an unexpected exception is raised when the compiler is creating the expected exception in a `throw` statement, the unexpected exception is raised instead of the expected one. Consider this code where a `throw` statement raises an explicit exception of class `myException`.

```
class myException{
    myException(){
        msg = new char[10];
        //...
    }
    //...
};

foo(){
    try{
        //..
        throw myException();
    }
    catch(myException& e){
        //...
    }
}
```

During construction of the temporary `myException` object, the `new` operator can raise a `bad_alloc` exception. In such a case, the `throw` statement raises a `bad_alloc` exception instead of `myException`. Because `myException` was the expected exception, the catch block is incompatible with `bad_alloc`. The `bad_alloc` exception becomes an unhandled exception. It might cause the program to abort abnormally without unwinding the stack, leading to resource leak and security vulnerabilities.

Unexpected exceptions arising from the argument of a `throw` statement can cause resource leaks and security vulnerabilities. To prevent such unwanted outcome, avoid using expressions that might raise exceptions as argument in a `throw` statement.

Polyspace Implementation

Polyspace flags the expressions in `throw` statements that can raise an exception. Expressions that can raise exceptions can include:

- Functions that are specified as `noexcept(false)`
- Functions that contain one or more explicit `throw` statements
- Constructors that perform memory allocation operations
- Expressions that involve dynamic casting

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Avoid Expressions That Can Raise Exceptions in `throw` Statements

This example shows how Polyspace flags the expressions in `throw` statements that can raise unexpected exceptions.

```
int f_throw() noexcept(false);

class WithDynamicAlloc {
public:
    WithDynamicAlloc(int n) {
        m_data = new int[n];
    }
    ~WithDynamicAlloc() {
        delete[] m_data;
    }
private:
    int* m_data;
};

class MightThrow {
public:
    MightThrow(bool b) {
        if (b) {
            throw 42;
        }
    }
};

class Base {
    virtual void bar() =0;
};
class Derived: public Base {
    void bar();
};
class UsingDerived {
public:
    UsingDerived(const Base& b) {
        m_d =
        dynamic_cast<const Derived&>(b);
```

```
        }
    private:
        Derived m_d;
    };
    class CopyThrows {
    public:
        CopyThrows() noexcept(true);
        CopyThrows(const CopyThrows& other) noexcept(false);
    };
    int foo(){
        try{
            //...
            throw WithDynamicAlloc(10); //Noncompliant
            //...
            throw MightThrow(false);//Noncompliant
            throw MightThrow(true);//Noncompliant
            //...
            Derived d;
            throw  UsingDerived(d);// Noncompliant
            //...
            throw f_throw(); //Noncompliant
            CopyThrows except;
            throw except;//Noncompliant
        }
        catch(WithDynamicAlloc& e){
            //...
        }
        catch(MightThrow& e){
            //...
        }
        catch(UsingDerived& e){
            //...
        }
    }
```

- When constructing a `WithDyamicAlloc` object by calling the constructor `WithDynamicAlloc(10)`, exceptions can be raised during dynamic memory allocation. Because the expression `WithDynamicAlloc(10)` can raise an exception, Polyspace flags the `throw` statement `throw WithDynamicAlloc(10);`

- When constructing a `UsingDerived` object by calling the constructor `UsingDervide()`, exceptions can be raised during the dynamic casting operation. Because the expression `UsingDerived(d)` can raise exceptions, Polyspace flags the statement `throw UsingDerived(d)`.

- In the function `MightThrow()`, exceptions can be raised depending on the input to the function. Because Polyspace analyzes functions statically, it assumes that the function `MightThrow()` can raise exceptions. Polyspace flags the statements `throw MightThrow(false)` and `throw MightThrow(true)`.

- In the statement `throw except`, the object `except` is copied by implicitly calling the copy constructor of the class `CopyThrows`. Because the copy constructor is specified as `noexcept(false)`, Polyspace assumes that the copy operation might raise exceptions. Polyspace flags the statement `throw except`

- Because the function `f_throw()` is specified as `noexcept(false)`, Polyspace assumes that it can raise exceptions. Polyspace flags the statement `throw f_throw()`.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2020b

MISRA C++:2008 Rule 15-1-2

NULL shall not be thrown explicitly

Description

Rule Definition

NULL shall not be thrown explicitly.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-1-3

An empty throw (throw;) shall only be used in the compound- statement of a catch handler

Description

Rule Definition

An empty throw (throw;) shall only be used in the compound- statement of a catch handler.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-3-1

Exceptions shall be raised only after start-up and before termination of the program

Description

Rule Definition

Exceptions shall be raised only after start-up and before termination of the program.

Rationale

In c++, the process of exception handling runs during execution of `main()`, where exceptions arising in different scopes are handled by exception handlers in the same or adjacent scopes. Before starting the execution of `main()`, the compiler is in startup phase, and after finishing the execution of `main()`, the compiler is in termination phase. During these two phases, the compiler performs a set of predefined operations but does not execute any code.

If an exception is raised during either the startup phase or the termination phase, you cannot write an exception handler that the compiler can execute in those phases. For instance, you might implement `main()` as a `function-try-catch` block to handle exceptions. The `catch` blocks in `main()` can handle only the exceptions raised in `main()`. None of the `catch` blocks can handle exceptions raised during startup or termination phase. When such exceptions are raised, the compiler might abnormally terminate the code execution without unwinding the stack. Consider this code where the construction and destruction of the static object `obj` might cause an exception.

```
class A{
    A(){throw(0);}
    ~A(){throw(0)}
};

static A obj;

main(){
    //...
}
```

The static object `obj` is constructed by calling `A()` before `main()` starts, and it is destroyed by calling `~A()` after `main()` ends. When `A()` or `~A()` raises an exception, no exception handler can be matched with them. Based on the implementation, such an exception can result in program termination without stack unwinding, leading to memory leak and security vulnerabilities.

Avoid operations that might raise an exception in the parts of your code that might be executed before startup or after termination of the program. For instance, avoid operations that might raise exceptions in the constructor and destructor of static or global objects.

Polyspace Implementation

Polyspace flags global or static variable declaration that uses a callable entity that might raise an exception. For instance:

- Functions: When you call an initializer function or constructor directly to initialize a global or static variable, Polyspace checks whether the function raises an exception and flags the variable

declaration if the function might raise an exception. Polyspace deduces whether a function might raise an exception regardless of its exception specification. For instance, If a `noexcept` constructor raises an exception, Polyspace flags it. If the initializer or constructor calls another function, Polyspace assumes the called function might raise an exception only if it is specified as `noexcept(<false>)`. Some standard library functions, such as the constructor of `std::string`, uses pointers to functions to perform memory allocation, which might raise exceptions. Polyspace does not flag the variable declaration when these functions are used.

- External function: When you call external functions to initialize a global or static variable, Polyspace flags the declaration if the external function is specified as `noexcept(<false>)`.

- Virtual function: When you call a virtual function to initialize a global or static variable, Polyspace flags it if the virtual function is specified as `noexcept(<false>)` in any derived class. For instance, if you use a virtual initializer function that is declared as `noexcept(<true>)` in the base class, and `noexcept(<false>)` in a subsequent derived class, Polyspace flags it.

- Pointers to function: When you use a pointer to a function to initialize a global or static variable, Polyspace assumes that pointer to function do not raise exceptions.

Polyspace ignores:

- Exceptions raised in destructors
- Exceptions raised in `atext()` operations

Polyspace also ignores the dynamic context when checking for exceptions. For instance, you might initialize a global or static variable by using function that raises exceptions only in certain dynamic context. Polyspace flags such a declaration even if the exception might never be raised. You can justify such a violation using comments in Polyspace.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Avoid Exceptions Before `main()` Starts

This example shows how Polyspace flags construction or initialization of a global or static variable that might raise an exception. Consider this code where static and global objects are initialized by using various callable entities.

```
#include <stdexcept>
#include <string>
class C
{
public:
    C ( ){throw ( 0 );}
    ~C ( ){throw ( 0 );}
};
int LibraryFunc();
int LibraryFunc_noexcept_false() noexcept(false);
int LibraryFunc_noexcept_true() noexcept(true);
int  g() noexcept {
    throw std::runtime_error("dead code");
    return 0;
}
```

```
int f() noexcept {
    return g();
}
int init(int a) {
    if (a>10) {
        throw std::runtime_error("invalid case");
    }
    return a;
}
void* alloc(size_t s) noexcept {
    return new int[s];
}
int a = LibraryFunc() +
LibraryFunc_noexcept_true();            // Compliant
int c =
LibraryFunc_noexcept_false() +          // Noncompliant
LibraryFunc_noexcept_true();
static C static_c;                       //Noncompliant
static C static_d;                       //Compliant
C &get_static_c(){
    return static_c;
}
C global_c;                              //Noncompliant
int a3 = f();                            //Compliant
int b3 = g();                            //Noncompliant
int a4 = init(5);                        //Noncompliant
int b5 = init(20);                       //Noncompliant
int* arr = (int*)alloc(5);               //Noncompliant

int main(){
    //...
}
```

- The global pointer `arr` is initialized by using the function `alloc()`. Because `alloc()` uses `new` to allocate memory, it can raise an exception when initializing `arr` during the startup of the program. Polyspace flags the declaration of `arr` and highlights the use of `new` in the function `alloc()`.

- The integer variable `b3` is initialized by calling the function `g()`, which is specified as `noexcept`. Polyspace deduces that the correct exception specification of `g()` is `noexcept(false)` because it contains a `throw()` statement. Initializing the global variable `b3` by using `g()` might raise an exception when initializing `arr` during the startup of the program. Polyspace flags the declaration of `b3` and highlights the `throw` statement in `g()`. The declaration of `a3` by calling `f()` is not flagged. Because `f()` is a `noexcept` function that does not throw, and calls another `noexcept` function, Polyspace deduces that `f()` does not raise an exception.

- The global variables `a4` and `b5` are initialized by calling the function `init()`. The function `init()` might raise an exception in certain cases, depending on the context. Because Polyspace deduces the exception specification of a function statically, it assumes that `init()` might raise an exception regardless of context. Consequently, Polyspace flags the declarations of both `a4` and `b5`, even though `init()` raises an exception only when initializing `b5`.

- The global variable `global_int` is initialized by calling two external functions. The external function `LibraryFunc_noexcept_false()` is specified as `noexcept(false)` and Polyspace assumes that this external function might raise an exception. Polyspace flags the declaration of `global_int`. Polyspace does not flag the declaration of `a` because it is initialized by calling external functions that are not specified as `noexcept(false)`.

- The static variable `static_c` and the nonstatic global variable `global_c`is declared and initialized by using the constructor of the class C, which might raise an exception. Polyspace flags the declarations of these variables and highlights the `throw()` statement in the constructor of class C. Polyspace does not flag the declaration of the unused static variable `static_d`, even though its constructor might raise an exception. Because it is unused, `static_d` is not initialized and its constructor is not called. Its declaration does not raise any exception.

Check Information

Group: Templates
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2020b

MISRA C++:2008 Rule 15-3-2

There should be at least one exception handler to catch all otherwise unhandled exceptions

Description

Rule Definition

There should be at least one exception handler to catch all otherwise unhandled exceptions.

Polyspace Implementation

The checker shows a violation if there is no `try`/`catch` in the `main` function or the `catch` does not handle all exceptions (with ellipsis `...`). The rule is not checked if a `main` function does not exist.

The checker does not determine if an exception of an unhandled type actually propagates to `main`.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Exception Handling
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-3-3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases

Description

Rule Definition

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-3-4

Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point

Description

Rule Definition

Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

Rationale

In C++, when an operation raises an exception, the compiler tries to match the exception with a compatible exception handler in the current and adjacent scopes. If no compatible exception handler for a raised exception exists, the compiler invokes the function `std::terminate()` implicitly. The function `std::terminate()` terminates the program execution in an implementation-defined manner. That is, the exact process of program termination depends on the particular set of software and hardware that you use. For instance, `std::terminate()` might invoke `std::abort()` to abnormally abort the execution without unwinding the stack. If the stack is not unwound before program termination, then the destructors of the variables in the stack are not invoked, leading to resource leak and security vulnerabilities.

Consider this code where multiple exceptions are raised in the try block of code.

```
class General{/*...  */};
class Specific : public General{/*...*/};
class Different{}
void foo() noexcept
{
    try{
        //...
        throw(General e);
        //..
        throw( Specific e);
        // ...
        throw(Different e);
    }
    catch (General& b){

    }
}
```

The catch block of code accepts references to the base class `General`. This catch block is compatible with exceptions of the base class `General` and the derived class `Specific`. The exception of class `Different` does not have a compatible handler. This unhandled exception violates this rule and might result in resource leaks and security vulnerabilities.

Because unhandled exceptions can lead to resource leak and security vulnerabilities, match the explicitly raised exceptions in your code with a compatible handler.

Polyspace Implementation

- Polyspace flags a `throw` statement in a function if a compatible catch statement is absent in the call path of the function. If the function is not specified as `noexcept`, Polyspace ignores it if its call path lacks an entry point like `main()`.

- Polyspace flags a `throw` statement that uses a `catch(…)` statement to handle the raised exceptions.

- Polyspace does not flag rethrow statements, that is, `throw` statements within catch blocks.

- You might have compatible catch blocks for the `throw` statements in your function in a nested try-catch block Polyspace ignores nested try-catch blocks. Justify `throw` statements that have compatible catch blocks in a nested structure by using comments. Alternatively, use a single level of try-catch in your functions.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Match `throw` Statements with Compatible Catch Blocks

This example shows how Polyspace flags operations that raise exceptions without any compatible handler. Consider this code.

```
#include <stdexcept>

class MyException : public std::runtime_error {
public:
    MyException() : std::runtime_error("MyException") {}
};

void ThrowingFunc() {
    throw MyException(); //Noncompliant
}

void CompliantCaller() {
    try {
        ThrowingFunc();
    } catch (std::exception& e) {
        /* ... */
    }
}

void NoncompliantCaller() {
    ThrowingFunc();
}

int main(void) {
    CompliantCaller();
    NoncompliantCaller();
}

void GenericHandler() {
    try {
        throw MyException(); //Noncompliant
```

```
    } catch (...) {
        /* ... */
    }
}

void TrueNoexcept() noexcept {
    try {
        throw MyException();//Compliant
    } catch (std::exception& e) {
        /* ... */
    }
}

void NotNoexcept() noexcept {
    try {
        throw MyException(); //Noncompliant
    } catch (std::logic_error& e) {
        /* ... */
    }
}
```

- The function `ThrowingFunc()` raises an exception. This function has multiple call paths:

  - `main()->CompliantCaller()->ThrowingFunc()`: In this call path, the function `CompliantCaller()` has a catch block that is compatible with the exception raised by `ThrowingFunc()`. This call path is compliant with the rule.

  - `main()->NoncompliantCaller()->ThrowingFunc()`: In this call path, there are no compatible handlers for the exception raised by `ThrowingFunc()`. Polyspace flags the `throw` statement in `ThrowingFunc()` and highlights the call path in the code.

  The function `main()` is the entry point for both of these call paths. If `main()` is commented out, Polyspace ignores both of these call paths. If you want to analyze a call path that lacks an entry point, specify the top most calling function as `noexcept`.

- The function `GenericHandler()` raises an exception by using a `throw` statement and handles the raised exception by using a generic catch-all block. Because Polyspace considers such catch-all handler to be incompatible with exceptions that are raised by explicit `throw` statements, Polyspace flags the `throw` statement in `GenericHandler()`.

- The `noexcept` function `TrueNoexcept()` contains an explicit `throw`statement and a catch block of compatible type. Because this `throw` statement is matched with a compatible `catch` block, it is compliant with the rule.

- The `noexcept` function `NotNoexcept()` contains an explicit `throw` statement, but the catch block is not compatible with the raised exception. Because this `throw` statement is not matched with a compatible catch block, Polyspace flags the `throw` statement in `NotNoexcept()`.

Check Information

Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2020b

MISRA C++:2008 Rule 15-3-5

A class type exception shall always be caught by reference

Description

Rule Definition

A class type exception shall always be caught by reference.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-3-6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class

Description

Rule Definition

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-3-7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last

Description

Rule Definition

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-4-1

If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids

Description

This checker is deactivated in a default Polyspace as You Code *analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" (Polyspace Bug Finder Access).*

Rule Definition

If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-5-1

A class destructor shall not exit with an exception

Description

Rule Definition

A class destructor shall not exit with an exception.

Polyspace Implementation

The checker flags exceptions thrown in the body of the destructor. If the destructor calls another function, the checker does not detect if that function throws an exception.

The checker does not detect these situations:

- A `catch` statement does not catch exceptions of all types that are thrown.

  The checker considers the presence of a `catch` statement corresponding to a `try` block as indication that an exception is caught.
- `throw` statements inside `catch` blocks

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-5-2

Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s)

Description

Rule Definition

Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).

Polyspace Implementation

The checker flags situations where the data type of the exception thrown does not match the exception type listed in the function specification.

For instance:

```
void goo ( ) throw ( Exception )
  {
    throw 21; // Non-compliant - int is not listed
  }
```

The checker limits detection to `throw` statements that are in the body of the function. If the function calls another function, the checker does not detect if the called function throws an exception.

The checker does not detect `throw` statements inside `catch` blocks.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 15-5-3

The terminate() function shall not be called implicitly

Description

Rule Definition

The terminate() function shall not be called implicitly.

Polyspace Implementation

The checker flags these situations when the terminate() function can be called implicitly:

- An exception escapes uncaught. This also violates `MISRA C++:2008 Rule 15-3-2`. For instance:

  - Before an exception is caught, it escapes through another function that throws an uncaught exception. For instance, a catch statement or exception handler invokes a copy constructor that throws an uncaught exception.
  - A throw expression with no operand rethrows an uncaught exception.

- A class destructor throws an exception. This also violates `MISRA C++:2008 Rule 15-5-1`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Exception Handling
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2018a

MISRA C++:2008 Rule 16-0-1

#include directives in a file shall only be preceded by other preprocessor directives or comments

Description

Rule Definition

#include directives in a file shall only be preceded by other preprocessor directives or comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-0-2

Macros shall only be #define 'd or #undef 'd in the global namespace

Description

Rule Definition

Macros shall only be #define 'd or #undef 'd in the global namespace.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-0-3

#undef shall not be used

Description

Rule Definition

#undef shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-0-4

Function-like macros shall not be defined

Description

Rule Definition

Function-like macros shall not be defined.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-0-5

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives

Description

Rule Definition

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-0-6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##

Description

Rule Definition

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-0-7

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator

Description

Rule Definition

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-0-8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token

Description

Rule Definition

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-1-1

The defined preprocessor operator shall only be used in one of the two standard forms

Description

Rule Definition

The defined preprocessor operator shall only be used in one of the two standard forms.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-1-2

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related

Description

Rule Definition

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-2-1

The preprocessor shall only be used for file inclusion and include guards

Description

Rule Definition

The preprocessor shall only be used for file inclusion and include guards.

Polyspace Implementation

The checker flags `#ifdef` and `#define` statements in files that are not include files.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-2-2

C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers

Description

Rule Definition

C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.

Polyspace Implementation

The checker flags `#define` statements where the macros expand to something other than include guards, type qualifiers or storage class specifiers such as `static`, `inline`, `volatile`, `auto`, `register` and `const`.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-2-3

Include guards shall be provided

Description

Rule Definition

Include guards shall be provided.

Polyspace Implementation

The checker raises a violation if a header file does not contain an include guard.

For instance, this code uses an include guard for the `#define` and `#include` statements and does not violate the rule:

```
// Contents of a header file
#ifndef FILE_H

#define FILE_H
#include "libFile.h"

#endif
```

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-2-4

The ', ", /* or // characters shall not occur in a header file name

Description

Rule Definition

The ', ", / or // characters shall not occur in a header file name.*

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-2-5

The \ character should not occur in a header file name

Description

Rule Definition

The \ character should not occur in a header file name.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-2-6

The #include directive shall be followed by either a <filename> or "filename" sequence

Description

Rule Definition

The #include directive shall be followed by either a <filename> or "filename" sequence.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-3-1

There shall be at most one occurrence of the # or ## operators in a single macro definition

Description

Rule Definition

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-3-2

The # and ## operators should not be used

Description

Rule Definition

The # and ## operators should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Preprocessing Directives
Category: Advisory

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 16-6-1

All uses of the #pragma directive shall be documented

Description

Rule Definition

All uses of the #pragma directive shall be documented.

Polyspace Implementation

To check this rule, you must list the pragmas that are allowed in source files by using the option `Allowed pragmas (-allowed-pragmas)`. If Polyspace finds a pragma not in the allowed pragma list, a violation is raised.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Preprocessing Directives
Category: Document

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2016b

MISRA C++:2008 Rule 17-0-1

Reserved identifiers, macros and functions in the Standard Library shall not be defined, redefined or undefined

Description

Rule Definition

Reserved identifiers, macros and functions in the Standard Library shall not be defined, redefined or undefined.

Rationale

Redefining or undefining reserved identifiers, macros and functions from the Standard Library is not good practice. In some cases, these actions can lead to undefined behavior.

Polyspace Implementation

The checker raises a violation if identifiers and macros from the Standard Library are defined, redefined or undefined.

In general, the checker considers identifiers and macros that begin with an underscore followed by an uppercase letter as reserved for the Standard Library.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Library Introduction
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 17-0-2

The names of standard library macros and objects shall not be reused

Description

Rule Definition

The names of standard library macros and objects shall not be reused.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Library Introduction
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 17-0-3

The names of standard library functions shall not be overridden

Description

Rule Definition

The names of standard library functions shall not be overridden.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Library Introduction
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2018a

MISRA C++:2008 Rule 17-0-5

The setjmp macro and the longjmp function shall not be used

Description

Rule Definition

The setjmp macro and the longjmp function shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Library Introduction
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 18-0-1

The C library shall not be used

Description

Rule Definition

The C library shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Language Support Library
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 18-0-2

The library functions atof, atoi and atol from library <cstdlib> shall not be used

Description

Rule Definition

The library functions atof, atoi and atol from library <cstdlib> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Language Support Library
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 18-0-3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used

Description

Rule Definition

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Language Support Library
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 18-0-4

The time handling functions of library <ctime> shall not be used

Description

Rule Definition

The time handling functions of library <ctime> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Language Support Library
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 18-0-5

The unbounded functions of library <cstring> shall not be used

Description

Rule Definition

The unbounded functions of library <cstring> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Language Support Library
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 18-2-1

The macro offsetof shall not be used

Description

Rule Definition

The macro offsetof shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Language Support Library
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 18-4-1

Dynamic heap memory allocation shall not be used

Description

Rule Definition

Dynamic heap memory allocation shall not be used.

Rationale

Dynamic memory allocation uses heap memory, which can lead to issues such as memory leaks, data inconsistency, memory exhaustion, and nondeterministic behavior.

Polyspace Implementation

The checker flags uses of the `malloc`, `calloc`, `realloc` and `free` functions, and non-placement versions of the `new` and `delete` operator.

The checker also flags uses of the `alloca` function. Though memory leak cannot happen with the `alloca` function, other issues associated with dynamic memory allocation can still occur.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information
Group: Language Support Library
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 18-7-1

The signal handling facilities of <csignal> shall not be used

Description

Rule Definition

The signal handling facilities of <csignal> shall not be used.

Rationale

Signal handling functions such as `signal` contains undefined and implementation-specific behavior.

You have to be very careful when using `signal` to avoid these behaviors.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Check Information

Group: Language Support Library
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 19-3-1

The error indicator errno shall not be used

Description

Rule Definition

The error indicator errno shall not be used.

Rationale

Observing this rule encourages the good practice of not relying on `errno` to check error conditions.

Checking `errno` is not sufficient to guarantee absence of errors. Functions such as `fopen` might not set `errno` on error conditions. Often, you have to check the return value of such functions for error conditions.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of `errno`

```
#include <cstdlib>
#include <cerrno>

void func (const char* str) {
    errno = 0;  // Noncompliant
    int i = atoi(str);
    if(errno != 0) { // Noncompliant
        //Handle Error
    }
}
```

The use of `errno` violates this rule. The function `atoi` is not required to set `errno` if the input string cannot be converted to an integer. Checking `errno` later does not safeguard against possible failures in conversion.

Check Information
Group: Diagnostic Library
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

MISRA C++:2008 Rule 27-0-1

The stream input/output library <cstdio> shall not be used

Description

Rule Definition

The stream input/output library <cstdio> shall not be used.

Rationale

Functions in `cstdio` such as `gets`, `fgetpos`, `fopen`, `ftell`, etc. have unspecified, undefined and implementation-defined behavior.

For instance:

- The `gets` function:

  ```
  char * gets ( char * buf );
  ```

  does not check if the number of characters provided at the standard input exceeds the buffer `buf`. The function can have unexpected behavior when the input exceeds the buffer.
- The `fopen` function has implementation-specific behavior related to whether it sets `errno` on errors or whether it accepts additional characters following the standard mode specifiers.

Troubleshooting

If you expect a rule violation but do not see it, refer to "Coding Standard Violations Not Displayed".

Examples

Use of `gets`

```
#include <cstdio>

void func()
{
    char array[10];
    fgets(array, sizeof array, stdin); //Noncompliant
}
```

The use of `fgets` violates this rule.

Check Information
Group: Input/output Library
Category: Required

See Also

Topics
"Check for Coding Standard Violations"

Introduced in R2013b

Code Metrics

Comment Density

Ratio of number of comments to number of statements

Description

The metric specifies the ratio of comments to statements expressed as a percentage.

Based on HIS specifications:

- Multi-line comments count as one comment.

  For instance, the following constitutes one comment:

  ```
  // This function implements
  // regular maintenance on an internal database
  ```
- Comments that start with the source code line do not count as comments.

  For instance, this comment does not count as a comment for the metric but counts as a statement instead:

  ```
    remove(i); // Remove employee record
  ```
- A statement typically ends with a semi-colon with some exceptions. Exceptions include semi-colons in `for` loops or structure field declarations.

  For instance, the initialization, condition and increment within parentheses in a `for` loop is counted as one statement. The following counts as one statement:

  ```
  for(i=0; i <100; i++)
  ```

  If you also declare the loop counter at initialization, it counts as two statements.

The recommended lower limit for this metric is 20. For better readability of your code, try to place at least one comment for every five statements.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Examples

Comment Density Calculation

```
    struct record {
    char name[40];
    long double salary;
    int isEmployed;
};

struct record dataBase[100];
```

```
struct record fetch(void);
void remove(int);

void maintenanceRoutines() {
// This function implements
// regular maintenance on an internal database
    int i;
    struct record tempRecord;

    for(i=0; i <100; i++) {
        tempRecord = fetch(); // This function fetches a record
        // from the database
        if(tempRecord.isEmployed == 0)
            remove(i);          // Remove employee record
        //from the database
    }
}
```

In this example, the comment density is 38. The calculation is done as follows:

Code	Running Total of Comments	Running Total of Statements

`struct record {`
`    char name[40];`
`    long double salary;`
`    int isEmployed;`
`};`
	0	1

`struct record dataBase[100];`
`struct record fetch(void);`
`void remove(int);`
	0	4

`void maintenanceRoutines() {`	0	4
`// This function implements` `// regular maintenance on an internal database`	1	4
`int i;` `struct record tempRecord;`	1	6
`for(i=0; i <100; i++) {`	1	6

` tempRecord = fetch(); // This`
`        function fetches a record`
`            // from the database`
	2	7

`if(tempRecord.isEmployed == 0)`
`            remove(i);`
`        // Remove employee record`
`        //from the database`
`    }`
`}`
	3	8

There are 3 comments and 8 statements. The comment density is 3/8*100 = 38.

Metric Information
Group: File
Acronym: COMF
HIS Metric: Yes

See Also
Calculate code metrics (-code-metrics)

Cyclomatic Complexity

Number of linearly independent paths in function body

Description

This metric calculates the number of decision points in a function and adds one to the total. A decision point is a statement that causes your program to branch into two paths.

The recommended upper limit for this metric is 10. If the cyclomatic complexity is high, the code is both difficult to read and can cause more orange checks. Therefore, try to limit the value of this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Computation Details

The metric calculation uses the following rules to identify decision points:

- An `if` statement is one decision point.
- The statements `for` and `while` count as one decision point, even when no condition is evaluated, for example, in infinite loops.
- Boolean combinations (`&&, ||`) do not count as decision points.
- `case` statements do not count as decision points unless they are followed by a `break` statement. For instance, this code has a cyclomatic complexity of two:

```
switch(num) {
        case 0:
        case 1:
        case 2:
            break;
        case 3:
        case 4:
    }
```

- The calculation is done after preprocessing:

  - Macros are expanded.
  - Conditional compilation is applied. The blocks hidden by preprocessing directives are ignored.

Examples

Function with Nested `if` Statements

```
int foo(int x,int y)
{
```

```
        int flag;
        if (x <= 0)
            /* Decision point 1*/
            flag = 1;
        else
        {
            if (x < y )
                /* Decision point 2*/
                flag = 1;
            else if (x==y)
                /* Decision point 3*/
                flag = 0;
            else
                flag = -1;
        }
        return flag;
}
```

In this example, the cyclomatic complexity of `foo` is 4.

Function with ? Operator

```
int foo (int x, int y) {
    if((x <0) ||(y < 0))
        /* Decision point 1*/
        return 0;
    else
        return (x > y ? x: y);
        /* Decision point 2*/
}
```

In this example, the cyclomatic complexity of `foo` is 3. The ? operator is the second decision point.

Function with switch Statement

```
#include <stdio.h>


int foo(int x,int y, int ch)
{
    int val = 0;
    switch(ch) {
    case 1:
        /* Decision point 1*/
        val = x + y;
        break;
    case 2:
        /* Decision point 2*/
        val = x - y;
        break;
    default:
        printf("Invalid choice.");
    }
    return val;
}
```

In this example, the cyclomatic complexity of `foo` is 3.

Function with Nesting of Different Control-Flow Statements

```
int foo(int x,int y, int bound)
{
    int count = 0;
    if (x <= y)
        /* Decision point 1*/
        count = 1;
    else
        while(x>y) {
            /* Decision point 2*/
            x--;
            if(count< bound) {
                /* Decision point 3*/
                count++;
            }
        }
    return count;
}
```

In this example, the cyclomatic complexity of `foo` is 4.

Metric Information

Group: Function
Acronym: VG
HIS Metric: Yes

See Also

Calculate code metrics (-code-metrics)

Estimated Function Coupling

Measure of complexity between levels of call tree

Description

This metric provides an approximate measure of complexity between different levels of the call tree. The metric is defined as:

number of call occurrences – number of function definitions + 1

If there are more function definitions than function calls, the estimated function coupling result is negative.

This metric:

- Counts function calls and function definitions in the current file only.

  It does not count function definitions in a header file included in the current file.
- Treats `static` and `inline` functions like any other function.

Examples

Same Function Called Multiple Times

```
void checkBounds(int *);
int getUnboundedValue();

int getBoundedValue(void) {
    int num = getUnboundedValue();
    checkBounds(&num);
    return num;
}

void main() {
    int input1=getBoundedValue(), input2= getBoundedValue(), prod;
    prod = input1 * input2;
    checkBounds(&prod);
}
```

In this example, there are:

- 5 call occurrences. Both `getBoundedValue` and `checkBounds` are called twice and `getUnboundedValue` is called once.
- 2 function definitions. `main` and `getBoundedValue` are defined.

Therefore, the Estimated function coupling is 5 - 2 + 1 = 4.

Negative Estimated Function Coupling

```
int foobar(int a, int b){
```

```
    return a+b;
}

int bar(int b){
    return b+2;
}

int foo(int a){
    return a<<2;
}

int main(int x){
    foobar(x,x+2);
    return 0;
}
```

This example shows how you can get a negative estimated function coupling result. In this example, you see:

- `1` function call in `main`.

- `4` defined functions: `foobar`, `bar`, `foo`, and `main`.

Therefore, the estimated function coupling is `1 - 4 + 1 = -2`.

Metric Information

Group: File
Acronym: FCO
HIS Metric: No

See Also

`Number of Call Occurrences` | `Calculate code metrics (-code-metrics)`

Higher Estimate of Local Variable Size

Total size of all local variables in function

Description

This metric provides a conservative estimate of the total size of local variables in a function. The metric is the sum of the following sizes in bytes:

- Size of function return value
- Sizes of function parameters
- Sizes of local variables
- Additional padding introduced for memory alignment

Your actual stack usage due to local variables can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. For instance, compilers store the address to which the execution returns following the function call. When computing this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. When computing this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage due to local variables.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric also takes into account `#pragma pack` directives in your code.

Examples

All Variables of Same Type

```
int flag();


int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, assuming 4 bytes for `int`, the higher estimate of local variable size for `func`is 28. The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	4	4
Parameter `param`	4	8
Local variables `var_1` and `var_2`	4+4=8	16
Local variables defined in the `if` condition	(4+4)+4=12 The size of variables in the first branch is eight bytes. The size in the second branch is four bytes. The sum of the two branches is 12 bytes.	28

No padding is introduced for memory alignment because all the variables involved have the same type.

Variables of Different Types

```
char func(char param) {
  int var_1;
  char var_2;
  double var_3;
}
```

In this example, assuming one byte for `char`, four bytes for `int` and eight bytes for `double` and four bytes for alignment, the higher estimate of local variable size for `func` is 20. The alignment is usually the word size on your platform. In your Polyspace project, you specify the alignment through your target processor. For more information, see the Alignment column in `Target processor type (-target)`.

The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	1	1
Additional padding introduced before `param` is stored	0 No memory alignment is required because the next variable `param` has the same size.	1
Parameter `param`	1	2
Additional padding introduced before `var_1` is stored	2 Memory must be aligned using padding because the next variable `var_1` requires four bytes. The storage must start from a memory address at a multiple of four.	4

Variable	Size (in Bytes)	Running Total
`var_1`	4	8
Additional padding introduced before `var_2` is stored	0 No memory alignment is required because the next variable `var_2` has smaller size.	8
`var_2`	1	9
Additional padding introduced before `var_3` is stored	3 Memory must be aligned using padding because the next variable `var_3` has eight bytes. The storage must start from a memory address at a multiple of the alignment, four bytes.	12
`var_3`	8	20

The rules for the amount of padding are:

- If the next variable stored has the same or smaller size, no padding is required.
- If the next variable has a greater size:
  - If the variable size is the same as or less than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of its size.
  - If the variable size is greater than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of the alignment.

C++ Methods and Objects

```
class MySimpleClass {
  public:

    MySimpleClass() {};

    MySimpleClass(int) {};

    ~MySimpleClass() {};
};


int main() {
  MySimpleClass c;
  return 0;
}
```

In this example, the estimated local variable sizes are:

- Constructor `MySimpleClass::MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer, which is an implicit argument to the constructor. You specify the pointer size using the option `Target processor type (-target)`.

- Constructor `MySimpleClass::MySimpleClass(int)`: Eight bytes.

  The size comes from the `this` pointer and the `int` argument.
- Destructor `MySimpleClass::~MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer.
- `main()`: Five bytes.

  The size comes from the `int` return value and the size of object `c`. The minimum size of an object is the alignment that you specify using the option `Target processor type (-target)`.

C++ Functions with Object Arguments

```
class MyClass {
  public:
    MyClass() {};
    MyClass(int) {};
    ~MyClass() {};
  private:
    int i[10];
};
void func1(const MyClass& c) {
}


void func2() {
  func1(4);
}
```

In this example, the estimated local variable size for `func2` is 40 bytes. When `func2` calls `func1`, a temporary object of the class `MyClass` is created. The object has ten `int` variables, each with a size of four bytes.

Metric Information
Group: Function
Acronym: LOCAL_VARS_MAX
HIS Metric: No

See Also
Lower Estimate of Local Variable Size | Calculate code metrics (-code-metrics)

Introduced in R2016b

Language Scope

Language scope

Description

This metric measures the cost of maintaining or changing a function. It is calculated as:

```
(N1 + N2)/(n1 + n2)
```

Where:

- `N1` is the number of occurrences of operators.

  Other than identifiers (variable or function names) and literal constants, everything else counts as operators.
- `N2` is the number of occurrences of operands.
- `n1` is the number of distinct operators.
- `n2` is the number of distinct operands.

  The metric considers a literal constant with a suffix as different from the constant without the suffix. For instance, `0` and `0U` are considered different.

When reporting this metric, Polyspace rounds the calculated language scope to the first decimal place. Because the intent of this metric is to indicate the maintainability of a function, language scope of functions defined within local classes are not computed.

Tip To find `N1 + N2`, count the total number of tokens. To find `n1 + n2`, count the number of unique tokens.

The recommended upper limit for this metric is 4. For lower maintenance cost for a function, try to enforce an upper limit on this metric. For instance, if the same operand occurs many times, to change the operand name, you have to make many substitutions.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Examples

Language Scope Calculation

```
int g(int);
int f(int i)
{
    if (i == 1)
        return i;
```

```
    else
        return i * g(i-1);
}
```

In this example:

- `N1` = 19.

- `N2` = 9.

- `n1` = 12.

    The distinct operators are `int`, `(`, `)`, `{`, `if`, `==`, `return`, `else`, `*`, `-`, `;`, `}`.

- `n2` = 4.

    The distinct operands are `f`, `i`, `1` and `g`.

The language scope of `f` is $(19 + 9) / (12 + 4) = 1.8$.

C++ Namespaces in Language Scope Calculation

```
namespace std {
  int func2() {
    return 123;
  }
};

namespace my_namespace {
  using namespace std;

  int func1(int a, int b) {
    return func2();
  }
};
```

In this example, the namespace `std` is implicitly associated with `func2`. The language scope computation treats `func2()` as `std::func2()`. Likewise, the computation treats `func1()` as `my_namespace::func1()`.

For instance, the language scope value for `func1` is 1.3. To break down this calculation:

- `N1 + N2` = 20.

- `n1 + n2` = 15.

    The distinct operators are `int`, `::`, `(`, comma, `)`, `{`, `return`, `;`, and `}`.

    The distinct operands are `my_namespace`, `func1`, `a`, `b`, `std`, and `func2`.

Metric Information
Group: Function
Acronym: VOCF
HIS Metric: Yes

See Also
Calculate code metrics (-code-metrics)

Lower Estimate of Local Variable Size

Total size of local variables in function taking nested scopes into account

Description

This metric provides an optimistic estimate of the total size of local variables in a function. The metric is the sum of the following sizes in bytes:

- Size of function return value
- Sizes of function parameters
- Sizes of local variables

  Suppose that the function has variable definitions in nested scopes as follows:

  ```
  type func (type param_1, ...) {

    {
      /* Scope 1 */
      type var_1, ...;
    }
    {
      /* Scope 2 */
      type var_2, ...;
    }
  }
  ```

  The software computes the total variable size in each scope and uses whichever total is greatest. For instance, if a conditional statement has variable definitions, the software computes the total variable size in each branch, and then uses whichever total is greatest. If a nested scope itself has further nested scopes, the same process is repeated for the inner scopes.

  A variable defined in a nested scope is not visible outside the scope. Therefore, some compilers reuse stack space for variables defined in separate scopes. This metric provides a more accurate estimate of stack usage for such compilers. Otherwise, use the metric `Higher Estimate of Local Variable Size`. This metric adds the size of all local variables, whether or not they are defined in nested scopes.

- Additional padding introduced for memory alignment

Your actual stack usage due to local variables can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When computing this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When computing this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage due to local variables.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric also takes into account `#pragma pack` directives in your code.

Examples

All Variables of Same Type

```
int flag();

int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, assuming four bytes for `int`, the lower estimate of local variable size is 24. The breakup of the metric is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	4	4
Parameter `param`	4	8
Local variables `var_1` and `var_2`	4+4=8	16
Local variables defined in the `if` condition	`max(4+4,4)= 8` The size of variables in the first branch is eight bytes. The size in the second branch is four bytes. The maximum of the two branches is eight bytes.	24

No padding is introduced for memory alignment because all the variables involved have the same type.

Variables of Different Types

```
char func(char param) {
  int var_1;
  char var_2;
  double var_3;
}
```

In this example, assuming one byte for `char`, four bytes for `int`, eight bytes for `double` and four bytes for alignment, the lower estimate of local variable size is 20. The alignment is usually the word size on your platform. In your Polyspace project, you specify the alignment through your target processor. For more information, see the Alignment column in `Target processor type (-target)`.

The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	1	1
Additional padding introduced before `param` is stored	0 No memory alignment is required because the next variable `param` has the same size.	1
Parameter `param`	1	2
Additional padding introduced before `var_1` is stored	2 Memory must be aligned using padding because the next variable `var_1` requires four bytes. The storage must start from a memory address at a multiple of four.	4
`var_1`	4	8
Additional padding introduced before `var_2` is stored	0 No memory alignment is required because the next variable `var_2` has smaller size.	8
`var_2`	1	9
Additional padding introduced before `var_3` is stored	3 Memory must be aligned using padding because the next variable `var_3` requires eight bytes. The storage must start from a memory address at a multiple of the alignment, four bytes.	12
`var_3`	8	20

The rules for the amount of padding are:

- If the next variable stored has the same or smaller size, no padding is required.
- If the next variable has a greater size:
  - If the variable size is the same as or less than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of its size.
  - If the variable size is greater than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of the alignment.

C++ Methods and Objects

```
class MySimpleClass {
  public:
```

```
    MySimpleClass() {};

    MySimpleClass(int) {};

    ~MySimpleClass() {};
};


int main() {
  MySimpleClass c;
  return 0;
}
```

In this example, the estimated local variable sizes are:

- Constructor `MySimpleClass::MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer, which is an implicit argument to the constructor. You specify the pointer size using the option `Target processor type (-target)`.

- Constructor `MySimpleClass::MySimpleClass(int)`: Eight bytes.

  The size comes from the `this` pointer and the `int` argument.

- Destructor `MySimpleClass::~MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer.

- `main()`: Five bytes.

  The size comes from the `int` return value and the size of object `c`. The minimum size of an object is the alignment that you specify using the option `Target processor type (-target)`.

C++ Functions with Object Arguments

```
class MyClass {
  public:
    MyClass() {};
    MyClass(int) {};
    ~MyClass() {};
  private:
    int i[10];
};
void func1(const MyClass& c) {
}


void func2() {
  func1(4);
}
```

In this example, the estimated local variable size for `func2()` is 40 bytes. When `func2()` calls `func1()`, a temporary object of the class `MyClass` is created. The object has ten `int` variables, each with a size of four bytes.

Metric Information
Group: Function
Acronym: LOCAL_VARS_MIN
HIS Metric: No

See Also
Higher Estimate of Local Variable Size | Calculate code metrics (-code-metrics)

Introduced in R2016b

Maximum Stack Usage

Total size of local variables in function plus maximum stack usage from callees

Description

This metric is reported in a Code Prover analysis only.

This metric provides a conservative estimate of the stack usage by a function. The metric is the sum of these sizes in bytes:

- `Higher Estimate of Local Variable Size`
- Maximum value from the stack usages of the function callees. The computation uses the maximum stack usage of each callee.

  For instance, in this example, the maximum stack usage of `func` is the same as the maximum stack usage of `func1` or `func2`, *whichever is greater*.

  ```
  void func(void) {
      func1();
      func2();
  }
  ```

  If the function calls are in different branches of a conditional statement, this metric considers the branch with the greatest stack usage.

  The analysis does the stack size estimation later on when it has resolved which function calls actually occur. For instance, if a function call occurs in unreachable code, the stack size does not take the call into account. The analysis can also take into account calls through function pointers.

Your actual stack usage can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When estimating this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When estimating this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric takes into account `#pragma pack` directives in your code.

Examples

Function with One Callee

```
double func(int);
double func2(int);
```

```
double func(int status) {
    double res = func2(status);
    return res;
}


double func2(int status) {
    double res;
    if(status == 0) {
      int temp;
      res = 0.0;
    }
    else {
      double temp;
      res = 1.0;
    }
    return res;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func2`: 32 bytes

  This value includes the sizes of its parameter (4 bytes), local variable `res` (8 bytes), local variable `temp` counted twice (4+8=12 bytes), and return value (8 bytes).

  The metric does not take into account that the first `temp` is no longer live when the second `temp` is defined.

- `func`: 52 bytes

  This value includes the sizes of its parameter, local variable `res`, and return value, a total of 20 bytes. This value includes the 32 bytes of maximum stack usage by its callee, `func2`.

Function with Multiple Callees

```
void func1(int);
void func2(void);



void func(int status) {
    func1(status);
    func2();
}


void func1(int status) {
    if(status == 0) {
      int val;
    }
    else {
      double val2;
    }
}
```

```
void func2(void) {
    double val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 16 bytes

  This value includes the sizes of its parameter (4 bytes) and local variables `val` and `val2` (4+8=12 bytes).
- `func2`: 8 bytes
- `func`: 20 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum of stack usages of `func1` and `func2` (16 bytes).

Function with Multiple Callees in Different Branches

```
void func1(void);
void func2(void);


void func(int status) {
    if(status==0)
        func1();
    else
        func2();
}

void func1(void) {
    double val;
}


void func2(void) {
    int val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 8 bytes
- `func2`: 4 bytes
- `func`: 12 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum stack usage from the two branches (8 bytes).

Functions with Variable Number of Parameters (Variadic Functions)

```
#include <stdarg.h>


void fun_vararg(int x, ...) {
  va_list ap;
```

```
  va_start(ap, x);
  int i;
  for (i=0; i<x; i++) {
    int j = va_arg(ap, int);
  }
  va_end(ap);
}


void call_fun_vararg1(void) {
  long long int l = 0;
  fun_vararg(3, 4, 5, 6, l);
}


void call_fun_vararg2(void) {
  fun_vararg(1,0);
}
```

In this function, `fun_vararg` is a function with variable number of parameters. The maximum stack usage of `fun_vararg` takes into account the call to `fun_vararg` with the maximum number of arguments. The call with the maximum number of arguments is the call in `call_fun_vararg1` with five arguments (one for the fixed parameter and four for the variable parameters). The maximum stack usages are:

- `fun_vararg`: 36 bytes.

  This value takes into account:

  - The size of the fixed parameter `x` (4 bytes).
  - The sizes of the variable parameters from the call with the maximum number of parameters. In that call, there are four variable arguments: three `int` and one `long long int` variable (3 times 4 + 1 times 8 = 20 bytes).
  - The sizes of the local variables `i`, `j` and `ap` (12 bytes). The size of the `va_list` variable uses the pointer size defined in the target (in this case, 4 bytes).
- `call_fun_vararg1`: 44 bytes.

  This value takes into account:

  - The stack size usage of `fun_vararg` with five arguments (36 bytes).
  - The size of local variable `l` (8 bytes).
- `call_fun_vararg2`: 20 bytes.

  Since `call_fun_vararg2` has no local variables, this value is the same as the stack size usage of `fun_vararg` with two arguments (20 bytes, of which 12 bytes are for the local variables and 8 bytes are for the two parameters of `fun_vararg`).

Metric Information
Group: Function
Acronym: MAX_STACK
HIS Metric: No

See Also

Minimum Stack Usage | Program Maximum Stack Usage | Higher Estimate of Local Variable Size | Calculate code metrics (-code-metrics)

Topics
"Determination of Program Stack Usage" on page 4-37

Introduced in R2017b

Minimum Stack Usage

Total size of local variables in function taking nested scopes into account plus maximum stack usage from callees

Description

This metric is reported in a Code Prover analysis only.

This metric provides an optimistic estimate of the stack usage by a function. Unlike the metric `Maximum Stack Usage`, this metric takes nested scopes into account. For instance, if variables are defined in two mutually exclusive branches of a conditional statement, the metric considers that the stack space allocated to the variables in one branch can be reused in the other branch.

The metric is the sum of these sizes in bytes:

- `Lower Estimate of Local Variable Size`.

- Maximum value from the stack usages of the function callees. The computation uses the minimum stack usage of each callee.

  For instance, in this example, the minimum stack usage of `func` is the same as the minimum stack usage of `func1` or `func2`, *whichever is greater*.

  ```
  void func(void) {
      func1();
      func2();
  }
  ```

  If the function calls are in different branches of a conditional statement, this metric considers the branch with the least stack usage.

  The analysis does the stack size estimation later on when it has resolved which function calls actually occur. For instance, if a function call occurs in unreachable code, the stack size does not take the call into account. The analysis can also take into account calls through function pointers.

Your actual stack usage can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When estimating this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When estimating this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric takes into account `#pragma pack` directives in your code.

Examples

Function with One Callee

```
double func2(int);


double func(int status) {
    double res = func2(status);
    return res;
}


double func2(int status) {
    double res;
    if(status == 0) {
      int temp;
      res = 0.0;
    }
    else {
      double temp;
      res = 1.0;
    }
    return res;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the minimum stack usages are:

- `func2`: 28 bytes

  This value includes the sizes of its parameter (4 bytes), local variable `res` (8 bytes), one of the two local variables `temp` (8 bytes), and return value (8 bytes).

  The metric takes into account that the first `temp` is no longer live when the second `temp` is defined. It uses the variable `temp` with data type `double` because its size is greater.

- `func`: 48 bytes

  This value includes the sizes of its parameter, local variable `res`, and return value, a total of 20 bytes. This value includes the 28 bytes of minimum stack usage by its callee, `func2`.

Function with Multiple Callees

```
void func1(int);
void func2(void);


void func(int status) {
    func1(status);
    func2();
}


void func1(int status) {
    if(status == 0) {
      int val;
```

```
      }
    else {
      double val2;
    }
}


void func2(void) {
    double val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the minimum stack usages are:

- `func1`: 12 bytes

  This value includes the sizes of its parameter (4 bytes) and the larger of the two local variables, in this case, `val2` (8 bytes).
- `func2`: 8 bytes
- `func`: 16 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum of stack usages of `func1` and `func2` (12 bytes).

Function with Multiple Callees in Different Branches

```
void func1(void);
void func2(void);



void func(int status) {
    if(status==0)
        func1();
    else
        func2();
}


void func1(void) {
    double val;
}


void func2(void) {
    int val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the minimum stack usages are:

- `func1`: 8 bytes
- `func2`: 4 bytes
- `func`: 8 bytes

This value includes the sizes of its parameter (4 bytes) and the minimum stack usage from the two branches (4 bytes).

Functions with Variable Number of Parameters (Variadic Functions)

```
#include <stdarg.h>


void fun_vararg(int x, ...) {
  va_list ap;
  va_start(ap, x);
  int i;
  for (i=0; i<x; i++) {
    int j = va_arg(ap, int);
  }
  va_end(ap);
}



void call_fun_vararg1(void) {
  long long int l = 0;
  fun_vararg(3, 4, 5, 6, l);
}



void call_fun_vararg2(void) {
  fun_vararg(1,0);
}
```

In this function, `fun_vararg` is a function with variable number of parameters. The minimum stack usage of `fun_vararg` takes into account the call to `fun_vararg` with the minimum number of arguments. The call with the minimum number of arguments is the call in `call_fun_vararg2` with two arguments (one for the fixed parameter and one for the variable parameter). The minimum stack usages are:

- `fun_vararg`: 20 bytes.

  This value takes into account:

  - The size of the fixed parameter `x` (4 bytes).
  - The sizes of the variable parameters from the call with the minimum number of parameters. In that call, there is only one variable argument of type `int` (4 bytes).
  - The sizes of the local variables `i`, `j` and `ap` (12 bytes). The size of the `va_list` variable uses the pointer size defined in the target (in this case, 4 bytes).

- `call_fun_vararg1`: 44 bytes.

  This value takes into account:

  - The stack size usage of `fun_vararg` with five arguments (36 bytes, of which 12 bytes are for the local variable sizes and 20 bytes are for the fixed and variable parameters of `fun_vararg`).
  - The size of local variable `l` (8 bytes).

- `call_fun_vararg2`: 20 bytes.

Since `call_fun_vararg2` has no local variables, this value is the same as the stack size usage of `fun_vararg` with two arguments (20 bytes).

Metric Information

Group: Function
Acronym: MIN_STACK
HIS Metric: No

See Also

Program Minimum Stack Usage | Lower Estimate of Local Variable Size | Maximum Stack Usage | Calculate code metrics (-code-metrics)

Topics
"Determination of Program Stack Usage" on page 4-37

Introduced in R2017b

Number of Call Levels

Maximum depth of nesting of control flow structures

Description

This metric specifies the maximum nesting depth of control flow statements such as `if`, `switch`, `for`, or `while` in a function. A function without control-flow statements has a call level 1.

The recommended upper limit for this metric is 4. For better readability of your code, try to enforce an upper limit for this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Examples

Function with Nested `if` Statements

```
int foo(int x,int y)
{
    int flag = 0;
    if (x <= 0)
        /* Call level 1*/
        flag = 1;
    else
    {
        if (x <= y )
            /* Call level 2*/
            flag = 1;
        else
            flag = -1;
    }
    return flag;
}
```

In this example, the number of call levels of `foo` is 2.

Function with Nesting of Different Control-Flow Statements

```
int foo(int x,int y, int bound)
{
    int count = 0;
    if (x <= y)
        /* Call level 1*/
        count = 1;
    else
        while(x>y) {
```

```
            /* Call level 2*/
            x--;
            if(count< bound) {
                /* Call level 3*/
                count++;
            }
        }
    return count;
}
```

In this example, the number of call levels of `foo` is 3.

Metric Information
Group: Function
Acronym: LEVEL
HIS Metric: Yes

See Also
`Calculate code metrics (-code-metrics)`

Number of Call Occurrences

Number of calls in function body

Description

This metric specifies the number of function calls in the body of a function.

Calls through a function pointer are not counted. Calls in unreachable code and calls to standard library functions are counted. `assert` is considered as a macro and not a function, so it is not counted.

When calculating this metric in C++ code, Polyspace ignores the implicit functions that the compiler generates, such as default constructors and destructors. User-defined constructors and destructors are counted as function calls. In a class hierarchy, if a base class has user-defined constructors, Polyspace counts the corresponding constructors of the derived classes as functions.

Examples

Same Function Called Multiple Times

```
int func1(void);
int func2(void);


int foo() {
    return (func1() + func1()*func1() + 2*func2());
}
```

In this example, the number of call occurrences in `foo` is 4.

Function Called in a Loop

```
#include<stdio.h>
int getVal(void);


void fillArraySize10(int *arr) {
    for(int i=0; i<10; i++)
        arr[i]=getVal();
}

int getVal(void) {
    int val;
    printf("Enter a value:");
    scanf("%d", &val);
    return val;
}
```

In this example, the number of call occurrences in `fillArraySize10` is 1.

Recursive Function

```
#include <stdio.h>
int fibonacci(int);
```

```
void main() {
 int count;
 printf("How many numbers ?");
 scanf("%d",&count);
 fibonacci(count);
}

int fibonacci(int num)
{
   if ( num == 0 )
      return 0;
   else if ( num == 1 )
      return 1;
   else
      return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of call occurrences in `fibonacci` is 2.

Constructor Function

```
                #include<iostream>
class A{
    public:
    A(){
        std::cout<<"Create A\n";
    }
    ~A() = default;
    A(const A&)=default;
    A(A&&) = default;
    virtual void bar(){ std::cout<<"A";}
};
class B: public A{
    public:
    B() = default;
     void bar() override {std::cout<<"B";}
};

void func(A& a){
    a.bar();
}

int main(){
    A obj;
    A obj2 = obj;
    B objB;
    func(obj);
    return 0;
}
```

In this example, the number of call occurances in `main` is three:

1   The constructor of class A in `A obj;`. This user defined constructor counts as a function call.

2   The constructor of class B in `B objB;`. Because the constructor of the base class A is user-defined, the constructor of B counts as a function call even though `B::B()` is declared as `=default`.

3 The call to function `func`.

The class A uses the default or implicit copy constructor. The call to the copy constructor in `A obj2 = obj;` does not count as a function call.

Metric Information
Group: Function
Acronym: NCALLS
HIS Metric: No

See Also
`Number of Called Functions` | `Calculate code metrics (-code-metrics)`

Number of Called Functions

Number of distinct functions called within the body of a function

Description

This metric specifies the number of distinct functions that are called by a function within its body.

Calls through a function pointer are not counted. Calls in unreachable code and calls to standard library functions are counted. `assert` is considered as a macro and not a function, so it is not counted.

When calculating this metric in C++ code, Polyspace ignores the implicit functions that the compiler generates, such as default constructors and destructors. User-defined constructors and destructors are counted as function calls. In a class hierarchy, if a base class has user-defined constructors, Polyspace counts the corresponding constructors of the derived classes as functions.

The recommended upper limit for this metric is 7. For more self-contained code, try to enforce an upper limit on this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Examples

Same Function Called Multiple Times

```
int func1(void);
int func2(void);

int foo() {
    return (func1() + func1()*func1() + 2*func2());
}
```

In this example, the number of called functions in `foo` is 2. The called functions are `func1` and `func2`.

Recursive Function

```
#include <stdio.h>
int fibonacci(int);

void main() {
 int count;
 printf("How many numbers ?");
 scanf("%d",&count);
 fibonacci(count);
}

int fibonacci(int num)
```

```
{
    if ( num == 0 )
        return 0;
    else if ( num == 1 )
        return 1;
    else
        return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of called functions in `fibonacci` is 1. The called function is `fibonacci` itself.

Constructor Function

```
            #include<iostream>
class A{
    public:
    A(){
        std::cout<<"Create A\n";
    }
    ~A() = default;
    A(const A&)=default;
    A(A&&) = default;
    virtual void bar(){ std::cout<<"A";}
};
class B: public A{
    public:
    B() = default;
     void bar() override {std::cout<<"B";}
};

void func(A& a){
    a.bar();
}

int main(){
    A obj;
    A obj2 = obj;
    B objB;
    func(obj);
    return 0;
}
```

In this example, the number of called function in `main` is three:

1 The constructor of class A. This user defined constructor counts as a function call.

2 The constructor of class B. Because the constructor of the base class A is user-defined, the constructor of B counts as a function call even though `B::B()` is declared as =`default`.

3 The function `func`.

The class A uses the default or implicit copy constructor. The call to the copy constructor in `A obj2 = obj;` does not count as a function call.

Metric Information

Group: Function
Acronym: CALLS
HIS Metric: Yes

See Also

Number of Call Occurrences|Number of Calling Functions|Calculate code metrics (-code-metrics)

Number of Calling Functions

Number of distinct callers of a function

Description

This metric measures the number of distinct callers of a function.

In C++ , Polyspace does not calculate this metric for virtual functions and compiler generated implicit functions, such as default constructors and destructors. The metric is calculated for user-defined constructors and destructors. In a class hierarchy, if a base class has user-defined constructors, Polyspace counts this metric for corresponding constructors of the derived classes.

The recommended upper limit for this metric is 5. For more self-contained code, try to enforce an upper limit on this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Computation Details

Note that the metric:

- Takes into account direct callers only.
- Does not consider calls through a function pointer.
- Takes into account all function calls, including ones in unreachable code.

  However, if a caller calls a function more than once, the caller is counted only once when this metric is calculated.

Examples

Same Function Calling a Function Multiple Times

```
#include <stdio.h>


    int getVal() {
    int myVal;
    printf("Enter a value:");
    scanf("%d", &myVal);
    return myVal;
}

int func() {
    int val=getVal();
    if(val<0)
        return 0;
    else
```

```
        return val;
}

int func2() {
    int val=getVal();
    while(val<0)
        val=getVal();
    return val;
}
```

In this example, the number of calling functions for `getVal` is 2. The calling functions are `func` and `func2`.

Recursive Function

```
#include <stdio.h>



    int fibonacci(int num)
{
   if ( num == 0 )
      return 0;
   else if ( num == 1 )
      return 1;
   else
      return ( fibonacci(num-1) + fibonacci(num-2) );
}

void main() {
 int count;
 printf("How many numbers ?");
 scanf("%d",&count);
 fibonacci(count);
}
```

In this example, the number of calling functions for `fibonacci` is 2. The calling functions are `main` and `fibonacci` itself.

Constructor Function

```
                #include<iostream>
class A{
    public:

    A(){
        std::cout<<"Create A\n";
    }
    ~A() = default;
    A(const A&)=default;
    A(A&&) = default;
    virtual void bar(){ std::cout<<"A";}
};
class B: public A{
    public:
    B() = default;
```

```
        void bar() override {std::cout<<"B";}
};

void func(A& a){
    a.bar();
}
int main(){
    A obj;
    A obj2 = obj;
    B objB;
    func(obj);
    return 0;
}
```

In this example:

- The number of calling functions for `A::A` is two. `A::A` is called once to create `obj` and again to create `objB`. Similarly, the number of calling function for `B::` is one.

- Because both `A::bar` and `B::bar` are virtual functions, Polyspace does not calculate their number of calling functions.

- The number of calling function for `func` is one.

Metric Information
Group: Function
Acronym: `CALLING`
HIS Metric: Yes

See Also
`Number of Called Functions` | `Calculate code metrics (-code-metrics)`

Number of Direct Recursions

Number of instances of a function calling itself directly

Description

This metric specifies the number of direct recursions in your project.

A direct recursion is a recursion where a function calls itself in its own body. If indirect recursions do not occur, the number of direct recursions is equal to the number of recursive functions.

The recommended upper limit for this metric is 0. To avoid the possibility of exceeding available stack space, do not use recursions in your code. To detect use of recursions, check for violations of MISRA C:2012 Rule 17.2.

To enforce limits on metrics, see "Compare Metrics Against Software Quality Objectives".

Examples

Direct Recursion

```
int getVal(void);
int sum(int val) {
    if(val<0)
        return 0;
    else
        return (val + sum(val-1));
}

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}
```

In this example, the number of direct recursions is 1.

Metric Information
Group: Project
Acronym: AP_CG_DIRECT_CYCLE
HIS Metric: Yes

See Also
MISRA C:2012 Rule 17.2 | Calculate code metrics (-code-metrics)

Number of Executable Lines

Number of executable lines in function body

Description

This metric measures the number of executable lines in a function body. When calculating the value of this metric, Polyspace excludes declarations, comments, blank lines, braces or preprocessing directives.

If the function body contains a `#include` directive, the included file source code is also calculated as part of this metric.

This metric is not calculated for C++ templates.

Examples

Function with Declarations, Braces and Comments

```
void func(int, double);

    int getSign(int arg) {
    int sign; //Excluded
    static int siNumber = 0; //Excluded
    double dNumber = 5;//Excluded
    if(arg<0) {
        sign=-1;
        func(-arg,dNumber);
        ++siNumber;
        /* func takes positive first argument */   //Excluded
    }//Excluded
    else if(arg==0)
    sign=0;
    else {
        sign=1;
        func(arg,dNumber);
        ++siNumber;
    }//Excluded
    return sign;
}//Excluded
```

In this example, the number of executable lines of `getSign` is 11. The calculation excludes:

- The declarations.
- The comment `/* ... */`.
- The lines with braces only.

Metric Information
Group: Function
Acronym: FXLN
HIS Metric: No

See Also

Number of Lines Within Body | Number of Instructions | Calculate code metrics (-code-metrics)

Number of Files

Number of source files

Description

This metric calculates the number of source files in your project.

Examples

Source File Calling Function in Included File

```
#include<iostream>
#include"pow.cpp"

double power(double, double);
//Function to calculate approximate index
 double AppxIndex(double m, double f){
     double U = (power(m,2) - 1)/(power(m,2)+2); //First term
     double V = (power(m,4) + 27*power(m,2)+38)/
     (2*power(m,2)+3);// Second term
     return (1+2*f*power(U,2)*(1+power(m,2)*U*V +
      power(m,3)/power(m,3)*(U-V)))
     /( (1-2*f*power(U,2)*(1+power(m,2)*U*V
     + power(m,3)/power(m,3)*(U-V))));
 }


 int main(){
     return 0;
 }
```

In this example, the code calls the function `power`, which is defined in an included source file `pow.cpp`. When analysing the code, Polyspace does not consider the included source file. Because the included source file is not considered in the analysis, Polyspace evaluates the number of source file to be one.

Metric Information

Group: Project
Acronym: FILES
HIS Metric: No

See Also

Number of Header Files | Calculate code metrics (-code-metrics)

Number of Function Parameters

Number of function arguments

Description

This metric measures the number of function arguments.

If ellipsis is used to denote variable number of arguments, when calculating this metric, the ellipsis is not counted.

The recommended upper limit for this metric is 5. For less dependency between functions and fewer side effects, try to enforce an upper limit on this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Examples

Function with Fixed Arguments

```
int initializeArray(int* arr, int size) {
}
```

In this example, `initializeArray` has two parameters.

Function with Type Definition in Arguments

```
int getValueInLoc(struct {int* arr; int size;}myArray, int loc) {
}
```

In this example, `getValueInLoc` has two parameters.

Function with Variable Arguments

```
double average ( int num, ... )
{
    va_list arg;
    double sum = 0;

    va_start ( arg, num );

    for ( int x = 0; x < num; x++ )
    {
        sum += va_arg ( arg, double );
    }
```

```
    va_end ( arg);

    return sum / num;
}
```

In this example, `average` has one parameter. The ellipsis denoting variable number of arguments is not counted.

Metric Information
Group: Function
Acronym: PARAM
HIS Metric: Yes

See Also
Calculate code metrics (-code-metrics)

Number of Goto Statements

Number of `goto` statements

Description

This metric measures the number of `goto` statements in a function.

`break` and `continue` statements are not counted.

The recommended upper limit on this metric is 0. For better readability of your code, avoid `goto` statements in your code. To detect use of `goto` statements, check for violations of `MISRA C:2012 Rule 15.1`.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Examples

Function with `goto` Statements

```
#define SIZE 10
int initialize(int **arr, int loc);
void printString(int *);
void printErrorMessage(void);
void printExecutionMessage(void);


int main()
{
    int *arrayOfStrings[SIZE],len[SIZE],i;
    for ( i = 0; i < SIZE; i++ )
    {
        len[i] = initialize(arrayOfStrings,i);
    }

    for ( i = 0; i < SIZE; i++ )
    {
        if(len[i] == 0)
            goto emptyString;
        else
            goto nonEmptyString;
        loop: printExecutionMessage();
    }

emptyString:
    printErrorMessage();
    goto loop;
nonEmptyString:
    printString(arrayOfStrings[i]);
```

```
    goto loop;
}
```

In this example, the function `main` has 4 `goto` statements.

Metric Information
Group: Function
Acronym: GOTO
HIS Metric: Yes

See Also
`Calculate code metrics (-code-metrics)`

Number of Header Files

Number of included header files

Description

This metric measures the number of header files in the project that is considered in an analysis. Both directly and indirectly included header files are counted.

The metric gives a slightly different number than the actual number of header files that you use because Polyspace® internal header files and header files included by those files are also counted. For the same reason, the metric can vary slightly even if you do not explicitly include new header files or remove inclusion of header files from your code. For instance, the number of Polyspace® internal header files can vary if you change your analysis options.

Examples

Header Files Not Considered in Analysis

```
#include<iostream>
#include<string>
// FUnction to calculate power
 long long power(double x, int n){
     long long BN = 1;// long long
     for(int i = 0; i<n;++i){
         BN*=x;
     }
     return BN;
 }
```

In this example, the code does not call any of the functions defined in the `iostream` and `string`. These headers are not considered in the analysis. Polyspace calculates this metric to be zero.

Metric Information
Group: Project
Acronym: INCLUDES
HIS Metric: No

See Also
Number of Files|Calculate code metrics (-code-metrics)

Number of Instructions

Number of instructions per function

Description

This metric measures the number of instructions in a function body.

The recommended upper limit for this metric is 50. For more modular code, try to enforce an upper limit for this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Computation Details

The metric is calculated using the following rules:

- A simple statement ending with a `;` is one instruction.

  If the statement is empty, it does not count as an instruction.
- A variable declaration counts as one instruction if both of these conditions are true:

  - The variable is not static.
  - The variable is initialized.
- Control flow statements such as `if`, `for`, `break`, `goto`, `return`, `switch`, `while`, `do-while` count as one instruction.
- The following do not count as instructions by themselves:

  - Beginning of a block of code

    For instance, the following counts as one instruction:

    ```
    {
        var = 1;
    }
    ```
  - Labels

    For instance, the following counts as two instructions. The `case` labels do not count as instructions.

    ```
    switch (1) {  // Instruction 1: switch
        case 0:
        case 1:
        case 2:
        default:
        break;    // Instruction 2: break
     }
    ```

Examples

Calculation of Number of Instructions

```
int func(int* arr, int size) {
    int i, countPos=0, countNeg=0, countZero = 0;
    for(i=0; i<size; i++) {
        if(arr[i] >0)
            countPos++;
        else if(arr[i] ==0)
            countZero++;
        else
            countNeg++;
    }
}
```

In this example, the number of instructions in `func` is 9. The instructions are:

1   `countPos=0`

2   `countNeg=0`

3   `countZero=0`

4   `for(i=0;i<size;i++) { ... }`

5   `if(arr[i] >=0)`

6   `countPos++`

7   `else if(arr[i]==0)`

    The ending `else` is counted as part of the `if-else` instruction.

8   `countZero++`

9   `countNeg++`

Note This metric is different from the number of executable lines. For instance:

- `for(i=0;i<size;i++)` has 1 instruction and 1 executable line.
- The following code has 1 instruction but 3 executable lines.

```
for(i=0;
    i<size;
    i++)
```

Metric Information

Group: Function
Acronym: STMT
HIS Metric: Yes

See Also

`Calculate code metrics (-code-metrics)`

Number of Lines

Total number of lines in a file

Description

This metric calculates the number of lines in a file. When calculating the value of this metric, Polyspace includes comments and blank lines.

This metric is calculated for source files and header files in the same folders as source files. If you want:

- The metric reported for other header files, change the default value of the option Generate results for sources and (-generate-results-for).
- The metric not reported for header files at all, change the value of the option Do not generate results for (-do-not-generate-results-for) to all-headers.

Examples

Code Containing Comments and Blank Lines

```
// Function to calculate power
 long long power(double x, int n){
     long long BN = 1;// long long
     for(int i = 0; i<n;++i){
         BN*=x;
     }
     return BN;
 }
//Function to calculate approximate index
 double AppxIndex(double m, double f){
     double U = (power(m,2) - 1)/(power(m,2)+2); //First term
     double V = (power(m,4) + 27*power(m,2)+38)/
     (2*power(m,2)+3);// Second term
     return (1+2*f*power(U,2)*(1+power(m,2)*U*V +
      power(m,3)/power(m,3)*(U-V)))
     /( (1-2*f*power(U,2)*(1+power(m,2)*U*V
     + power(m,3)/power(m,3)*(U-V))));
 }
```

Because Polyspace includes comments and black lines when calculating this metric, the total number of line in this file is 18.

Metric Information
Group: File
Acronym: TOTAL_LINES
HIS Metric: No

See Also
Number of Lines Without Comment | Calculate code metrics (-code-metrics)

Number of Lines Within Body

Number of lines in function body

Description

This metric calculates the number of lines in function body. When calculating the value of this metric, Polyspace includes declarations, comments, blank lines, braces and preprocessing directives.

If the function body contains a `#include` directive, the included file source code is also calculated as part of this metric.

This metric is not calculated for C++ templates.

Examples

Function with Declarations, Braces and Comments

```
void func(int);


int getSign(int arg) {
    int sign;
    if(arg<0) {
        sign=-1;
        func(-arg);
        /* func takes positive arguments */
    }
    else if(arg==0)
        sign=0;
    else {
        sign=1;
        func(arg);
    }
    return sign;
}
```

In this example, the number of executable lines of `getSign` is 13. The calculation includes:

- The declaration `int sign;`.
- The comment `/* ... */`.
- The two lines with braces only.

Metric Information
Group: Function
Acronym: FLIN
HIS Metric: No

See Also
Number of Executable Lines | Calculate code metrics (-code-metrics)

Number of Lines Without Comment

Number of lines of code excluding lines that are comments or blank

Description

This metric calculates the number of executable lines in a file. When calculating the value of this metric, Polyspace excludes lines that are comments or blank.

This metric is calculated for source files and header files in the same folders as source files. If you want:

- The metric reported for other header files, change the default value of the option `Generate results for sources and (-generate-results-for)`.

- The metric not reported for header files at all, change the value of the option `Do not generate results for (-do-not-generate-results-for)` to `all-headers`.

Examples

Inline Comments

```
// Function to calculate power
 long long power(double x, int n){
     long long BN = 1;// long long
     for(int i = 0; i<n;++i){
         BN*=x;
     }
     return BN;
 }
//Function to calculate approximate index
 double AppxIndex(double m, double f){
     double U = (power(m,2) - 1)/(power(m,2)+2); //First term
     double V = (power(m,4) + 27*power(m,2)+38)/
     (2*power(m,2)+3);// Second term
     return (1+2*f*power(U,2)*(1+power(m,2)*U*V +
      power(m,3)/power(m,3)*(U-V)))
     /( (1-2*f*power(U,2)*(1+power(m,2)*U*V
     + power(m,3)/power(m,3)*(U-V))));
 }
```

In this example, Polyspace calculates the number of lines in the file that are neither comment nor blank lines. Ignoring lines that are comments, such as the line `// FUnction to calculate power`, and blank lines, Polyspace evaluates the metric as `16`. When evaluating this metric, the lines of code that contain inline comments are counted as lines of code.

Metric Information
Group: File
Acronym: LINES_WITHOUT_CMT
HIS Metric: No

See Also
Number of Lines | Calculate code metrics (-code-metrics)

Number of Local Non-Static Variables

Total number of local variables in function

Description

This metric provides the number of local variables in a function.

The metric excludes static variables. To find number of static variables, use the metric `Number of Local Static Variables`.

Examples

Non-Structured Variables

```
int flag();


int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, the number of local non-static variables in `func` is 5. The number does not include the function arguments and return value.

Arrays and Structured Variables

```
typedef struct myStruct{
    char  arr1[50];
    char  arr2[50];
    int   val;
} myStruct;


void func(void) {
  myStruct var;
  char localArr[50];
}
```

In this example, the number of local non-static variables in `func` is 2: the structured variable `var` and the array `localArr`.

Variables in Class Methods

```
class Rectangle {
    int width, height;
```

```
  public:
    void set (int,int);
    int area (void);
} rect;

int Rectangle::area (void) {
    int temp;
    temp = width * height;
    return(temp);
}
```

In this example, the number of local non-static variables in `Rectangle::area` is 1: the variable `temp`.

Metric Information
Group: Function
Acronym: LOCAL_VARS
HIS Metric: No

See Also
`Number of Local Static Variables` | `Higher Estimate of Local Variable Size` | `Lower Estimate of Local Variable Size` | `Calculate code metrics (-code-metrics)`

Introduced in R2017a

Number of Local Static Variables

Total number of local static variables in function

Description

This metric provides the number of local static variables in a function.

Examples

Number of Static Variables

```
void func(void) {
  static int var_1 = 0;
  int var_2;
}
```

In this example, the number of static variables in func is 1. For examples of different types of variables, see Number of Local Non-Static Variables.

Metric Information

Group: Function
Acronym: LOCAL_STATIC_VARS
HIS Metric: No

See Also

Higher Estimate of Local Variable Size|Number of Local Non-Static Variables|
Lower Estimate of Local Variable Size|Calculate code metrics (-code-metrics)

Introduced in R2017a

Number of Paths

Estimated static path count

Description

This metric measures the number of paths in a function.

The recommended upper limit for this metric is 80. If the number of paths is high, the code is difficult to read and can cause more orange checks. Try to limit the value of this metric.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Computation Details

The number of paths is calculated according to these rules:

- If the statements in a function do not break the control flow, the number of paths is one.

  Even an empty statement such as `;` or empty block such as `{}` counts as one path.
- A control flow statement introduces branches and adds to the original one path.

  - `if-else if-else`: Each `if` keyword introduces a new branch. The contribution from an `if-else if-else` block is the number of branches plus one (the original path). If a catch-all `else` is present, all paths go through the block; otherwise, one path bypasses the block.

    For instance, a function with an `if(..) {} else if(..) {} else {}` statement has three paths. A function with one `if() {}` only has two paths, one that goes through the `if` block and one that bypasses the block.
  - `switch-case`: Each `case` label introduces a new branch. The contribution from a `switch` block is the number of `case` labels plus one (the original path). If a catch-all `default` is present, all paths go through the block; otherwise, one path bypasses the block.

    For instance, a function with a statement `switch (var) { case 1: .. break; case 2: .. break; default: .. }` has three paths, all going through the `switch` block. If you omit the `default`, the function still has three paths, two going through the `switch` block and one bypassing the block.
  - `for` and `while`: Each loop statement introduces a new branch. The contribution from a loop is two - a path that goes through the loop and a path that bypasses the loop.
  - `do-while`: Each `do-while` statement introduces a new branch except when the condition of the `while` statement is explicitly false. Statements written as `do{/*..*/}while(0)` do not function as loops. Such statements are often used for enclosing multiple lines of macros within braces. For instance, this `do-while` statement serves to encapsulate the multiline macro rather than create a new path:

```
#define myfunc(x) do{ ...\\
                      ...\\
                      ...}while(0);
```

Polyspace considers such statements to be a single path.

Note that a statement with a ternary operator such as

```
result = a > b ? a : b;
```

is not considered as a statement that breaks the control flow.

- If more than one control flow statement are present in a sequence without any nesting, the number of paths is the product of the contributions from each control flow statement.

  For instance, if a function has three `for` loops and two `if-else` blocks, one after another, the number of paths is $2 \times 2 \times 2 \times 2 \times 2 = 32$.

  If many control flow statements are present in a function, the number of paths can be large. Nested control flow statements reduce the number of paths at the cost of increasing the depth of nesting. For an example, see "Function with Nested Control Flow Statements" on page 8-64.

- The software displays specific values in cases where the metric is not calculated:

  - If `goto` statements are present in the body of the function, Polyspace cannot calculate the number of paths and shows the number of paths as `Not Computed` instead.

  - If the number of paths reaches an upper limit of 1,000,000,000, Polyspace stops the calculation and displays just the upper limit. The actual value might be higher.

Examples

Function with One Path

```
int func(int ch) {
    return (ch * 2);
}
```

In this example, `func` has one path.

Function with Control Flow Statement Causing Multiple Paths

```
void func(int ch) {
    switch (ch)
    {
    case 1:
        break;
    case 2:
        break;
    case 3:
        break;
    case 4:
        break;
    default:
    }
}
```

In this example, `func` has five paths. Apart from the path that goes through the `default`, each `case` label followed by a statement causes the creation of a new path.

Function with Nested Control Flow Statements

```
void func()
{
    int i = 0, j = 0, k = 0;
    for (i=0; i<10; i++)
    {
        for (j=0; j<10; j++)
        {
            for (k=0; k<10; k++)
            {
                if (i < 2 )
                    ;
                else
                {
                    if (i > 5)
                        ;
                    else
                        ;
                }
            }
        }
    }
}
```

In this example, `func` has six paths: three from the `for` statements, two from the `if` statements plus the original path that bypasses all control flow statements.

Metric Information
Group: Function
Acronym: PATH
HIS Metric: Yes

See Also
Calculate code metrics (-code-metrics)

Number of Potentially Unprotected Shared Variables

Number of unprotected shared variables

Description

This metric measures the number of variables with the following properties:

- The variable is used in more than one task.
- At least one operation on the variable is not protected from interruption by operations in other tasks.

Examples

Unprotected Shared Variables

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}
```

In this example, Polyspace calculates the number of potentially unprotected shared variable to be one. The shared variabel `shared_var` is unprotected if you specify `task` and `interrupt_handler` as entry points and do not specify protection mechanisms.

The operation `shared_var = INT_MAX` can interrupt the other operations on `shared_var` and cause unpredictable behavior.

Metric Information
Group: Project
Acronym: UNPSHV
HIS Metric: No

See Also
Calculate code metrics (-code-metrics)

Introduced in R2018b

Number of Protected Shared Variables

Number of protected shared variables

Description

This metric measures the number of variables with the following properties:

- The variable is used in more than one task.
- All operations on the variable are protected from interruption through critical sections or temporal exclusions.

Examples

Shared Variables Protected Through Temporal Exclusion

```c
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following options:

Option	Value
Tasks (-entry-points)	task interrupt_handler
Temporally exclusive tasks (-temporal-exclusions-file)	temporal_exclusion.txt

In the command line, specify these options to run the example:

```
-entry-points task,interrupt_handler  -temporal-exclusions-file temporal_exclusion.txt
```

temporal_exclusion.txt is a text file containing the temporally exclusive tasks separated by a space in a single line:

```
task interrupt_handler
```

The variable is shared between task and interrupt_handler. However, because task and interrupt_handler are temporally exclusive, operations on the variable cannot interrupt each other.

Shared Variables Protected Through Critical Sections

```c
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void take_semaphore(void);
void give_semaphore(void);

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        reset();
        inc();
        inc();
        give_semaphore();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
```

```
    while(randomValue) {
        take_semaphore();
        interrupt();
        give_semaphore();
    }
}

void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following:

Option	Value	

Entry points	`task`
	
	`interrupt_handler`	

Critical section details	**Starting routine**	**Ending routine**
	`take_semaphore`	`give_semaphore`

In the command line, specify these options to run the example:

```
-critical-section-begin  take_semaphore:cs1
-critical-section-end give_semaphore:cs1
-entry-points task,interrupt_handle
```

The variable is shared between `task` and `interrupt_handler`. However, because operations on the variable are between calls to the starting and ending procedure of the same critical section, they cannot interrupt each other.

Metric Information

Group: Project
Acronym: PSHV
HIS Metric: No

See Also

Calculate code metrics (-code-metrics)|Critical section details (-critical-section-begin -critical-section-end)|Tasks (-entry-points)|Temporally exclusive tasks (-temporal-exclusions-file)

Introduced in R2018b

Number of Recursions

Number of call graph cycles over one or more functions

Description

The metric provides a quantitative estimate of the number of recursion cycles in your project. The metric is the sum of:

- Number of direct recursions (self recursive functions or functions calling themselves).
- Number of strongly connected components formed by the indirect recursion cycles in your project. If you consider the recursion cycles as a directed graph, the graph is strongly connected if there is a path between all pairs of vertices.

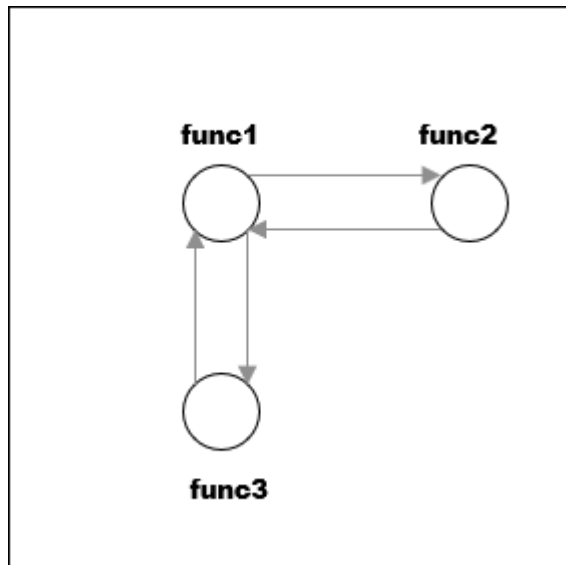 To compute the number of strongly connected components:

  1 Draw the recursion cycles in your code.

  For instance, the recursion cycles in this example are shown below.

  ```
  volatile int checkStatus;
  void func1() {
     if(checkStatus) {
          func2();
     }
     else {
          func3();
     }
  }

  func2() {
     func1();
  }

  func3() {
     func1();
  }
  ```

2   Identify the number of strongly connected components formed by the recursion cycles.

In the preceding example, there is one strongly connected component. You can move from any vertex to another vertex by following the paths in the graph.

The event list below the metric shows one of the recursion cycles in the strongly connected component.

Calls through a function pointer are not considered.

The recommended upper limit for this metric is 0. To avoid the possibility of exceeding available stack space, do not use recursions in your code. Recursions can tend to exhaust stack space easily. See examples of stack size growth with recursions described for this CERT-C rule that forbids recursions.

To detect use of recursions, check for violations of one of `MISRA C:2012 Rule 17.2`,MISRA C: 2004 Rule 16.2, `MISRA C++:2008 Rule 7-5-4` or JSF Rule 119. Note that:

• The rule checkers report each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.

• The rule checkers consider explicit function calls only. For instance, in C++ code, the rule checkers ignore implicit calls to constructors during object creation. However, the metrics computation considers both implicit and explicit calls.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Examples

Direct Recursion

```
int getVal(void);
int sum(int val) {
    if(val<0)
        return 0;
    else
        return (val + sum(val-1));
}

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}
```

In this example, the number of recursions is 1.

A direct recursion is a recursion where a function calls itself in its own body. For direct recursions, the number of recursions is equal to the number of recursive functions.

Indirect Recursion with One Call Graph Cycle

```
volatile int signal;
void operation2(void);

void operation1(void) {
    int stop = signal%2;
    if(!stop)
        operation2();
}

void operation2(void) {
    operation1();
}

void main() {
    operation1();
}
```
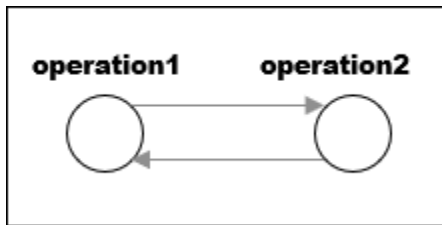
In this example, the number of recursions is one. The two functions `operation1` and `operation2` are involved in the call graph cycle `operation1` → `operation2` → `operation1`.

An indirect function is a recursion where a function calls itself through other functions. For indirect recursions, the number of recursions can be different from the number of recursive functions.

Multiple Call Graph Cycles Forming One Strongly Connected Component

```
volatile int checkStatus;
void func1() {
   if(checkStatus) {
        func2();
   }
   else {
        func3();
   }
}

func2() {
   func1();
}

func3() {
   func1();
}
```

In this example, there are two call graph cycles:

- func1 → func2 → func1
- func1 → func3 → func1

However, the cycles form one strongly connected component. You can move from any vertex to another vertex by following the paths in the graph. Hence, the number of recursions is one.

Indirect Recursion with Two Call Graph Cycles

```
volatile int signal;
void operation1_1();
void operation2_1();

void operation1() {
    int stop = signal%2;
    if(!stop)
        operation1_1();
}


void operation1_1() {
    operation1();
}

void operation2() {
    int stop = signal%2;
    if(!stop)
        operation2_1();
}

void operation2_1() {
    operation2();
}

void main(){
    operation1();
    operation2();
}
```

In this example, the number of recursions is two.

There are two call graph cycles:

- operation1 → operation1_1 → operation1
- operation2 → operation2_1 → operation2

The call graph cycles form two strongly connected components.

Same Function Called in Direct and Indirect Recursion

```
volatile int signal;
void operation2();

void operation1() {
    int stop = signal%3;
    if(stop==1)
        operation1();
    else if(stop==2)
        operation2();
}

void operation2() {
    operation1();
}

void main() {
    operation1();
}
```

In this example, the number of recursions is two:

- The strongly connected component formed by the cycle operation1 → operation2 → operation1.
- The self-recursive function operation1.

Metric Information
Group: Project
Acronym: AP_CG_CYCLE
HIS Metric: Yes

See Also
MISRA C:2012 Rule 17.2 | Calculate code metrics (-code-metrics)

Number of Return Statements

Number of `return` statements in a function

Description

This metric measures the number of `return` statements in a function.

The recommended upper limit for this metric is 1. If one return statement is present, when reading the code, you can easily identify what the function returns.

To enforce limits on metrics:

- In the Polyspace user interface, see "Compute Code Complexity Metrics".
- In the Polyspace Metrics web interface, see "Compare Metrics Against Software Quality Objectives".

Examples

Function with Return Points

```
int getSign (int arg) {
    if(arg <0)
        return -1;
    else if(arg > 0)
        return 1;
    return 0;
}
```

In this example, `getSign` has 3 `return` statements.

Metric Information

Group: Function
Acronym: RETURN
HIS Metric: Yes

See Also

`Calculate code metrics (-code-metrics)`

Topics
"Compute Code Complexity Metrics"
"Compare Metrics Against Software Quality Objectives"

Program Maximum Stack Usage

Maximum stack usage in the analyzed program

Description

This metric is reported in a Code Prover analysis only.

This metric shows the maximum stack usage from your program.

The metric shows the maximum stack usage for the function with the highest stack usage. If you provide a complete application, the function with the highest stack usage is typically the `main` function because the `main` function is at the top of the call hierarchy. For a description of maximum stack usage for a function, see the metric `Maximum Stack Usage`.

Metric Information
Group: Project
Acronym: PROG_MAX_STACK
HIS Metric: No

See Also
`Higher Estimate of Local Variable Size` | `Maximum Stack Usage` | `Program Minimum Stack Usage` | `Calculate code metrics (-code-metrics)`

Topics
"Determination of Program Stack Usage" on page 4-37

Introduced in R2017b

Program Minimum Stack Usage

Maximum stack usage in the analyzed program taking nested scopes into account

Description

This metric is reported in a Code Prover analysis only.

This metric shows the maximum stack usage from your program, taking nested scopes into account.

The metric shows the minimum stack usage for the function with the highest stack usage. If you provide a complete application, the function with the highest stack usage is typically the `main` function because the `main` function is at the top of the call hierarchy. For a description of minimum stack usage for a function, see the metric `Minimum Stack Usage`.

Considering nested scopes is useful for compilers that reuse stack space for variables defined in nested scopes. For instance, in this code, the space for `var_1` is reused for `var_2`.

```
type func (type param_1, ...) {

  {
    /* Scope 1 */
    type var_1, ...;
  }
  {
    /* Scope 2 */
    type var_2, ...;
  }
}
```

Metric Information
Group: Project
Acronym: PROG_MIN_STACK
HIS Metric: No

See Also
Lower Estimate of Local Variable Size | Minimum Stack Usage | Program Maximum Stack Usage | Calculate code metrics (-code-metrics)

Topics
"Determination of Program Stack Usage" on page 4-37

Introduced in R2017b

Custom Coding Rules

Group 1: Files

The custom rules 1.*x* in Polyspace enforce naming conventions for files and folders. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
1.1	All source file names must follow the specified pattern.	Only the base name is checked. A source file is a file that is not included.
1.2	All source folder names must follow the specified pattern.	Only the folder name is checked. A source file is a file that is not included.
1.3	All include file names must follow the specified pattern.	Only the base name is checked. An include file is a file that is included.
1.4	All include folder names must follow the specified pattern.	Only the folder name is checked. An include file is a file that is included.

Group 2: Preprocessing

The custom rules 2.*x* in Polyspace enforce naming conventions for macros. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
2.1	All macros must follow the specified pattern.	Macro names are checked before preprocessing.
2.2	All macro parameters must follow the specified pattern.	Macro parameters are checked before preprocessing.

Group 3: Type definitions

The custom rules 3.*x* in Polyspace enforce naming conventions for fundamental data types. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
3.1	All integer types must follow the specified pattern.	Applies to integer and boolean types specified by `typedef` statements. Does not apply to enumeration types. For example: `typedef signed int int32_t;`
3.2	All float types must follow the specified pattern.	Applies to double and float types specified by `typedef` statements. For example: `typedef float f32_t;`
3.3	All pointer types must follow the specified pattern.	Applies to pointer types specified by `typedef` statements. For example: `typedef int* p_int;`
3.4	All array types must follow the specified pattern.	Applies to array types specified by `typedef` statements. For example: `typedef int a_int_3[3];`
3.5	All function pointer types must follow the specified pattern.	Applies to function pointer types specified by `typedef` statements. For example: `typedef void (*pf_callback) (int);`

Group 4: Structures

The custom rules 4.*x* in Polyspace enforce naming conventions for structured data types. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
4.1	All `struct` tags must follow the specified pattern.	
4.2	All `struct` types must follow the specified pattern.	`struct` types are aliases for previously defined structures (defined with the `typedef` or `using` keyword).
4.3	All `struct` fields must follow the specified pattern.	
4.4	All `struct` bit fields must follow the specified pattern.	

Group 5: Classes (C++)

The custom rules 5.*x* in Polyspace enforce naming conventions for classes and class members. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
5.1	All class names must follow the specified pattern.	
5.2	All class types must follow the specified pattern.	Class types are aliases for previously defined classes (defined with the `typedef` or `using` keyword).
5.3	All data members must follow the specified pattern.	
5.4	All function members must follow the specified pattern.	
5.5	All static data members must follow the specified pattern.	
5.6	All static function members must follow the specified pattern.	
5.7	All bitfield members must follow the specified pattern.	

Group 6: Enumerations

The custom rules 6.*x* in Polyspace enforce naming conventions for enumerations. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
6.1	All enumeration tags must follow the specified pattern.	
6.2	All enumeration types must follow the specified pattern.	Enumeration types are aliases for previously defined enumerations (defined with the `typedef` or `using` keyword).
6.3	All enumeration constants must follow the specified pattern.	

Group 7: Functions

The custom rules 7.*x* in Polyspace enforce naming conventions for functions and function parameters. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
7.1	All global functions must follow the specified pattern.	A global function is a function with external linkage.
7.2	All static functions must follow the specified pattern.	A static function is a function with internal linkage. This rule does not apply to class member functions.
7.3	All function parameters must follow the specified pattern.	In C++, applies to non-member functions.

Group 8: Constants

The custom rules 8.*x* in Polyspace enforce naming conventions for constants. For information on how to enable these rules, see `Check custom rules (-custom-rules)`. These rules do not apply to constants that are defined within a class or to constants that are function parameters.

Number	Rule Applied	Other details
8.1	All global nonstatic constants must follow the specified pattern.	A global nonstatic constant is a constant with external linkage.
8.2	All global static constants must follow the specified pattern.	A global static constant is a constant with internal linkage.
8.3	All local nonstatic constants must follow the specified pattern.	A local nonstatic constant is a constant without linkage.
8.4	All local static constants must follow the specified pattern.	A local static constant is a constant declared static in a function.

Group 9: Variables

The custom rules 9.*x* in Polyspace enforce naming conventions for variables. For information on how to enable these rules, see `Check custom rules (-custom-rules)`. These rules do not apply to variables that are defined within a class or to variables that are function parameters.

Number	Rule Applied	Other details
9.1	All global nonstatic variables must follow the specified pattern.	A global nonstatic variable is a variable with external linkage.
9.2	All global static variables must follow the specified pattern.	A global static variable is a variable with internal linkage.
9.3	All local nonstatic variables must follow the specified pattern.	A local nonstatic variable is a variable without linkage.
9.4	All local static variables must follow the specified pattern.	A local static variable is a variable declared static in a function.

Group 10: Name spaces (C++)

The custom rules 10.*x* in Polyspace enforce naming conventions for namespaces. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied
10.1	All names spaces must follow the specified pattern.

Group 11: Class templates (C++)

The custom rules 11.*x* in Polyspace enforce naming conventions for class templates. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
11.1	All class templates must follow the specified pattern.	
11.2	All class template parameters must follow the specified pattern.	

Group 12: Function templates (C++)

The custom rules 12.*x* in Polyspace enforce naming conventions for function templates. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details
12.1	All function templates must follow the specified pattern.	Applies to nonmember function templates.
12.2	All function template parameters must follow the specified pattern.	Applies to member and nonmember function templates.
12.3	All function template members must follow the specified pattern.	

Group 20: Style

The custom rules 20.*x* in Polyspace enforce coding style conventions such as number of characters per line. For information on how to enable these rules, see `Check custom rules (-custom-rules)`.

Number	Rule Applied	Other details

20.1	Source line length must not exceed specified number of characters.	When configuring the checker, specify:
		
		• A number for the character limit. Use the **Pattern** column on the configuration or the `pattern=` line in the custom rules text file.
		• A violation message such as:
		
		  `Line exceeds` *n* `characters.`
		
		  Use the **Convention** column on the configuration or the `convention=` line in the custom rules xml file.

Global Variables

Potentially unprotected variable

Global variables shared between multiple tasks but not protected from concurrent access by the tasks

Description

A **shared unprotected global variable** has the following properties:

- The variable is used in more than one task.
- Polyspace determines that at least one operation on the variable is not protected from interruption by operations in other tasks.

In code that is not intended for multitasking, all global variables are non-shared.

In your verification results, these variables are colored orange on the **Source**, **Results List** and **Variable Access** panes. On the **Source** pane, the coloring is applied to the variable only during declaration.

Examples

Unprotected Shared Variables

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}
```
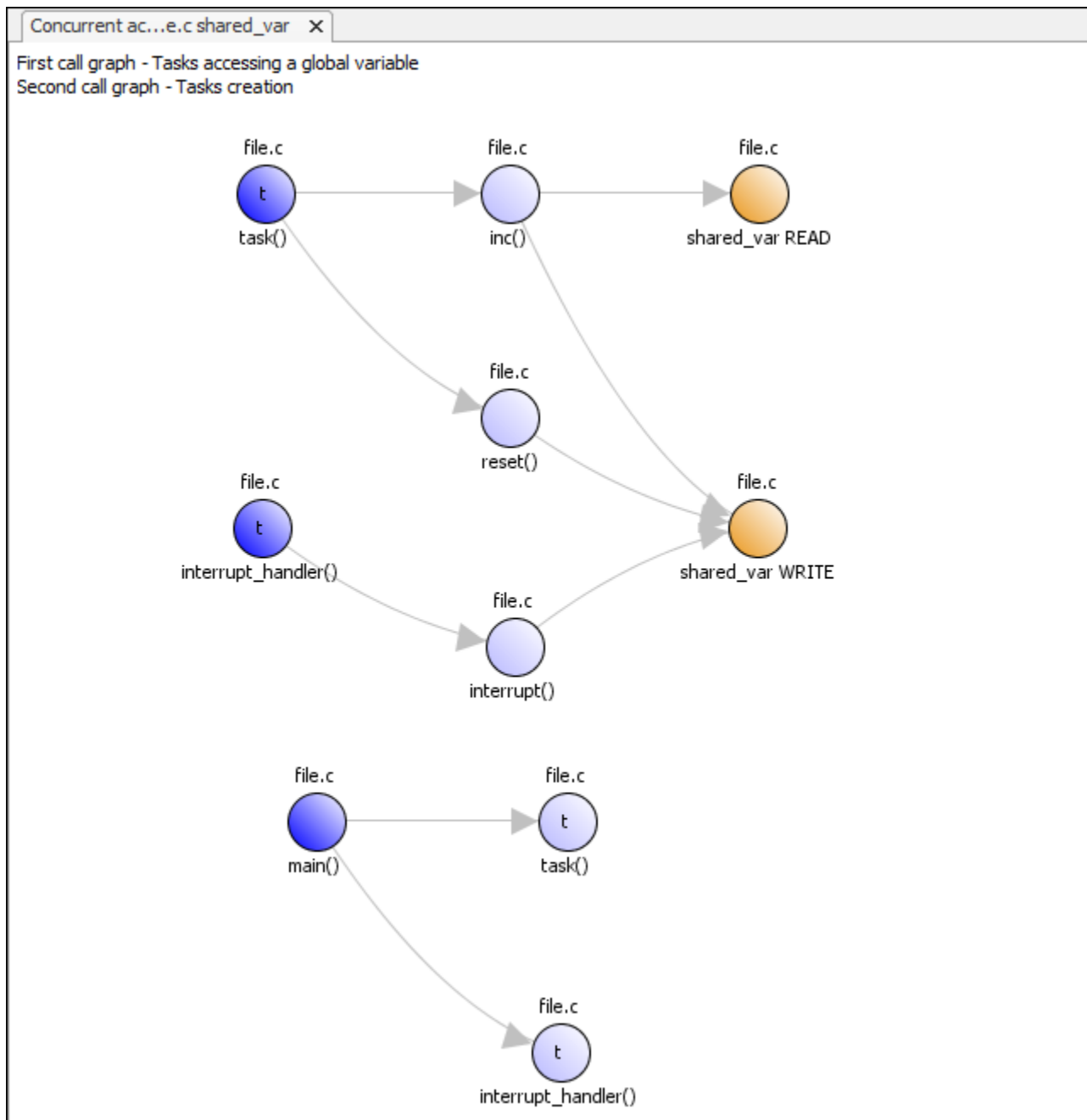
```
void main() {
}
```

In this example, `shared_var` is an unprotected shared variable if you specify the following multitasking options:

Option	Value
Configure multitasking manually on page 1-116	☑

Tasks on page 1-118	`task`
	
	`interrupt_handler`

You do not specify protection mechanisms such as critical sections.

The operation `shared_var = INT_MAX` can interrupt the other operations on `shared_var` and cause unpredictable behavior.

If you click the ⬚ (graph) icon on the **Result Details** pane, you see the two concurrent tasks (threads).

The first graph shows how the tasks access the variable. For instance, the task `interrupt_handler` calls a function `interrupt` that writes to the shared variable `shared_var`.

The second graph shows how the tasks are created. In this example, both tasks are created after `main` completes. In other cases, tasks might be created within functions called from `main`.

Check Information

Language: C | C++

See Also

Critical section details (-critical-section-begin -critical-section-end) | Multitasking | Shared variable | Show global variable sharing and usage only (-shared-variables-mode) | Tasks (-entry-points) | Temporally exclusive tasks (-temporal-exclusions-file) | Unused variable | Used non-shared variable

Topics
"Analyze Multitasking Programs in Polyspace"
"Protections for Shared Variables in Multitasking Code"

Shared variable

Global variables shared between multiple tasks and protected from concurrent access by the tasks

Description

A **shared protected global variable** has the following properties:

- The variable is used in more than one task.
- All operations on the variable are protected from interruption through critical sections or temporal exclusion. The calls to functions beginning and ending a critical section must be reachable.

In code that is not intended for multitasking, all global variables are non-shared.

In your verification results, these variables are colored green on the **Source**, **Results List** and **Variable Access** panes. On the **Source** pane, the coloring is applied to the variable only during declaration.

Examples

Shared Variables Protected Through Temporal Exclusion

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}
```

```
void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following multitasking options:

Option	Value
Configure multitasking manually on page 1-116	☑

Tasks on page 1-118	`task`
	
	`interrupt_handler`

Temporally exclusive tasks on page 1-129	`task interrupt_handler`

On the command-line, you can use the following:

```
 polyspace-code-prover
   -entry-points task,interrupt_handler
   -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task interrupt_handler
```

The variable is shared between `task` and `interrupt_handler`. However, because `task` and `interrupt_handler` are temporally exclusive, operations on the variable cannot interrupt each other.

Shared Variables Protected Through Critical Sections

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void take_semaphore(void);
void give_semaphore(void);

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        reset();
        inc();
        inc();
        give_semaphore();
    }
}
```

```
void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        interrupt();
        give_semaphore();
    }
}

void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following:

Option	Value	
Configure multitasking manually on page 1-116	☑	

Tasks on page 1-118	`task`
	
	`interrupt_handler`	

Critical section details on page 1-126	**Starting routine**	**Ending routine**
	`take_semaphore`	`give_semaphore`

On the command-line, you can use the following:

```
 polyspace-code-prover
   -entry-points task,interrupt_handler
   -critical-section-begin take_semaphore:cs1
   -critical-section-end give_semaphore:cs1
```

The variable is shared between `task` and `interrupt_handler`. However, because operations on the variable are between calls to the starting and ending procedure of the same critical section, they cannot interrupt each other.

Shared Structure Variables Protected Through Access Pattern

```
struct S {
    unsigned int var_1;
    unsigned int var_2;
};

volatile int randomVal;

struct S sharedStruct;

void task1(void) {
    while(randomVal)
        operation1();
}
```

```
void task2(void) {
    while(randomVal)
        operation2();
}

void operation1(void) {
        sharedStruct.var_1++;
}

void operation2(void) {
        sharedStruct.var_2++;
}

int main(void) {
    return 0;
}
```

In this example, `sharedStruct` is a protected shared variable if you specify the following:

Option	Value
Configure multitasking manually on page 1-116	☑
Tasks on page 1-118	task1 task2

On the command-line, you can use the following:

```
 polyspace-code-prover
    -entry-points task1,task2
```

The software determines that `sharedStruct` is protected because:

- `task1` operates only on `sharedStruct.var_1`.
- `task2` operates only on `sharedStruct.var_2`.

If you select the result, the **Result Details** pane indicates that the access pattern protects all operations on the variable. On the **Variable Access** pane, the row for variable `sharedStruct` lists `Access pattern` as the protection type.

Shared Variables Protected Through Design Pattern and Mutex

```
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t lock;
pthread_t id1, id2;

int var;

void * t1(void* b) {
    pthread_mutex_lock(&lock);
    var++;
    pthread_mutex_unlock(&lock);
}
```

```
void * t2(void* a) {
    pthread_mutex_lock(&lock);
    var = 1;
    pthread_mutex_unlock(&lock);
}

int main(void) {
    pthread_create(&id1, NULL, t1, NULL);
    pthread_create(&id2, NULL, t2, NULL);

    return 0;
}
```

`var` is a shared, protected variable if you specify the following options:

Option Name	Value
Enable automatic concurrency detection on page 1-105	☑

On the command-line, you can use the following:

```
 polyspace-code-prover
    -enable-concurrency-detection
```

In this example, if you specify the concurrency detection option, Polyspace Code Prover detects that your program uses multitasking. Two task, `lock` and `var`, share two variables. `lock` is a pthread mutex variable, which `pthread_mutex_lock` and `pthread_mutex_unlock` use to lock and unlock their mutexes. The inherent pthread design patterns protect `lock`. The **Results Details** pane and **Variable Access** pane list `Design Pattern` as the protection type.

The mutex locking and unlocking mechanisms protect `var`, the other shared variable. The **Results Details** pane and **Variable Access** pane list `Mutex` as the protection type.

Check Information
Language: C | C++

See Also
`Critical section details (-critical-section-begin -critical-section-end)` | `Potentially unprotected variable` | `Show global variable sharing and usage only (-shared-variables-mode)` | `Tasks (-entry-points)` | `Temporally exclusive tasks (-temporal-exclusions-file)` | `Unused variable` | `Used non-shared variable`

Topics
"Analyze Multitasking Programs in Polyspace"
"Protections for Shared Variables in Multitasking Code"

Non-shared unused global variable

Global variables declared but not used

Description

A **non-shared unused** global variable has the following properties:

- The variable is declared in the code.
- Polyspace cannot detect a read or write operation on the variable.

In your verification results, these variables are colored gray on the **Source**, **Results List** and **Variable Access** panes. On the **Source** pane, the coloring is applied to the variable only during declaration. In the **Result Details** pane, the variable name appears along with the name of the file where it is defined (for `extern` variables where the definition is not available, `?extern` is used for file name.)

Note The software does not display a complete list of unused global variables. Especially, in C++ projects, unused global variables can be suppressed from display.

Examples

Used and Unused Global Variables

```
int var1;
int var2;
int var3;
int var4;

int input(void);

void main() {
    int loc_var = input(), flag=0;

    var1 = loc_var;
    if(0) {
        var3 = loc_var;
    }
    if(flag!=0) {
        var4 =loc_var;
    }


}
```

If you verify the above code in a C project, the software lists `var2`, `var3` and `var4` as non-shared unused variables, and `var1` as a non-shared used variable.

`var3` and `var4` are used in unreachable code and are therefore marked as unused.

Note In a C++ project, the software does not list the unused variable `var2`.

Check Information
Language: C | C++

See Also
Potentially unprotected variable | Shared variable | Show global variable sharing and usage only (-shared-variables-mode) | Used non-shared variable

Topics
"Interpret Code Prover Results in Polyspace Desktop User Interface"

Used non-shared variable

Global variables used in a single task

Description

A **non-shared used** global variable has the following properties:

- The variable is used only in a single task.
- Polyspace detects at least one read or write operation on the variable.

In code that is not intended for multitasking, all global variables are non-shared.

In your verification results, these variables are colored black on the **Results List** and **Variable Access** panes.

Examples

Used and Unused Global Variables

```
int var1;
int var2;
int var3;
int var4;

int input(void);

void main() {
    int loc_var = input(), flag=0;

    var1 = loc_var;
    if(0) {
        var3 = loc_var;
    }
    if(flag!=0) {
        var4 =loc_var;
    }


}
```

If you verify the above code in a C project, the software lists `var2`, `var3` and `var4` as non-shared unused variables, and `var1` as a non-shared used variable.

`var3` and `var4` are used in unreachable code and are therefore marked as unused.

Note In a C++ project, the software does not list the unused variable `var2`.

Non-shared variables in multitasking code

```
unsigned int var_1;
unsigned int var_2;
```

```
volatile int randomVal;

void task1(void) {
    while(randomVal)
        operation(1);
}

void task2(void) {
    while(randomVal)
        operation(2);
}

void operation(int i) {
    if(i==1) {
        var_1++;
    }
    else    {
        var_2++;
    }
}

int main(void) {
    return 0;
}
```

In this example, even when you specify `task1` and `task2` for the option `Tasks (-entry-points)`, the software determines that `var_1` and `var_2` are non-shared.

Even though both `task1` and `task2` call the function `operation`, because of the `if` statement in `operation`, `task1` can operate only on `var_1` and `task2` only on `var_2`.

Check Information
Language: C | C++

See Also
Potentially unprotected variable | Shared variable | Show global variable sharing and usage only (-shared-variables-mode) | Unused variable

Report Components

Acronym Definitions

Create table of Polyspace acronyms used in report and their full forms

Description

This component creates a table containing the acronyms used in the report and their full forms. Acronyms are used for Polyspace checks and result status.

See Also

Topics
"Customize Existing Code Prover Report Template"

Call Hierarchy

Create table showing call graph in source code

Description

This component creates a table showing the call hierarchy in your source code. For each function call in your source code, the table displays the following information:

- Level of call hierarchy, where the function is called.

  Each level is denoted by |. If a function call appears in the table as |||->
  file_name.*function_name*, the function call occurs at the third level of the hierarchy. Beginning from `main` or an entry point, there are three function calls leading to the current call.

- File containing the function call.

  In Code Prover, the line and column is also displayed.

- File containing the function definition.

  In Code Prover, the line and column where the function definition begins is also displayed.

In addition, the table also displays uncalled functions.

This table captures the information available on the **Call Hierarchy** pane in the Polyspace user interface.

See Also

Topics
"Customize Existing Code Prover Report Template"

Code and Verification Information

Create table of verification times and code characteristics

Description

This component creates tables containing verification times and code characteristics such as number of lines.

Properties

Include Verification Time Information

If you select this option, the report contains verification times broken down by phase.

- For Polyspace Bug Finder, the phases are `compilation`, `pass0`, `pass1`, etc.
- For Polyspace Code Prover, the phases are `compilation`, `global`, `function`, etc.

Include Code Details

If you select this option, the report contains the following code characteristics:

- Number of files
- Number of lines
- Number of lines without comment

See Also

Topics
"Customize Existing Code Prover Report Template"

Code Metrics Details

Create table of Polyspace metrics broken down by file and function

Description

This component creates a table containing metrics from a Polyspace project. The metrics appear broken down by file and function.

Properties

Project Metrics

If you select this option, the report contains the following metrics about the project:

- Number of direct recursions
- Number of files
- Number of headers
- Number of protected and unprotected shared variables

File Metrics

If you select this option, the report contains the following metrics about each file in the project:

- Estimated function coupling
- Lines without comment
- Comment density
- Total lines

Function Metrics

If you select this option, the report contains the following metrics about each function in the project:

- Cyclomatic complexity
- Language scope
- Lower and higher estimates of local variable size
- Number of lines within body
- Number of executable lines
- Number of `goto` statements
- Number of call levels
- Number of called functions
- Number of call occurrences
- Number of function parameters
- Number of paths
- Number of `return` statements

- Number of instructions
- Number of calling functions

See Also

Topics
"Customize Existing Code Prover Report Template"

Code Metrics Summary

Create table of Polyspace metrics

Description

This component creates a table containing metrics from a Polyspace project. The metrics are the same as those displayed under `Code Metrics Details`. However, the file and function metrics are not broken down by individual files and functions. Instead, the table provides the minimum and maximum value of a file metric over all files and a function metric over all functions.

See Also

Topics
"Customize Existing Code Prover Report Template"

Code Verification Summary

Create table of Polyspace analysis results

Description

This component creates tables containing the following results:

- Number of results
- Number of coding rule violations for each coding rule type such as MISRA C
- Number of defects, for Polyspace Bug Finder results
- Number of checks of each color, for Polyspace Code Prover results
- Whether the project passed or failed the software quality objective

Properties

Include Checks from Polyspace Standard Library Stub Functions

Unless you deselect this option, the tables contain Polyspace Code Prover checks that appear in Polyspace stubs for the standard library functions.

See Also

Topics
"Customize Existing Code Prover Report Template"

Coding Rules Details

Create table of coding rule violations broken down by file

Description

This component creates tables containing coding rule violations broken down by each file in the Polyspace project. For each rule violation, the table contains the following information:

- Rule number
- Rule description
- Function containing the violation
- (Code Prover only) Line and column number
- Review information such as classification, status and comments

Properties

Select Coding Rules Type

Using this option, you can choose which coding rule violations to display. You can display violations for the following set of coding rules:

- MISRA C rules
- MISRA AC AGC rules
- MISRA C++ rules
- JSF C++ rules
- Custom coding rules

Display by

Using this option, you can break down the display of coding rule violations by file.

See Also

Topics
"Customize Existing Code Prover Report Template"

Coding Rules Summary

Create table with number of coding rule violations

Description

This component creates a table containing the number of coding rule violations. You can choose whether to break this information down by rule number or file.

Properties

Select Coding Rules Type

Using this option, you can choose which coding rule violations to display. You can display violations for the following set of coding rules:

- MISRA C rules
- MISRA AC AGC rules
- MISRA C++ rules
- JSF C++ rules
- Custom coding rules

Include Files/Rules with No Problems Detected

If you select this option, the table displays:

- Files that do not contain coding rule violations
- Rules that your code does not violate

Display by

Using this option, you can break down the display of coding rule violations by:

- Rule number
- File

See Also

Topics
"Customize Existing Code Prover Report Template"

Configuration Parameters

Create table of analysis options, assumptions and coding rules configuration

Description

This component creates the following tables:

- *Polyspace settings*: The analysis options that you used to obtain your results. The table lists command-line version of the options along with their values.
- *Analysis assumptions*: The assumptions used to obtain your Code Prover results. The table lists only the modifiable assumptions. For assumptions that you cannot change, see the Polyspace documentation.
- *Coding rules configuration*: The coding rules whose violations you checked for. The table lists the rule number, rule description and other information about the rules.
- *Files with compilation errors*: If your project has source files with compilation errors, these files are listed.

See Also

Topics
"Customize Existing Code Prover Report Template"

Defects Summary

Create table of defects (Bug Finder only)

Description

This component creates a table of Polyspace Bug Finder defects. From this table, you can see the number of defects of each type.

Properties

Include Checkers with No Defects Detected

If you select this option, the table includes all defect types that Polyspace Bug Finder can detect, including those that do not occur in your code.

See Also

Topics
"Customize Existing Code Prover Report Template"

Global Variable Checks

Create table of global variables (Code Prover only)

Description

This component creates a table of Polyspace Code Prover global variables. From this table, you can see the number of global variables of each type.

See Also

Topics
"Customize Existing Code Prover Report Template"

Recursive Functions

Create table of recursive functions

Description

This component creates a table containing the recursive functions in your source code (along with the files containing the functions).

- For each direct recursion (function calling itself directly), the table lists the recursive function.
- For each indirect recursion cycle (function calling itself through other functions), the table lists one function in the cycle.

For instance, the following code contains two indirect recursion cycles.

```
volatile int signal;

void operation1() {
    int stop = signal%2;
    if(!stop)
        operation1_1();
}


void operation1_1() {
    operation1();
}

void operation2() {
    int stop = signal%2;
    if(!stop)
        operation2_1();
}

void operation2_1() {
    operation2();
}

void main(){
    operation1();
    operation2();
}
```
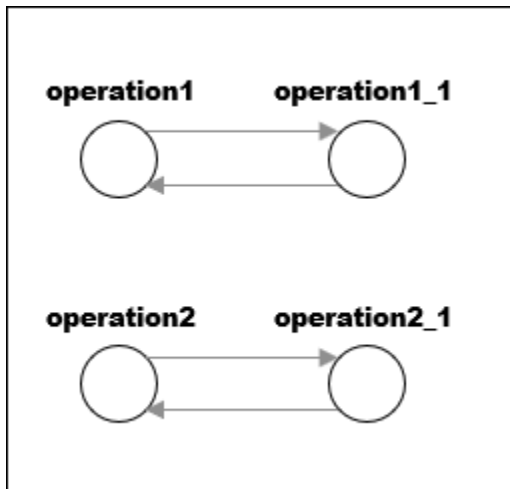
The two call graph cycles are:

- operation1 → operation1_1 → operation1
- operation2 → operation2_1 → operation2

This report component shows one function from each of the two cycles: `operation1` and `operation2`. To see the full cycle, open the results in the Polyspace user interface.

See Also

Topics
"Customize Existing Code Prover Report Template"

Report Customization (Filtering)

Create filters that apply to your Polyspace reports

Description

This component allows you to filter unwanted information from existing Polyspace report templates. To apply global filters, place this component immediately below the node representing the report name.

Properties

Code Metrics Filters

The properties in table below apply to the inclusion of code metrics in your report.

Property	Purpose	User Action
Include Project Metrics	Choose whether to include metrics about your Polyspace project.	Select the check box to include project metrics.
Project metrics to include	Specify project metrics to include or exclude from report.	Enter a MATLAB regular expression.
Include File Metrics	Choose whether to include per file metrics in report.	Select the check box to include per file metrics.
File Metrics > Files to include	Specify files to include or exclude when reporting file metrics.	Enter a MATLAB regular expression.
File metrics to include	Specify file metrics to include or exclude from report.	Enter a MATLAB regular expression.
Include Function Metrics	Choose whether to include per function metrics in report.	Select the check box to include per function metrics.
Function Metrics > Files to include	Specify files to include or exclude when reporting function metrics.	Enter a MATLAB regular expression.
Functions to include	Specify functions to include or exclude when reporting function metrics.	Enter a MATLAB regular expression.
Function metrics to include	Specify function metrics to include or exclude from report.	Enter a MATLAB regular expression.

Coding Rules Filters

The properties in table below apply to the inclusion of coding rule violations in your report.

Property	Purpose	User Action
Files to include	Specify files to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.
Coding rule numbers to include	Specify coding rules to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.
Classifications to include	Specify classifications to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.
Status types to include	Specify statuses to include or exclude when reporting coding rule violations.	Enter a MATLAB regular expression.

Run-time Check Filters

The properties in table below apply to the inclusion of Polyspace Code Prover checks in your report.

Property	Purpose
Red Checks	Specify whether to include red checks in your report. Red checks indicate proven run-time errors.
Gray Checks	Specify whether to include gray checks in your report. Gray checks indicate unreachable code.
Orange Checks	Specify whether to include orange checks in your report. Orange checks indicate possible run-time errors.
Green Checks	Specify whether to include green checks in your report. Green checks indicate that an operation does not contain a specific run-time error.
Inspection Point Checks	Specify whether to include inspection point checks in your report. These checks allow an user to find the values that a variable can take at a certain point in the code.
Unreachable Functions	Specify whether to include unreachable functions in your report.

Advanced Filters

The properties in table below apply to the inclusion of metrics, coding rule violations and Polyspace Code Prover checks in your report.

Property	Purpose	User Action
Justification status	Choose whether to report only justified checks, only unjustified checks or all checks.	Choose an option from the dropdown list.

11-17

Property	Purpose	User Action
Files to include	Specify files to include or exclude from your report.	Enter a MATLAB regular expression.
Check types to include	Specify Polyspace Code Prover checks to include in your report.	Enter a MATLAB regular expression.
Function names to include	Specify functions to include or exclude from your report.	Enter a MATLAB regular expression.
Classification types to include	Specify classifications to include or exclude from your report.	Enter a MATLAB regular expression.
Status types to include	Specify statuses to include or exclude from your report.	Enter a MATLAB regular expression.
Comments to include	Specify comments to include or exclude from your report.	Enter a MATLAB regular expression.

See Also

Topics
"Customize Existing Code Prover Report Template"
"Regular Expressions"

Run-time Checks Details Ordered by Color/File

Create overrides for global filters in Polyspace reports (Code Prover only)

Description

This component adds detailed information about the run-time checks to your report. This component can also be used to override global filters in specific chapters of your report. Use the following workflow when using filters in your report:

1   To create filters that apply to all chapters of your report, use the **Report Customization (Filtering)** component. For more information, see `Report Customization (Filtering)`.

2   To override some of the filters in individual chapters, use the **Run-time Checks Details Ordered by Color/File** component. Select the **Override Global Report filter** box.

Properties

Categories To Include

The properties in table below apply to the inclusion of Polyspace Code Prover checks in your report.

Property	Purpose
Red Checks	Specify whether to include red checks in your report. Red checks indicate proven run-time errors.
Gray Checks	Specify whether to include gray checks in your report. Gray checks indicate unreachable code.
Orange Checks	Specify whether to include orange checks in your report. Orange checks indicate possible run-time errors.
Green Checks	Specify whether to include green checks in your report. Green checks indicate that an operation does not contain a specific run-time error.
Inspection Point Checks	Specify whether to include inspection point checks in your report. These checks allow an user to find the values that a variable can take at a certain point in the code.
Unreachable Functions	Specify whether to include unreachable functions in your report.

Advanced Filters

The properties in table below apply to the inclusion of metrics, coding rule violations and Polyspace Code Prover checks in your report.

Property	Purpose	User Action
Justification status	Choose whether to report only justified checks, only unjustified checks or all checks.	Choose an option from the dropdown list.
Files to include	Specify files to include or exclude from your report.	Enter a regular MATLAB expression.
Check types to include	Specify Polyspace Code Prover checks to include in your report.	Enter a regular MATLAB expression.
Function names to include	Specify functions to include or exclude from your report.	Enter a regular MATLAB expression.
Classification types to include	Specify classifications to include or exclude from your report.	Enter a regular MATLAB expression.
Status types to include	Specify statuses to include or exclude from your report.	Enter a regular MATLAB expression.
Comments to include	Specify comments to include or exclude from your report.	Enter a regular MATLAB expression.

See Also

Topics
"Customize Existing Code Prover Report Template"

Run-time Checks Details Ordered by Review Information

Create table with run-time checks ordered by review information (Code Prover only)

Description

This component creates tables displaying the Polyspace Code Prover checks in your code. All checks with same combination of **Severity** and **Status** appear in the same table.

See Also

Topics
"Customize Existing Code Prover Report Template"

Run-time Checks Summary Ordered by File

Create table with run-time checks ordered by file (Code Prover only)

Description

This component creates a table displaying the number of Polyspace Code Prover checks per file in your code.

Properties

Sort the data

Use this option to sort the rows in the table alphabetically by filename or by percentage of unproven code.

Display as

Use this option to display the number of checks in a table or in bar charts.

Display ratio of checks in a file

Select this option to display the number of checks of a certain color as a ratio of total number of checks in the file.

Include checks from Polyspace standard library stub functions

Select this option to include the checks from Polyspace standard library stub functions in your display.

See Also

Topics
"Customize Existing Code Prover Report Template"

Software Quality Objectives - Coding Rules Summary

Create table of coding rule violations in results downloaded from Polyspace Metrics

Description

This component creates a table containing coding rule violations in results downloaded from Polyspace Metrics.

See Also

Topics
"Customize Existing Code Prover Report Template"

Software Quality Objectives - Run-time Checks Details

Create table of result details for results downloaded from Polyspace Metrics

Description

This component creates tables showing results downloaded from Polyspace Metrics.

The component `Software Quality Objectives - Run-time Checks Summary` shows the distribution of results. This component shows individual instances of results. Each file has a dedicated table showing the findings in the file.

See Also

Topics
"Customize Existing Code Prover Report Template"

Software Quality Objectives - Run-time Checks Summary

Create table of results summary for results downloaded from Polyspace Metrics

Description

This component creates a table showing the distribution of run-time checks in results downloaded from Polyspace Metrics.

This component shows the distribution of run-time checks. The component `Software Quality Objectives - Run-time Checks Details` shows the individual instances of run-time checks.

See Also

Topics
"Customize Existing Code Prover Report Template"

Summary By File

Create table showing summary of Polyspace results by file

Description

This component creates a table showing a breakdown of Polyspace results by file.

See Also

Topics
"Customize Existing Code Prover Report Template"

Variable Access

Create table showing global variable access in source code (Code Prover only)

Description

This component creates a table showing the global variable access in your source code. For each global variable, the table displays the following information:

- Variable name.

  The entry for each variable is denoted by |.
- Type of the variable.
- Number of read and write operations on the variable.
- Details of read and write operations. For each read or write operation, the table displays the following information:

  - File and function containing the operation in the form *file_name.function_name*.

    The entry for each read or write operation is denoted by ||. Write operations are denoted by < and read operations by >.
  - Line and column number of the operation.

This table captures the information available on the **Variable Access** pane in the Polyspace user interface.

The table showing variable access contains only the names of files. Below this table, a second table shows the full paths to files (in two columns, **Filename** and **Full filename**). If a variable access occurs in a Standard library function, the two columns contain this information:

- **Filename**: `__polyspace__stdstubs.c` (the file containing Polyspace implementation of Standard Library functions)
- **Full filename**: `Std library`

See Also

Topics
"Customize Existing Code Prover Report Template"

Variable Checks Details Ordered By Review Information

Create table with global variable results ordered by review information (Code Prover only)

Description

This component creates tables displaying the Polyspace Code Prover global variable results in your code. All checks with same combination of **Severity** and **Status** appear in the same table.

See Also

Topics
"Customize Existing Code Prover Report Template"

Configuration Parameters

Settings from (C)

Select settings for the analysis configuration. You can quickly activate coding rules checking for generated C code

Model Configuration Parameters Category: Polyspace

Settings

Default: `Project configuration`

`Project configuration`

Run Polyspace with the options specified in the "Project configuration" on page 12-7 or "Use custom project file" on page 12-6.

You do not check coding rules unless you select a rule set in the configuration.

`Project configuration and MISRA AC AGC checking`

Run Polyspace with the options specified in the **Project configuration** plus MISRA AC-AGC obligatory and recommended rules.

`Project configuration and MISRA C 2004 checking`

Run Polyspace with the options specified in the **Project configuration** plus all MISRA C 2004 rules.

`Project configuration and MISRA C 2012 checking`

Run Polyspace with the options specified in the **Project configuration** plus all MISRA C 2012 rules. This option automatically applies the rule categories for generated code. See `Use generated code requirements (-misra3-agc-mode)`.

`MISRA AC AGC checking`

Check compliance with the MISRA AC-AGC obligatory and recommended rules. After rules checking, Polyspace stops.

`MISRA C 2004 checking`

Check compliance with all MISRA C 2004 rules. After rules checking, Polyspace stops.

`MISRA C 2012 checking`

Check compliance with all MISRA C 2012 rules. This option automatically applies the rule categories for generated code. See `Use generated code requirements (-misra3-agc-mode)`. After rules checking, Polyspace stops.

Dependency

This setting overrides custom configuration settings in "Project configuration" on page 12-7 and "Use custom project file" on page 12-6. If you want to use your custom coding rule settings, select the `Project configuration` option.

Command-Line Information

Use the `pslinkoptions` property `VerificationSettings`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model.

Use the parameter `PSVerificationSettings` with the same value as for the `pslinkoptions` property `VerificationSettings`. See pslinkoptions.

See Also

pslinkoptions | `pslinkoptions`

Related Examples

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

Settings from (C++)

Select settings for the analysis configuration. This option allows you to quickly activate coding rules checking for generated C++ code.

Model Configuration Parameters Category: Polyspace

Settings

Default: `Project configuration`

`Project configuration`

> Run Polyspace with the options specified in the "Project configuration" on page 12-7 or "Use custom project file" on page 12-6.

> You do not check coding rules unless you select a rule set in the configuration.

`Project configuration and MISRA C++ checking`

> Run Polyspace with the options specified in the **Project configuration** plus MISRA C++ required rules.

`Project configuration and JSF C++ checking`

> Run Polyspace with the options specified in the **Project configuration** plus JSF C++ shall rules.

`MISRA C++ checking`

> Check compliance with the MISRA C++: 2008 required rules. After rules checking, Polyspace stops.

`JSF C++ checking`

> Check compliance with the JSF C++ shall rules. After rules checking, Polyspace stops.

Dependency

This setting overrides custom configuration settings in "Project configuration" on page 12-7 and "Use custom project file" on page 12-6. If you want to use your custom coding rule settings, select the `Project configuration` option.

Command-Line Information

Use the `pslinkoptions` property `CxxVerificationSettings`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSCxxVerificationSettings` with the same value as for the `pslinkoptions` property `CxxVerificationSettings`. See pslinkoptions.

See Also
pslinkoptions | pslinkoptions

Related Examples

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

Use custom project file

Set Polyspace configuration options with a custom `.psprj` file

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐
   Analysis uses configuration options from **Project configuration** on page 12-7 parameters.

On ☑
   Analysis uses configuration options from the specified `.psprj` project file.

Dependency

The **Settings from** parameter overrides custom configuration settings for coding rules. If you want to use your custom coding rule settings, set **Settings from** > **Project configuration**.

Command-Line Information

Use the `pslinkoptions` properties `EnablePrjConfigFile` and `PrjConfigFile`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameters `PSEnablePrjConfigFile` and `PSPrjConfigFile` with the same values as for the `pslinkoptions` properties `EnablePrjConfigFile` and `PrjConfigFile`. See pslinkoptions.

See Also
`pslinkoptions` | pslinkoptions

Related Examples
•   "Run Polyspace Analysis on Code Generated with Embedded Coder"

Project configuration

Set advanced configuration options to customize the analysis.

Settings

Open the Polyspace Configuration window by using the **Configure** button. Customize additional settings in this window and save your project configuration. If you added a custom project file in the parameter "Use custom project file" on page 12-6, that project file configuration is shown. Otherwise, the default project template is used.

For details about the advanced options, see "Analysis Options in Polyspace Code Prover".

Dependency

The **Settings from** parameter overrides custom configuration settings for coding rules. If you want to use your custom coding rule settings, set **Settings from > Project configuration**.

Command-Line Information

Use a Polyspace project (`.psprj` file) with the `pslinkoptions` properties `EnablePrjConfigFile` and `PrjConfigFile`.

See Also

`polyspace.ModelLinkOptions` | `pslinkoptions` | pslinkoptions

More About

- "Run Polyspace Analysis on Code Generated with Embedded Coder"

Enable additional file list

Add additional supporting code files to the analysis.

For instance, suppose you use C files for testing results from the generated code or providing inputs to the generated code. The analysis of generated code only considers files generated from the Simulink model. If you want the analysis to consider the C files that you use for testing or inputs, provide them as additional files.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐

    The analysis includes no additional files.

On ☑

    Polyspace analyzes the specified C/C++ files with the generated code. Use the **Select files** button to specify these additional files.

Command-Line Information

Use the `pslinkoptions` properties `EnableAdditionalFileList` and `AdditionalFileList`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameters `PSEnableAdditionalFileList` and `PSAdditionalFileList` with the same values as for the `pslinkoptions` properties `EnableAdditionalFileList` and `AdditionalFileList`. See pslinkoptions.

See Also
`pslinkoptions` | pslinkoptions

More About
•   "Run Polyspace Analysis on Code Generated with Embedded Coder"

Stub lookup tables

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

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

The option is relevant only if your model uses Lookup Table blocks.

Model Configuration Parameters Category: Polyspace

Settings

Default: On ☑

On ☑

> For autogenerated functions that use lookup tables with linear interpolation and no extrapolation, the verification:
>
> - Does not check for run-time errors in the function body.
> - Calls a function stub instead of the actual function at the function call sites. The stub ensures that the result of using the lookup table is within the bounds of the table.
>
> To identify if the lookup table in the function uses linear interpolation and no extrapolation, the verification uses information provided by the code generation product. For instance, if you use Embedded Coder to generate code, the lookup table functions with linear interpolation and no extrapolation follow specific naming conventions.

Off ☐

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

Tips

- The option applies only to autogenerated functions. If you integrate your own C/C++ S-Function using lookup tables with the model, the option does not cause them to be stubbed.
- The option is on by default. For certification purposes, if you want your verification tool to be independent of the code generation tool, turn off the option.

Command-Line Information

Use the `pslinkoptions` property `AutoStubLUT`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model.

Use the parameter `PSAutoStubLUT` with the same value as for the `pslinkoptions` property `AutoStubLUT`. See pslinkoptions.

See Also

pslinkoptions | pslinkoptions

More About

• "Run Polyspace Analysis on Code Generated with Embedded Coder"

Input

Choose whether to constrain Inport block variables.

Model Configuration Parameters Category: Polyspace

Settings

Default: `Use specified minimum and maximum values`

`Use specified minimum and maximum values`

Analysis assumes minimum and maximum values for input variables. These values are specified in the Inport block dialog box. Use this value to reduce the number of orange results.

`Unbounded inputs`

Analysis assumes full range for input variables. Use this value to run a robust analysis that includes values outside the expected range.

Command-Line Information

Use the `pslinkoptions` property `InputRangeMode`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSInputRangeMode` with the same value as for the `pslinkoptions` property `InputRangeMode`. See pslinkoptions.

See Also

`pslinkoptions` | pslinkoptions

More About

- "Run Polyspace Analysis on Code Generated with Embedded Coder"
- "External Constraints on Polyspace Analysis of Generated Code"

Tunable parameters

Choose how to treat tunable parameter values during the analysis. Treat values as either constants or a range of values.

Model Configuration Parameters Category: Polyspace

Settings

Default: `Use calibration data`

`Use calibration data`

> Analysis assumes constant values for tunable parameters. Use this value to run a contextual analysis. This option can reduce the number of orange results.

`Use specified minimum and maximum values`

> Analysis assumes a range of values for the tunable parameter variables. Specify maximum and minimum values in the model. Use this option to run a robust analysis that includes values outside the expected parameter value.

Command-Line Information

Use the `pslinkoptions` property `ParamRangeMode`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSParamRangeMode` with the same value as for the `pslinkoptions` property `ParamRangeMode`. See pslinkoptions.

See Also
pslinkoptions | pslinkoptions

More About
- "Run Polyspace Analysis on Code Generated with Embedded Coder"
- "External Constraints on Polyspace Analysis of Generated Code"

Output

Choose whether to verify output values.

Code Prover option only. Bug Finder cannot check output values.

Model Configuration Parameters Category: Polyspace

Settings

Default: No verification

No verification

Polyspace does not verify output values.

Verify outputs are within minimum and maximum values

Polyspace checks to see if the output variable values are within the expected minimum and maximum values. Specify the minimum and maximum values in the output block dialog boxes.

Command-Line Information

Use the `pslinkoptions` property `OutputRangeMode`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSOutputRangeMode` with the same value as for the `pslinkoptions` property `OutputRangeMode`. See pslinkoptions.

See Also
pslinkoptions | pslinkoptions

More About
• "Run Polyspace Analysis on Code Generated with Embedded Coder"
• "External Constraints on Polyspace Analysis of Generated Code"

Model reference verification depth

Only for models that use Embedded Coder generated code. Indicate how deep into the model hierarchy to analyze.

Model Configuration Parameters Category: Polyspace

Settings

Default: `Current model only`

`Current model only`

   Polyspace analyzes only the current model

`1`

   Polyspace analyzes the current model and the referenced models that are one level below the current model.

`2`

   Polyspace analyzes the current model and the referenced models that are up to two levels below the current model.

`3`

   Polyspace analyzes the current model and the referenced models that are up to three levels below the current model.

`All`

   Polyspace analyzes the current model and all referenced models.

Command-Line Information

Use the `pslinkoptions` property `ModelRefVerifDepth`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSModelRefVerifDepth` with the same value as for the `pslinkoptions` property `ModelRefVerifDepth`. See pslinkoptions.

See Also
`pslinkoptions` | pslinkoptions

More About
•   "Run Polyspace Analysis on Code Generated with Embedded Coder"

Model by model verification

Only for models that use Embedded Coder generated code. Analyze each model or referenced model individually. If you have a large project, this option can help modularize your analysis .

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐

Polyspace analyzes your models together. Model interactions are analyzed.

On ☑

Polyspace analyzes your model and each of its referenced models in isolation. This option does not analyze model interactions.

Command-Line Information

Use the `pslinkoptions` property `ModelRefByModelRefVerif`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSModelRefByModelRefVerif` with the same value as for the `pslinkoptions` property `ModelRefByModelRefVerif`. See pslinkoptions.

See Also
`pslinkoptions` | pslinkoptions

More About
• "Run Polyspace Analysis on Code Generated with Embedded Coder"

Output folder

Specify the location and folder name for your analysis results.

Model Configuration Parameters Category: Polyspace

Settings

Default: `results_$ModelName$`

Enter a path for your results folder. If you do not use a full path, the results folder is relative to your current MATLAB folder.

If you select "Add results to current Simulink project" on page 12-18, the results folder is relative to the Simulink project folder.

By default, the software stores your results in *Current Folder*\`results_model_name`.

Command-Line Information

Use the `pslinkoptions` property `ResultDir`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSResultDir` with the same value as for the `pslinkoptions` property `ResultDir`. See pslinkoptions.

See Also
`pslinkoptions` | pslinkoptions

More About
• "Run Polyspace Analysis on Code Generated with Embedded Coder"

Make output folder name unique by adding a suffix

Add a unique suffix to the results folder for every run to avoid overwriting previous results.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐

    Every time you rerun your analysis, your results are overwritten.

On ☑

    For each run of the analysis, Polyspace specifies a new location for the results folder by appending a unique number to the folder name.

Command-Line Information

Use the `pslinkoptions` property `AddSuffixToResultDir`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSAddSuffixToResultDir` with the same value as for the `pslinkoptions` property `AddSuffixToResultDir`. See pslinkoptions.

See Also
`pslinkoptions` | pslinkoptions

More About
•   "Run Polyspace Analysis on Code Generated with Embedded Coder"

Add results to current Simulink project

Add your Polyspace results to the current Simulink project. To use this option, you must have a Simulink project open.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐
  Results are saved to the current folder.

On ☑
  Results are saved to the currently open Simulink project.

Dependencies

You must have a Simulink project open to use this option.

Command-Line Information

Use the `pslinkoptions` property `AddToSimulinkProject`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSAddToSimulinkProject` with the same value as for the `pslinkoptions` property `AddToSimulinkProject`. See pslinkoptions.

See Also
`pslinkoptions` | pslinkoptions

More About
•   "Run Polyspace Analysis on Code Generated with Embedded Coder"

Open results automatically after verification

Decide whether to open your results in the Polyspace interface after running analysis from Simulink.

Model Configuration Parameters Category: Polyspace

Settings

Default: On ☑

On ☑

> After you run an analysis, your results open automatically in the Polyspace interface.

Off ☐

> You must manually open your results after running an analysis.

Command-Line Information

Use the `pslinkoptions` property `OpenProjectManager`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSOpenProjectManager` with the same value as for the `pslinkoptions` property `OpenProjectManager`. See pslinkoptions.

See Also
`pslinkoptions` | pslinkoptions

More About
• "Run Polyspace Analysis on Code Generated with Embedded Coder"

Check configuration before verification

Check whether model and code configurations are optimal for code analysis.

Model Configuration Parameters Category: Polyspace

Settings

Default: On (proceed with warnings)

On (proceed with warnings)

The process stops for errors, but continues the code analysis if the configuration has only warnings.

On (stop for warnings)

If the configuration has errors or warnings, the process stops.

Off

The software does not check the configuration.

Command-Line Information

Use the `pslinkoptions` property `CheckConfigBeforeAnalysis`. For details, see pslinkoptions.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSVerifALLSFcnInstances` with the same value as for the `pslinkoptions` property `VerifALLSFcnInstances`. See pslinkoptions.

See Also
pslinkoptions

More About
• "Run Polyspace Analysis on Code Generated with Embedded Coder"

Verify all S-function occurrences

For S-Function analyses only. Run an analysis on all instances of the selected S-Function.

Model Configuration Parameters Category: Polyspace

Settings

Default: Off ☐

Off ☐

> Analyze only the selected S-Function block. The analysis includes only information from the selected S-Function block.

On ☑

> Analyze all occurrences of the S-function in the model. If the S-Function is included in the model multiple times, information from all occurrences is included in the analysis.

Command-Line Information

Use the `pslinkoptions` property `VerifALLSFcnInstances`.

The `pslinkoptions` function allows you to create a Polyspace options object that you can reuse for multiple models. You can also use the `set_param` function to associate this property with the model. Use the parameter `PSVerifALLSFcnInstances` with the same value as for the `pslinkoptions` property `VerifALLSFcnInstances`. See pslinkoptions.

See Also
`pslinkoptions` | pslinkoptions

More About

• "Run Polyspace Analysis on Code Generated with Embedded Coder"