Polyspace® Code Prover™

Reference

MATLAB® & SIMULINK®

R2015a

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Polyspace® Code Prover™ Reference

Trademarks

Patents

Revision History

September 2013	Online Only	Revised for Version 9.0 (Release 2013b)
March 2014	Online Only	Revised for Version 9.1 (Release 2014a)
October 2014	Online Only	Revised for Version 9.2 (Release 2014b)
March 2015	Online Only	Revised for Version 9.3 (Release 2015a)

Contents

1 Option Descriptions

2 Option Descriptions specific to C++ Code

3 Polyspace Analysis Options — Command Line Only

4 Check Reference

5 Approximations Used During Verification

6 Examples

7 Functions

8 MISRA C 2012

9 Code Metrics

10 Custom Coding Rules

11 Global Variables

Option Descriptions

Target operating system (C/C++)

Specify the operating system of your target application. This option is available on the **Target & Compiler** node in the **Configuration** pane.

This information allows the corresponding system definitions to be used during preprocessing to analyze the included files properly.

A generic set of includes is provided with Polyspace®. These are automatically included when the operating system is set to `no-predefined-OS` or `Linux`. For projects developed for other operating systems, analyze these projects using the corresponding include files for that operating system.

Settings

Default: `no-predefined-OS`

`no-predefined-OS`

Analyzes with a general operating system set up. Use with preprocessor macros (`-U` or `-D`) to specify the system flags at compilation time.

`Linux`

Analyzes with the Linux® system definitions.

`Solaris`

Analyzes with the Solaris™ system definitions.

This option requires you to add a path to the Solaris include folder in your project, or use the `-I` option at the command line.

`VxWorks`

Analyzes with the VxWorks® system definitions.

This option requires you to add a path to the VxWorks include folder in your project, or use the `-I` option at the command line.

`Visual`

Analyzes with the Visual Studio® system definitions. Used for Microsoft® Windows® systems.

This option requires you to add a path to the Visual Studio include folder in your project, or use the `-I` option at the command line.

Dependencies

Setting this parameter changes the available **Dialect** options. All options are available with the `no-predefined-OS` option. The other operating systems only show usable dialects for that system.

Command-Line Information

Parameter: `-OS-target`
Value: `no-predefined-OS` | `Linux` | `Solaris` | `VxWorks` | `Visual`
Default: `no-predefined-OS`
Example: `polyspace-code-prover-nodesktop -os-target Linux`

See Also

"Dialect (C)" on page 1-13 | "Dialect (C++)" on page 2-5

Related Examples

- "Specify Analysis Options"

More About

- "Compile Operating System Dependent Code"

Target processor type (C)

Specify the target processor type. This option is available on the **Target & Compiler** node in the **Configuration** pane.

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

Settings:

Default: `i386`

You can modify some default attributes by selecting the browse button to the right of the **Target processor type** drop-down menu. The optional settings for each target are shown in [brackets] in the table.

Target	char	short	int	long	long long	float	double	long double	ptr	sign of char	endian	align
`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 / ColdFire`[a]	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	40[b]	32	signed	Little	32
`sharc21x61`	32	32	32	32	64	32	32 [64]	32 [64]	32	signed	Little	32
`NEC-V850`	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]

Target	char	short	int	long	long long	float	double	long double	ptr	sign of char	endian	align
`c18`	8	16	16	32 [24][e]	32	32	32	32	16 [24]	signed	Little	8
`x86_64`	8	16	32	64 [32]	64	32	64	128	64	signed	Little	64 [32]
`mcpu... (Advanced)`	8 [16]	8 [16]	16 [32]	32	32 [64]	32	32 [64]	32 [64]	16 [32]	signed	Little	32 [16, 8]

a. The M68k family (68000, 68020, etc.) includes the “ColdFire” processor
b. Operations on long double values will be imprecise.
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.
f. `mcpu` is a reconfigurable Micro Controller/Processor Unit target. You can use this type to configure one or more generic targets.

Tips

If your processor is not listed, use a similar processor that shares the same characteristics, or create an `mpcu` generic target processor. If your target processor does not match the characteristics of a processor described above, contact MathWorks® technical support for advice.

Command-Line Information

Parameter: `-target`
Value: `i386 | m68k | powerpc | c-167 | x86_64 | tms320c3x | sharc21x61 | necv850 | hc08 | hc12 | mpc5xx | c18 | mpcu`
Default: `i386`
Example: `polyspace-code-prover-nodesktop -lang c -target m68k`

See Also

“Generic target options (C/C++)” on page 1-9

Related Examples

- “Specify Analysis Options”

- "Modify Predefined Target Processor Attributes"
- "Define Generic Target Processors"

Generic target options (C/C++)

The **Generic target options** dialog box is only available when you select a `mcpu` target for **Target processor type**. The **Target processor type** option is available on the **Target & Compiler** node in the **Configuration** pane.

Allows the specification of a generic “Micro Controller/Processor Unit" target. Use the dialog box to specify the name of a new `mcpu` target — e.g., *`MyTarget`*.

The generic target option is incompatible with either:

- **Target operating system** set to `Visual`
- **Dialect** set to `visual*`

That new target is added to the **Target processor type** option list. The default characteristics of the new target are (using the *type [size, alignment]* format):

- *char [8, 8], char [16,16]*
- *short [8,8], short [16, 16]*
- *int [16, 16]*
- *long [32, 32], long long [32, 32]*
- *float [32, 32], double [32, 32], long double [32, 32]*
- *pointer [16, 16]*
- *char is signed*
- *little-endian*

Changing the genetic target has consequences for:

- Detection of overflow
- Computation of `sizeof` objects

Command-Line Options

When using the command line, specify your target with the other 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-nodesktop -lang c -target mcpu -little-endian`
`-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-nodesktop -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-nodesktop -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-nodesktop -target mcpu -char-is-16bits`

Option	Description	Available With...	Example
`-short-is-8bits`	Define `short` as 8 bits, regardless of sign	`mcpu`	`polyspace-code-prover-nodesktop -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-nodesktop -target mcpu -long-long-is-64bits`
`-long-long-is -64bits`	Define `long long` as 64 bits, regardless of sign. Alignment is also set to 64 bits.	`mcpu`	`polyspace-code-prover-nodesktop -target mcpu -long-long-is-64bits`

`-double-is -64bits`	Define `double` and `long double` as 64 bits, regardless of sign. Alignment is also set to 64 bits.	`mcpu`,
		`sharc21x6`
		`hc08`,
		`hc12`,
		`mpc5xx`
			`polyspace-code-prover-nodesktop -target mcpu -double-is-64bits`

`-pointer-is -32bits`	Define pointer as 32 bits, regardless of sign. Alignment is also 32 bits.	`mcpu`	`polyspace-code-prover-nodesktop -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`, Only 16 or 32 bits for: `hc08`, `hc12`, `mpc5xx`	`polyspace-code-prover-nodesktop -target mcpu -align 16`

See Also

“Target processor type (C)” on page 1-6 | “Target processor type (C++)” on page 2-3

Related Examples

- “Define Generic Target Processors”

More About

- “Common Generic Targets”

Dialect (C)

Allow syntax associated with C language extensions. This option is available on the **Target & Compiler** node in the **Configuration** pane.

Using this option allows additional structure types as keywords of the language, such as `sfr`, `sbit`, and `bit`. These structures and associated semantics are part of the compiler that extends the ANSI® C language.

Settings

Default: `none`

`none`

Analysis allows only ANSI C standard syntax.

`gnu4.6`

Analysis allows GCC 4.6 dialect syntax.

`gnu4.7`

Analysis allows GCC 4.7 dialect syntax.

For more information, see "Limitations" on page 1-14.

`gnu4.8`

Analysis allows GCC 4.8 dialect syntax.

`visual10`

Analysis allows Visual C++® 2010 syntax.

`visual11.0`

Analysis allows Visual C++ 2012 syntax.

`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).

Dependency

This parameter is dependant on the value of **Target operating system**. The dialect options work only with the applicable operating systems. You can use every dialect with the **Target operating system** option, `no-predefined-OS`.

Limitations

Polyspace does not support certain aspects of the GNU® 4.7 dialect. These limitations can cause compilation errors, incomplete results, or false positives.

- **Vector types and attributes** — Not supported, ignores attributes.

  Workaround: To reduce compilation issues

  - At the command line, use the option `-D _EMMINTRIN_H_INCLUDED -D _XMMINTRIN_H_INCLUDED`.
  - In the Polyspace environment, in **Macros** > **Preprocessor definitions**, add two rows: `_EMMINTRIN_H_INCLUDED` and `_XMMINTRIN_H_INCLUDED`.
- **Visibility attributes** — Not supported, ignored.

  This limitation can cause C++ linkage problems in Polyspace Code Prover™.

  Workaround: Remove all attributes during preprocessing,

  - At the command line, use the option `-D __attribute__(x)=`.
  - In the Polyspace environment, in **Macros** > **Preprocessor definitions**, add a row: `__attribute__(x)=`.
- **Complex types** — Only floating complex types supported, integral complex types cause an error.
- **Using built-in library function on complex types** — Not supported, stubbed during analysis. Calls to these functions will return variables with full ranges.

  Workaround: To make the analysis more precise, add an include file that defines the functions for complex variables.
- **Computed `goto`** — Not supported.

  This causes an error in Code Prover. To ignore the computed gotos, stub the functions containing the computed gotos:

  - At the command line, use the option `-functions-to-stub` *`funcList`* where *`funcList`* is the list of functions containing the computed gotos.
  - In the Polyspace environment, in the **Inputs & Stubbing** > **Functions to stub** table, use the ✚ button to add a row for each function containing the computed gotos.
- **Nested functions** — Not supported, causes an error.
- **Using built-in library functions on atomic operators** — Not supported, Polyspace stubs the functions. This limitation can cause imprecise results.
- **IEEE® floating point library functions** — Not supported, causes compilation error.

  This limitation includes `isnan`, `isnanf`, `isnanl`, `isinf`, `isinff`, `isinfl`, `isnormal`, and `isfinite`.

  Workaround: In each of your source files, include a file containing the function definitions or declarations:

  - At the command line, use the option `-include` *`filename`*.
  - In the Polyspace environment, in **Environment Settings** > **Include**, use the ✚ button to add a row for your definition/declaration file.

Command-Line Information

Parameter: `-dialect`
Value: `none | gnu4.6 | gnu4.7 | visual10 | visual11.0 | keil | iar`
Default: `none`
Example: `polyspace-code-prover-nodesktop -sources "`*`file1.c,file2.c`*`" -lang c -OS-target Linux -dialect gnu4.6`

See Also

“Target operating system (C/C++)” on page 1-4 | “Target processor type (C)” on page 1-6

Related Examples

- “Verify Keil or IAR Dialects”

Sfr type support (C)

Specify the `sfr` types. This option is available on the **Target & Compiler** node in the **Configuration** pane.

If the code uses `sfr` keywords, you must declare each `sfr` type using this option.

Settings

No Default

List each sfr name and its size in bits.

Dependency

Setting **Dialect** to `keil` or `iar` enables this parameter.

Command-Line Information

Parameter: `-sfr-types` *`sfr_name=size_in_bits`*`,...`
No Default
Name Value: an sfr name
Size Value: `8 | 16 | 32`
Example:

```
polyspace-code-prover-nodesktop -lang c -dialect iar -sfr-types sfr=8,sfr32=32,sfrb=16 ...
```

Division round down (C)

Specify how division and modulus of a negative numbers is interpreted by the analysis. This option is available on the **Target & Compiler** node in the **Configuration** pane.

The ANSI 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*".

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

Settings

Default: Off

□ Off

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

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

☑ On

If either operand `/` or `%` is negative, the result of the `/` operator is the largest integer less or equal than 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.

Command-Line Information

Parameter: `-div-round-down`
Default: Off
Example: `polyspace-code-prover-nodesktop -div-round-down`

Enum type definition (C)

Allow the analysis to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition. This option is available on the **Target & Compiler** node in the **Configuration** pane.

When using this option, each enum type is represented by the smallest integral type that can hold its enumeration values.

Settings

Default: `signed-int`

`signed-int`

Uses the signed integer type for all dialects except gnu.

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

`auto-signed-first`

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

`auto-unsigned-first`

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

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

Command-Line Information

Parameter: `-enum-type-definition`
Value: `signed-int | auto-signed-first | auto-unsigned-first`
Default: `signed-int`
Example: `polyspace-code-prover-nodesktop -lang -c -enum-type-definition auto-signed-first`

Signed right shift (C)

Choose between arithmetical and logical computation. This option is available on the **Target & Compiler** node in the **Configuration** pane.

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
```

Command-Line Information

When using the command line, arithmetic is the default computation mode. When this option is set, logical computation will be performed.
Parameter: `-logical-signed-right-shift`
Example: `polyspace-code-prover-nodesktop -logical-signed-right-shift`

Preprocessor definitions (C/C++)

Define macro compiler flags. This option is available on the **Macros** node in the **Configuration** pane.

Depending on your **Target operating system**, some compiler flags are defined by default. Use this option to define flags that are not already defined.

Settings

No Default

Using the ✚ button, add a row for the macro flag you want to define. The flag must be in the format *Flag*=*Value*. If you want Polyspace to ignore the flag, 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.

Tips

Sometimes, your source code contains non-ANSI extension keywords. Although your compiler supports the keywords, Polyspace does not support them. To avoid compilation errors caused by an unsupported keyword, use this option to replace all occurrences of the keyword with a blank string in preprocessed code.

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

- In the user interface, 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;
```

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: `polyspace-code-prover-nodesktop -D HAVE_MYLIB -D int32_t=int`

See Also

"Disabled preprocessor definitions (C/C++)" on page 1-22

Disabled preprocessor definitions (C/C++)

Disable macro compiler flags. This option is available on the **Macros** node in the **Configuration** pane.

Some **Target operating system** settings enable macro compilation flags by default. This option allows you disable these macros.

Settings

No Default

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

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: *`flag`*
Example: `polyspace-code-prover-nodesktop -U HAVE_MYLIB -U USE_COM1`

See Also

"Preprocessor definitions (C/C++)" on page 1-20

Code from DOS or Windows file system (C/C++)

Specify that DOS or Windows files are in analysis. This option is available on the **Environment Settings** node in the **Configuration** pane.

Use this options if the contents of the **Include** or **Source** folder come from a DOS or Windows file system. It deals with upper/lower case sensitivity and control character issues.

Settings

Default: On

☑ On

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"
```

□ Off

Characters are not controlled for files names or paths.

Command-Line Information

Parameter: `-dos`
Default: On
Example: `polyspace-code-prover-nodesktop -dos -I ./my_copied_include_dir -D test=1`

Calculate Code Metrics (C/C++)

Specify that Polyspace must compute and display code complexity metrics for your source code. For more information, see “Code Metrics”.

Settings

Default: Off

☑ On

Polyspace computes and displays code complexity metrics on the **Results Summary** pane.

□ Off

Polyspace does not compute complexity metrics.

Command-Line Information

Parameter: `-code-metrics`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-code-metrics`

Command/script to apply to preprocessed files (C/C++)

Specify a perl script to run on each source file after the preprocessing phase. This option is available on the **Environment Settings** node in the **Configuration** pane.

When this option is used, the specified script file or command is run just after the preprocessing phase on each preprocessed `.c` file.

The command should be designed to process the standard output from preprocessing and produce its results in accordance with that standard output. Additionally, It is important to preserve the number of lines in the preprocessed file. Adding a line or removing one could result in some unpredictable behavior on the location of checks and MACROS in the Polyspace viewer.

Note: The Compilation Assistant is automatically disabled when you specify this option.

Example Script

This script, called `replace_keywords`, replaces the keyword “Volatile” by “Import”.

```
#!/usr/bin/perl
my $TOOLS_VERSION = "V1_4_1";
binmode STDOUT;

# Process every line from STDIN until EOF
while ($line = <STDIN>)
{
  # Change Volatile to Import
  $line =~ s/Volatile/Import/;
  print $line;
}
```

To run this script on preprocessed files:

- On a Linux or Mac workstation: `polyspace-code-prover-nodesktop -post-preprocessing-command 'pwd'/replace_keywords`
- On a Windows workstation you must give the full path to the Perl scripter: `*matlabroot*\polyspace\bin\polyspace-code-prover-nodesktop.exe -post-preprocessing-command *matlabroot*\sys\perl\win32\bin\perl.exe <*absolute_path*>\replace_keywords`

Command-Line Information

Parameter: `-post-preprocessing-command`
No Default
Value: Path to executable file or command in quotes

Continue with compile error (C/C++)

Continue verification even if some source files do not compile. This option is available on the **Environment Settings** node in the **Configuration** pane.

Settings

Default: Off

□ Off

If a source file does not compile, the verification stops.

Functions that are used but not specified are stubbed automatically.

☑ On

Continues the verification even if only one file compiles. Files that have compilation errors are not verified. This means that the results may not contain all coding rule violations or errors.

Functions that are used but not specified are stubbed automatically.

Command-Line Information

Parameter: `-continue-with-compile-error`
Default: Off
Example: `polyspace-code-prover-nodesktop -continue-with-compile-error`

Include (C/C++)

Specify files to be included by each C file involved in the analysis. This option is available on the **Environment Settings** node in the **Configuration** pane

Settings

No Default

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

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

Tips

If you have compilation problems because Polyspace does not recognize certain keywords specific to your compiler, you can define the keywords in a header file and provide the header file with this option.

Command-Line Information

Parameter: `-include`
Default: None
Value: *`file`* (Use `-include` multiple times for multiple files)
Example: `polyspace-code-prover-nodesktop -include `pwd`/sources/a_file.h -include /inc/inc_file.h`

Include folders (C/C++)

View the include folders used for verification.

- 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. Unlike other options, you do not specify include folders on the **Configuration** pane. Instead, you add your include folders on the **Project Browser** pane.

Command-Line Information

Parameter: `-I`
Value: Folder name
Example: `polyspace-code-prover-nodesktop -I /com1/inc -I /com1/sys/inc`

See Also

-I | "Include (C/C++)"

Multitasking (C/C++)

Specify whether the code is intended for a multitasking application. This option is available on the **Multitasking** node in the **Configuration** pane.

Settings

Default: Off

☑ On

The code is intended for a multitasking application.

Polyspace verifies functions that are called by the `main` and other entry-point functions.

□ Off

The code is not intended for a multitasking application.

- If a `main` exists, Polyspace verifies only those functions that are called by the `main`.
- If a `main` does not exist, Polyspace verifies all functions. To verify all functions, Polyspace generates a `main` function and calls functions from the generated `main` in a sequence you specify. For more information, see "Verify module (C)" or "Verify module (C++)".

Dependencies

To enable this option, on the **Configuration** pane, select **Code Prover Verification**. Select **Verify whole application**.

Command-Line Information

There is no command-line option to solely turn on multitasking verification. However, using the option `-entry-points` turns on multitasking verification.

See Also

"Entry points (C/C++)" | "Critical section details (C/C++)" | "Temporally exclusive tasks (C/C++)"

Related Examples

- “Model Tasks”
- “Model Tasks if main Contains Infinite Loop”
- “Model Execution Sequence in Tasks”

More About

- “Verify Multitasking Applications”

Entry points (C/C++)

Specify functions that serve as entry points to your code. Use this option when your code is intended for multitasking. This option is available on the **Multitasking** node in the **Configuration** pane.

Settings

No Default

Click ✚ to add a field. Enter function name.

Dependencies

This option is enabled only if you select the **Multitasking** box.

To enable this option, on the **Configuration** pane, select **Code Prover Verification**. Select **Verify whole application**.

Tips

- The entry point function must have the form

  ```
  void functionName (void)
  ```

- If a function `func` takes arguments, you cannot use it directly as entry point. To use `func` as entry point:

  1. Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
  2. Declare arguments to `func` as `volatile` variables local to `newFunc`. Call `func` inside `newFunc`.
  3. Specify `newFunc` as entry point.

- If a function `func` models cyclic tasks or interrupts that can run zero or more times, to specify the multiple cycles for Polyspace:

  1. Create a new function `newFunc` of the form

     ```
     void newFunc (void)
     ```

2. In the body of newFunc, call func inside a loop with unspecified number of runs. Make the loop control variable volatile int. For example:

```
void newFunc(void) {
  volatile int randomValue = 0;
  while(randomValue) {
     func();
   }
}
```

3. Specify newFunc as entry point.

Command-Line Information

Parameter: -entry-points
No Default
Value: *function1*[,*function2*[,...]]
Example: polyspace-code-prover-nodesktop -sources *file_name* -entry-points func_1,func_2

See Also

“Critical section details (C/C++)” | “Temporally exclusive tasks (C/C++)”

Related Examples

- “Specify Analysis Options”
- “Model Tasks”
- “Model Tasks if main Contains Infinite Loop”
- “Model Execution Sequence in Tasks”

More About

- “Verify Multitasking Applications”

Critical section details (C/C++)

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock function and an unlock function. Specify the two function names. This option is available on the **Multitasking** node in the **Configuration** pane.

When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait until `my_task` calls the corresponding unlock function.

Settings

No Default

Click ✚ to add a field.

- In **Starting procedure**, enter name of lock function.
- In **Ending procedure**, enter name of unlock function.

Dependencies

This option is enabled only if you select the **Multitasking** box.

Tips

- 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 procedure: `func_begin`

  Ending procedure: `func_end`

  ```
  void func() {
     func_begin(1);
     /* Critical section code */
     func_end(1);
  }
  ```

  Starting procedure: `func_begin`

  Ending procedure: `func_end`

  ```
  void func() {
     func_begin(2);
     /* Critical section code */
     func_end(2);
  }
  ```

Command-Line Information

Parameter: `-critical-section-begin | -critical-section-end`

No Default
Value: `function1:cs1[,function2:cs2[,...]]`
Example: `polyspace-code-prover-nodesktop -sources file_name -critical-section-begin func_begin:cs1 -critical-section-end func_end:cs1`

See Also

Polyspace Analysis Options

"Multitasking (C/C++)" | "Entry points (C/C++)" | "Temporally exclusive tasks (C/C++)"

Polyspace Results

Shared protected global variable | Shared unprotected global variable

Related Examples

- "Specify Analysis Options"
- "Prevent Concurrent Access Using Critical Sections"

More About

- "Verify Multitasking Applications"

Temporally exclusive tasks (C/C++)

Specify functions that cannot execute concurrently. The execution of the functions cannot overlap with each other. Use this option to implement temporal exclusion in multitasking code. This option is available on the **Multitasking** node in the **Configuration** pane.

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.

Dependencies

This option is enabled only if you select the **Multitasking** box.

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.

Parameter: `-temporal-exclusions-file`
No Default
Value: Name of temporal exclusions file
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-temporal-exclusions-file "C:\exclusions_file.txt"`

See Also

Polyspace Analysis Options

“Multitasking (C/C++)” | “Entry points (C/C++)” | “Critical section details (C/C++)”

Polyspace Results

Shared protected global variable | Shared unprotected global variable

Related Examples

- “Specify Analysis Options”
- “Prevent Concurrent Access Using Temporally Exclusive Tasks”

More About

- “Verify Multitasking Applications”

Check MISRA C:2004

Specify whether to check for violation of MISRA C®:2004 rules. Each value of the option corresponds to a subset of rules to check. This option is available on the **Coding Rules** node in the **Configuration** pane.

After analysis, the **Results Summary** pane lists the coding rule 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: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)”.

`custom`

Specify coding rules to check. Click Edit to create a coding rules file. After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click Edit. Click 📁 to load the file.

Format of the custom file:

`rule number` `off|on`

Use # to enter comments in the file. For example:

```
10.5 off # rule 10.5: type conversion
17.2 on # rule 17.2: pointers
```

Tips

To reduce unproven results:

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.

Command-Line Information

Parameter: `-misra2`
Value: `required-rules` | `all-rules` | `SQO-subset1` | `SQO-subset2` | *`file`*
Default: `required-rules`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-misra2 all-rules`

See Also

"Files and folders to ignore (C)"

Related Examples

- "Specify Analysis Options"
- "Set Up Coding Rules Checking"

More About

- "Polyspace MISRA C 2004 and MISRA AC AGC Checkers"
- "Software Quality Objective Subsets (C:2004)"

Check MISRA AC AGC

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. This option is available on the **Coding Rules** node in the **Configuration** pane.

After analysis, the **Results Summary** pane lists the coding rule violations. On the **Source** pane, for every coding rule violation, 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.

`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)”.

`custom`

Specify coding rules to check. Click Edit to create a coding rules file.

After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click Edit. Click 📁 to load the file.

Format of the custom file:

```
rule number off|on
```

Use # to enter comments in the file. For example:

```
10.5 off # rule 10.5: type conversion
17.2 on # rule 17.2: pointers
```

Tips

To reduce unproven results:

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.

Command-Line Information

Parameter: `-misra-ac-agc`
Value: `OBL-rules | OBL-REC-rules | all-rules | SQO-subset1 | SQO-subset2 | file`
Default: `OBL-rules`
Example: `polyspace-code-prover-nodesktop -sources file_name -misra-ac-agc all-rules`

Related Examples

- "Specify Analysis Options"
- "Set Up Coding Rules Checking"

More About

- "Polyspace MISRA C 2004 and MISRA AC AGC Checkers"
- "MISRA C:2004 Coding Rules"
- "Software Quality Objective Subsets (AC AGC)"

Check MISRA C:2012

Specify whether to check for violations of MISRA C:2012 guidelines. Each value of the option corresponds to a subset of guidelines to check. This option is available on the **Coding Rules** node in the **Configuration** pane.

After analysis, the **Results Summary** pane lists the coding rule 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-required`

Check mandatory and required guidelines.

`mandatory`

Check mandatory guidelines.

`all`

Check mandatory, required, and advisory guidelines.

`SQO-subset1`

Check 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 a subset of guidelines, `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)”.

`custom`

Specify guidelines to check. Click Edit to create a coding rules file. Save the file. To reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click Edit. Click 📁 to load the file.

Custom file format:

rule number off|on
Use # to enter comments in the file. For example:

```
10.5 off # rule 10.5: essential type model
17.2 on # rule 17.2: functions
```

Tips

To reduce unproven results:

1. Find coding rule violations in `SQO-subset1`. Fix your code to address the violations. Rerun verification.
2. Find coding rule violations in `SQO-subset2`. Fix your code to address the violations. Rerun verification.

Command-Line Information

Parameter: `-misra3`
Value: `mandatory | mandatory-required | all | SQO-subset1 | SQO-subset2 | file`
Default: `mandatory-required`
Example: `polyspace-code-prover-nodesktop -lang c -sources file_name -misra3 mandatory-required`

See Also

"Files and folders to ignore (C)"

Related Examples

- "Specify Analysis Options"
- "Set Up Coding Rules Checking"

More About

- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Use generated code requirements (C)

Specify whether to use the MISRA C:2012 categories for automatically generated code. This option changes which rules are mandatory, required, or advisory. This option is available on the **Coding Rules** node in the **Configuration** pane.

Settings

Default: Off (On for analyses started from the Simulink® plug-in.)

□ Off

Use the normal categories (mandatory, required, advisory) for MISRA C:2012 coding guideline checking.

☑ On

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
- 14.4, 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** option.

Command-Line Information

Parameter: `-misra3-agc-mode`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-misra3 all -misra3-agc-mode`

See Also

“Files and folders to ignore (C)” | “Check MISRA C:2012” on page 1-42

Related Examples

- “Specify Analysis Options”
- “Set Up Coding Rules Checking”

More About

- “Polyspace MISRA C:2012 Checker”

Check custom rules (C/C++)

Define naming conventions for identifiers and check your code against them. This option is available on the **Coding Rules** node in the **Configuration** pane.

After analysis, the **Results Summary** 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.

Settings

Default: Off

☑ 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. The New File window opens.

  2 From the drop-down list **Set the following state to all Custom C**, select `Off`. Click **Apply**.

  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 **Check 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_`.

- Manually edit an existing custom coding rules file:

  1. Open the file with a text editor.
  2. For every custom rule you want to check, enter the following information in adjacent lines.

     a. Rule number, followed by `on`. For example:

     ```
     4.3 on
     ```

     b. The error message you want to display starting with `convention=`. For example:

     ```
     convention=All struct fields must begin with s_
     ```

     c. The text pattern starting with `pattern=`. For example:

     ```
     pattern=s_[A-Za-z0-9_]
     ```

To use an existing coding rules file, enter the full path to the file in the field provided or use 📁 in the New File window to navigate to the file location.

☐ Off

Polyspace does not check your code against custom naming conventions.

Command-Line Information

Parameter: `-custom-rules`
Value: Name of coding rules file
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-custom-rules "C:\Rules\myrules.txt"`

Related Examples

- "Specify Analysis Options"
- "Set Up Coding Rules Checking"
- "Create Custom Coding Rules"

More About

- “Format of Custom Coding Rules File”
- “Custom Coding Rules”

Files and folders to ignore (C)

Specify files and folders to ignore during coding rules checking. This option is available on the **Inputs & Stubbing** node in the **Configuration** pane.

The files and folders are **not** ignored during Code Prover verification.

Settings

Default: `all-headers`

`all-headers`

Ignore included `.h` files

`all`

Ignore all files in include folders

`custom`

Ignore include files and folders that you specify in the **File/Folder** view. To add files to the custom **File/Folder** list, select 📁 to choose the files and folders to exclude. To remove a file or folder from the list of excluded files and folders, select the row. Then click ✖.

Dependencies

This option is enabled only if you select one of the options **Check MISRA C:2004**, **Check MISRA C:2012**, **Check MISRA AC AGC** or **Check custom rules**.

Command-Line Information

Parameter: `-includes-to-ignore`
Value: `all-headers | all |` *`file1`*`[,`*`file2`*`[,...]] |` *`folder1`*`[,`*`folder2`*`[,...]]`
Default: `all-headers`
Example: `polyspace-code-prover-nodesktop -lang c -sources` *`file_name`* `-misra2 required-rules -includes-to-ignore "C:\usr\include"`

See Also

"Check MISRA C:2004" | "Check MISRA C:2012" | "Check MISRA AC AGC" | "Check custom rules (C/C++)"

Related Examples

- “Specify Analysis Options”
- “Set Up Coding Rules Checking”
- “Exclude Files from Rules Checking”

Effective boolean types (C)

Specify data types that you want Polyspace to treat as Boolean. You can specify a data type only if you have defined it through a `typedef` statement in your source code. This option is available on the **Coding Rules** node in the **Configuration** pane.

Use this option to allow Polyspace to check the following coding rules:

- MISRA C: 2004

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
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 the options **Check MISRA C:2004**, **Check MISRA AC AGC** or **Check MISRA C:2012**.

Command-Line Information

Parameter: `-boolean-types`
Value: `type1[,type2[,...]]`
No Default
Example: `polyspace-code-prover-nodesktop -sources filename -misra2 required-rules -boolean-types boolean1_t,boolean2_t`

See Also

“Check MISRA C:2004” | “Check MISRA AC AGC”

Related Examples

- “Set Up Coding Rules Checking”
- “Specify Effective Boolean Types”

More About

- “MISRA C:2004 Coding Rules”

Allowed pragmas (C)

Specify pragma directives for which MISRA C rule 3.4 should not be applied. MISRA C or MISRA® AC AGC rule 3.4 requires checking that all pragma directives are documented within the documentation of the compiler. This option is available on the **Coding Rules** node in the **Configuration** pane.

Settings

No Default

Click ✚ to add a field. Enter the pragma name that you want Polyspace to ignore during MISRA C checking .

Dependencies

This option is enabled only if you select one of the options **Check MISRA C:2004** or **Check MISRA AC AGC**.

Command-Line Information

Parameter: `-allowed-pragmas`
Value: `*pragma1*[,*pragma2*[,...]]`
No Default
Example: `polyspace-code-prover-nodesktop -sources *filename* -misra2 required-rules -allowed-pragmas pragma_01,pragma_02`

See Also

“Check MISRA C:2004” | “Check MISRA AC AGC”

Related Examples

- “Set Up Coding Rules Checking”

More About

- “MISRA C:2004 Coding Rules”

Verify whole application (C/C++)

Specify that Polyspace verification must stop if a `main` function is not present in the source files. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: Off

● On

Polyspace verification stops if it does not find a `main` function in the source files.

○ Off

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.

Command-Line Information

Parameter: `-main`
Default: On

See Also

"Verify module (C)" | "Verify module (C++)"

Related Examples

- "Specify Analysis Options"
- "Verify C Application Without main Function"

Verify module (C)

Specify that Polyspace must generate a `main` function if it does not find one in the source files. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: On

● On

Polyspace generates a `main` function if it does not find one in the source files. The generated `main`:

- Initializes variables that you specify using **Variables to initialize**.
- Calls functions that you specify using **Initialization functions** ahead of other functions.
- Calls functions that you specify using **Functions to call** in arbitrary order.

If you do not specify the above options explicitly, the generated `main`:

- Initializes all global variables except those declared with keywords `const` and `static`.
- Calls in arbitrary order 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 verification if a `main` function is not present in the source files.

Command-Line Information

Parameter: `-main-generator`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator ...`

See Also

"Verify whole application (C/C++)" | "Variables to initialize (C)" | "Parameters (C)" on page 1-65 | "Inputs (C)" on page 1-67 | "Initialization functions (C)" on page 1-69 | "Step functions (C)" on page 1-70 | "Termination functions (C)" on page 1-72

Related Examples

- "Specify Analysis Options"
- "Verify C Application Without main Function"
- "Configure Polyspace Analysis Options and Properties"

More About

- "Main Generation for Model Verification"

Variables to initialize (C)

Specify global variables that you want the generated main to initialize. Despite the initialization, Polyspace considers these variables to have any value allowed by their type. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: public

none

The generated main does not initialize global variables.

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 ✚ to add a field. Enter a global variable name.

Dependencies

This option is enabled only if you select **Code Prover Verification** > **Verify module**.

Command-Line Information

Parameter: -main-generator-writes-variables
Value: none | public | all | custom=*variable1*[,*variable2*[,...]]
Default: public
Example: polyspace-code-prover-nodesktop -sources *file_name* -main-generator -main-generator-writes-variables all

See Also

“Verify module (C)” | “Initialization functions (C)” | “Functions to call (C)”

Related Examples

- “Specify Analysis Options”
- “Verify C Application Without main Function”

Initialization functions (C)

Specify functions that you want the generated `main` to call ahead of other functions. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

No Default

Click ➕ to add a field. Enter the name of a 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: `polyspace-code-prover-nodesktop -sources *file_name* -main-generator -functions-called-before-main myfunc`

Dependencies

This option is enabled only if you select **Code Prover Verification** > **Verify module**.

See Also

“Verify module (C)” | “Variables to initialize (C)” | “Functions to call (C)”

Related Examples

- “Specify Analysis Options”
- “Verify C Application Without main Function”

Functions to call (C)

Specify the functions that you want the generated `main` to call. The `main` calls these functions after the ones you specify through the **Initialization functions** option. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

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. Click ✚ to add a field. Enter the name of a function.

Dependencies

This option is enabled only if you select **Code Prover Verification** > **Verify module**.

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.

Command-Line Information

Parameter: `-main-generator-calls`
Value: `none | unused | all | custom=`*`function1`*`[,`*`function2`*`[,...]]`
Default: `unused`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -main-generator-calls all`

See Also

"Verify module (C)" | "Variables to initialize (C)" | "Initialization functions (C)"

Related Examples

- "Specify Analysis Options"
- "Verify C Application Without main Function"

Verify files independently (C/C++)

Specify that a separate verification job will be created for each source file. 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. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: Off

☑ On

Polyspace creates a separate verification job for each source file.

□ Off

Polyspace creates a single verification job for all source files in a module.

Dependencies

This option is enabled only if you select **Code Prover Verification** > **Verify module** on the **Configuration** pane.

Tips

- If you perform a file by file verification, you cannot specify multitasking options.

Command-Line Information

Parameter: `-unit-by-unit`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-unit-by-unit`

See Also

"Common source files (C/C++)"

Related Examples

- "Specify Analysis Options"

- “Run File-by-File Local Verification”
- “Run File-by-File Remote Verification”

Common source files (C/C++)

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. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

For instance, if multiple source files call the same function, use this option to specify the file that contains the function definition. Otherwise, Polyspace stubs functions that are called but not defined in the source files.

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**.

Command-Line Information

Parameter: `-unit-by-unit-common-source`
Value: `*file1*[,*file2*[,...]]`
No Default
Example: `polyspace-code-prover-nodesktop -sources *file_name* -unit-by-unit -unit-by-unit-common-source definitions.c`

See Also

"Verify files independently (C/C++)"

Related Examples

- "Specify Analysis Options"
- "Run File-by-File Local Verification"
- "Run File-by-File Remote Verification"

Parameters (C)

This option is available only for model-generated code.

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. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: `public`

`public`

The generated `main` initializes all variables except those declared with keywords `static` and `const`.

`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.

Command-Line Information

Parameter: `-variables-written-before-loop`
Value: `none | public | all | custom=`*`variable1`*`[,`*`variable2`*`[,...]]`
Default: `public`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -variables-written-before-loop all`

See Also

"Inputs (C)" on page 1-67 | "Initialization functions (C)" on page 1-69 | "Step functions (C)" on page 1-70 | "Termination functions (C)" on page 1-72

Related Examples

- “Specify Analysis Options”
- “Configure Polyspace Analysis Options and Properties”

More About

- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Inputs (C)

This option is available only for model-generated code.

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 anyvalue allowed by their type. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: `public`

`public`

> The generated `main` initializes all variables except those declared with keywords `static` and `const`.

`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.

Command-Line Information

Parameter: `-variables-written-in-loop`
Value: `none | public | all | custom=`*`variable1`*`[,`*`variable2`*`[,...]]`
Default: `public`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -variables-written-in-loop all`

See Also

"Parameters (C)" on page 1-65 | "Initialization functions (C)" on page 1-69 | "Step functions (C)" on page 1-70 | "Termination functions (C)" on page 1-72

Related Examples

- “Specify Analysis Options”
- “Configure Polyspace Analysis Options and Properties”

More About

- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Initialization functions (C)

This option is available only for model-generated code.

Specify functions that the generated `main` must call before the cyclic code begins. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

No Default

Click ✚ to add a field. Enter function name.

Command-Line Information

Parameter: `-functions-called-before-loop`
No Default
Value: `*function1*[,*function2*[,...]]`
Example: `polyspace-code-prover-nodesktop -sources *file_name* -main-generator -functions-called-before-loop myfunc`

See Also

"Parameters (C)" on page 1-65 | "Inputs (C)" on page 1-67 | "Step functions (C)" on page 1-70 | "Termination functions (C)" on page 1-72

Related Examples

- "Specify Analysis Options"
- "Configure Polyspace Analysis Options and Properties"

More About

- "Recommended Polyspace options for Verifying Generated Code"
- "Main Generation for Model Verification"

Step functions (C)

This option is available only for model-generated code.

Specify functions that the generated `main` must call in each cycle of the cyclic code. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: `unused`

`none`

The generated `main` does not call functions in the cyclic code.

`unused`

The generated `main` calls all functions that are not called elsewhere in the code. In particular, 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. It also does not call inlined functions.

`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.

Tips

- When you select `unused`, the generated `main` does not call a function if it is called elsewhere. However, this rule does not apply to calls through function pointers. The generated `main` calls a function even when it is called elsewhere through a function pointer.
- If you have specified a function for the option **Initialization functions** or **Termination functions**, to call it inside the cyclic code, use `custom` and specify the function name.

Command-Line Information

Parameter: `-functions-called-in-loop`
Value: `none | unused | all | custom=`*`function1`*`[,`*`function2`*`[,...]]`
Default: `unused`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -functions-called-in-loop all`

See Also

"Parameters (C)" on page 1-65 | "Inputs (C)" on page 1-67 | "Initialization functions (C)" on page 1-69 | "Step functions (C)" on page 1-70 | "Termination functions (C)" on page 1-72

Related Examples

- "Specify Analysis Options"
- "Configure Polyspace Analysis Options and Properties"

More About

- "Recommended Polyspace options for Verifying Generated Code"
- "Main Generation for Model Verification"

Termination functions (C)

This option is available only for model-generated code.

Specify functions that the generated `main` must call after the cyclic code ends. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

No Default

Click ✚ to add a field. Enter function name.

Command-Line Information

Parameter: `-functions-called-after-loop`
No Default
Value: `*function1*[,*function2*[,...]]`
Example: `polyspace-code-prover-nodesktop -sources *file_name* -main-generator -functions-called-after-loop myfunc`

See Also

"Parameters (C)" on page 1-65 | "Inputs (C)" on page 1-67 | "Initialization functions (C)" on page 1-69 | "Step functions (C)" on page 1-70

Related Examples

- "Specify Analysis Options"
- "Configure Polyspace Analysis Options and Properties"

More About

- "Recommended Polyspace options for Verifying Generated Code"
- "Main Generation for Model Verification"

Variable/function range setup (C/C++)

Specify range for global variables or function outputs using a **Data Range Specifications** template file. The template file can be either a text or an `XML` file. This option is available on the **Inputs & Stubbing** node in the **Configuration** pane.

Settings

No Default

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

Command-Line Information

Parameter: `-data-range-specifications`
Value: *`file`*
No Default
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-data-range-specifications "C:\DRS\range.txt"`

See Also

"Functions to stub (C)" on page 1-78 | "Ignore default initialization of global variables (C)"

Related Examples

- "Specify Analysis Options"
- "Specify Constraints"
- "Constrain Global Variables"
- "Constrain Stubbed Functions"

More About

- "Constraints"
- "XML File Format for Constraints"

Ignore default initialization of global variables (C)

Specify that Polyspace must not treat global variables as initialized. This option is available on the **Inputs & Stubbing** node in the **Configuration** pane.

Settings

Default: Off

☑ On

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

□ Off

Polyspace considers global variables to be initialized according to ANSI C standards. For instance, the default values are:

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

Tips

- If you initialize a global variable using the generated `main`, Polyspace does not produce a red **Non-initialized variable** error if your code reads the variable before writing to it. The error is not produced even if you turn on the option **Ignore default initialization of global variables**.
- If you initialize a global variables using the generated `main`, Polyspace considers that before the first write operation on the variable in a function, the variable can take any value allowed by its type.

For more information on initializing global variables using the generated `main`, see "Variables to initialize (C)".

Command-Line Information

Parameter: `-no-def-init-glob`
Default: Off

See Also

Non-initialized variable

Related Examples

- “Specify Analysis Options”

No automatic stubbing (C/C++)

Specify that verification must stop if a function is not defined in the source files. This option is available on the **Inputs & Stubbing** node in the **Configuration** pane.

Settings

Default: Off

☑ On

Polyspace displays a list of undefined functions and stops verification.

□ Off

Polyspace stubs all undefined functions.

Tips

Use this option when:

- The code you are verifying must be complete. This option allows you to find functions that are not defined in your source.
- You prefer to stub undefined functions manually.

Command-Line Information

Parameter: `-no-automatic-stubbing`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`filename`* `-no-automatic-stubbing`

See Also

"Functions to stub (C)" | "Functions to stub (C++)" | "No STL stubs (C++)"

Related Examples

- "Specify Analysis Options"
- "Specify Functions to Stub Automatically"
- "Constrain Data with Stubbing"

More About

- “Stubbing Overview”
- “When to Provide Function Stubs”
- “Stubbing Examples”

Functions to stub (C)

Specify functions that you want the software to stub. This option is available on the **Inputs & Stubbing** node in the **Configuration** pane.

Settings

No Default

Click ✚ to add a field. Enter function name.

Tips

If you do not want to review checks in a certain function, you can stub the function. However, Polyspace makes certain assumptions about the arguments and return values of stubbed functions. The assumptions can affect the number of checks in the rest of the code. For example, the software considers that the return values assume the full range allowed by the return type. For more information, see “Assumptions About Stubbed Functions”.

You can specify external constraints on the arguments and return values of stubbed functions. See “Constrain Stubbed Functions”.

Command-Line Information

Parameter: `-functions-to-stub`
No Default
Value: `function1[,function2[,...]]`
Example: `polyspace-code-prover-nodesktop -sources file_name -functions-to-stub function_1,function_2`

See Also

“No automatic stubbing (C/C++)” | “Variable/function range setup (C/C++)” on page 1-73 | “Functions to stub (C++)”

Related Examples

- “Specify Analysis Options”

- “Specify Functions to Stub Automatically”
- “Constrain Data with Stubbing”

More About

- “Stubbing Overview”
- “When to Provide Function Stubs”
- “Stubbing Examples”

Respect types in fields (C/C++)

Specify that structure fields not declared initially as pointers will not be cast to pointers later. This option is available on the **Verification Assumptions** node in the **Configuration** pane.

Settings

Default: Off

☑ 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

`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.
	`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 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

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

See Also

Polyspace Analysis Options

"Respect types in global variables (C/C++)"

Polyspace Results

Non-initialized local variable

Related Examples

- "Specify Analysis Options"

Respect types in global variables (C/C++)

Specify that global variables not declared initially as pointers will not be cast to pointers later. This option is available on the **Verification Assumptions** node in the **Configuration** pane.

Settings

Default: Off

☑ On

The verification assumes that global variables not declared initially as pointers will not be cast to pointers later.

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.

□ Off

The verification assumes that global variables can be cast to pointers even when they are not declared as pointers.

Command-Line Information

Parameter: `-respect-types-in-globals`

Default: Off

See Also

Polyspace Analysis Options

“Respect types in fields (C/C++)”

Polyspace Results

Non-initialized local variable

Related Examples

- “Specify Analysis Options”

Ignore float rounding (C/C++)

Specify that operations involving `float` and `double` variables do not involve rounding. This option is available on the **Verification Assumptions** node in the **Configuration** pane.

Settings

Default: Off

☑ On

The verification considers that operations involving `float` and `double` variables do not involve rounding.

□ Off

The verification assumes that results of operations involving `float` and `double` are rounded to the nearest value according to the IEEE 754 standard:

- Simple precision on 32-bit targets
- Double precision on 64-bit targets

Command-Line Information

Parameter: `-ignore-float-rounding`
Default: Off

Related Examples

- "Specify Analysis Options"

Green absolute address checks (C/C++)

Specify that absolute addresses in your code are valid addresses. This option is available on the **Verification Assumptions** node in the **Configuration** pane.

Settings

Default: Off

☑ On

The verification assumes that the absolute addresses in your code are valid.

□ Off

The verification generates an orange `Absolute Address` check when an absolute address is assigned to a pointer. The orange check occurs because the software does not have information about the absolute address and cannot verify, for example, the validity of the address and the availability of memory.

Tips

Even if you use this option, you cannot assign an absolute address to a pointer and perform pointer arithmetic using the pointer. As soon as you perform pointer arithmetic, Polyspace cannot verify the validity of the next dereference using this pointer

Command-Line Information

Parameter: `-green-absolute-address-checks`
Default: Off

See Also

Polyspace Results
Absolute address

Related Examples

- "Specify Analysis Options"

Ignore overflowing computations on constants (C/C++)

Specify that the verification must allow overflow in computations involving constants. For instance, `char x = 0xff;` causes an overflow according to the ANSI C standard. However, if you use this option, Polyspace considers that this statement is equivalent to `char x = -1;`. This option is available on the **Check Behavior** node in the **Configuration** pane.

Settings

Default: Off

☑ On

The verification allows overflows in computations involving constants.

□ Off

If an overflow occurs in computations involving constants, the verification generates an `Overflow` error.

Tips

- This option applies to computations involving compile-time constants only. For instance, the statement `char x = (rand() ? 0xFF:0xFE);` causes an `Overflow` error irrespective of whether the option is used because the value of `x` is not known at compile-time.

Command-Line Information

Parameter: `-ignore-constant-overflows`
Default: Off

See Also

Polyspace Results

Overflow

Related Examples

- “Specify Analysis Options”

Allow negative operand for left shifts (C/C++)

Specify that the verification must allow shift operations on a negative number. Unless you use this option, following ANSI C standard, the verification generates an error for the shift operations. This option is available on the **Check Behavior** node in the **Configuration** pane.

Settings

Default: Off

☑ On

The verification allows shift operations on a negative number, for instance, `-2 << 2`.

□ Off

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

See Also

Polyspace Results

Shift operations

Related Examples

- “Specify Analysis Options”

Detect overflows (C/C++)

Specify integer overflows to check for. This option is available on the **Check Behavior** node in the **Configuration** pane.

Settings

Default: `signed`

`signed`

The verification checks for overflows in computations involving signed integers alone. This behavior conforms to the ANSI C (ISO® C++) standard.

`signed-and-unsigned`

The verification checks for overflows in all integer computations. This behavior is stricter than the ANSI C (ISO C++) standard.

`none`

The verification does not check for integer overflows. If a computed value exceeds the range of its type, the value is wrapped. For instance, in the following code, `x` is wrapped to 0 after the sum.

```
unsigned char x;
x = 255;
x = x+1;
```

Tips

- Following an overflow, unless you select `none`, Polyspace can either wrap the result or restrict it to its extremum value. Use **Overflow computation mode** to specify how the verification handles results of an overflow.
- Use the option `signed-and-unsigned` if you are computing the size of a buffer from `unsigned` integers. Using this option helps you detect an overflow at the buffer computation stage. Otherwise, you might see an error later due to insufficient buffer.
- If you use the option `signed-and-unsigned`, Polyspace does not produce an overflow error on bitwise NOT operations if you cast the result of the operation back to the operand type. For instance, Polyspace does not produce an overflow error on `(uint8_t)(~var)` where `var` is of type `uint8_t`.

Command-Line Information

Parameter: `-scalar-overflows-checks`
Value: `signed | signed-and-unsigned | none`
Default: `signed`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-scalar-overflows-checks signed`

See Also

Polyspace Analysis Options

"Overflow computation mode (C/C++)"

Polyspace Results

Overflow

Related Examples

- "Specify Analysis Options"
- "Detect Overflows in Buffer Size Computation"

Detect Overflows in Buffer Size Computation

If you are computing the size of a buffer from `unsigned` integers, for the **Detect overflows** option, use `signed-and-unsigned`. Using this option helps you detect an overflow at the buffer computation stage. Otherwise, you might see an error later due to insufficient buffer. This option is available on the **Check Behavior** node in the **Configuration** pane.

For this example, save the following C code in a file `display.c`:

```
#include <stdlib.h>
#include <stdio.h>

int get_value(void);

void display(unsigned int num_items) {
 int *array;
 array = (int *) malloc(num_items * sizeof(int)); // overflow error
  if (array) {
    for (unsigned int ctr = 0; ctr < num_items; ctr++)  {
      array[ctr] = get_value();
    }
    for (unsigned int ctr = 0; ctr < num_items; ctr++)  {
      printf("Value is %d.\n", ctr, array[ctr]);
    }
    free(array);
  }
}

void main() {
  display(33000);
}
```

1. Create a Polyspace project and add `display.c` to the project.
2. On the **Configuration** pane, select the following options:

   - **Target & Compiler**: From the **Target processor type** drop-down list, select a type with 16-bit `int` such as `c167`.
   - **Check Behavior**: From the **Detect overflows** drop-down list, select `signed`.
3. Run the verification and open the results.

Polyspace detects an orange **Illegally dereferenced pointer** error on the line `array[ctr] = get_value()` and a red **Non-terminating loop** error on the `for` loop.

This error follows from an earlier error. For a 16-bit `int`, there is an overflow on the computation `num_items * sizeof(int)`. Polyspace does not detect the overflow because it occurs in computation with `unsigned` integers. Instead Polyspace wraps the result of the computation causing the **Illegally dereferenced pointer** error later.

4. From the **Detect overflows** drop-down list, select `signed-and-unsigned`.
5. Polyspace detects a red **Overflow** error in the computation `num_items * sizeof(int)`.

See Also

Polyspace Analysis Options

"Detect overflows (C/C++)"

Polyspace Results

Overflow | Illegally dereferenced pointer

Overflow computation mode (C/C++)

Specify whether Polyspace must wrap the result of an integer overflow or restrict it to its extremum value. This option is available on the **Check Behavior** node in the **Configuration** pane.

Settings

Default: `truncate-on-error`

`truncate-on-error`

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. However, 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.

`wrap-around`

Polyspace analyzes the remaining code in the current scope even after a red integer **Overflow**. However, Polyspace wraps the result of the overflow. For instance, if you choose this option:

- In the following code, after the red **Overflow**, Polyspace considers that `i` has a value -2^{31}.

```
#include<stdio.h>

void main() {
 int i=1;
 i = i << 30;
 i = i *2;
 printf("%d",i);
}
```

- In the following code, before the orange **Overflow**, `i` has values in the range `[1..2`^{31}`-1]`. But, after the orange **Overflow**, Polyspace considers that `i` has even values in the range `[-2`^{31}`..2] or [2..2`^{31}`-2]`.

```
#include<stdio.h>
int getVal();

void main() {
 int i=getVal();
 if(i>0) {
  i = i*2;
  printf("%d",i);
 }
}
```

Command-Line Information

Parameter: `-scalar-overflows-behavior`
Value: `wrap-around | truncate-on-error`
Default: `truncate-on-error`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-scalar-overflows-behavior wrap-around`

See Also

Polyspace Analysis Options

"Detect overflows (C/C++)"

Polyspace Results

Overflow

Related Examples

- "Specify Analysis Options"

Enable pointer arithmetic across fields (C)

Specify that a pointer assigned to a structure field can point outside its bounds as long as it points within the structure. This option is available on the **Check Behavior** node in the **Configuration** pane.

Settings

Default: Off

☑ 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

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**.

Command-Line Information

Parameter: `-allow-ptr-arith-on-struct`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-allow-ptr-arith-on-struct`

See Also

Polyspace Analysis Options

"Allow incomplete or partial allocation of structures (C)"

Polyspace Results

Illegally dereferenced pointer

Related Examples

- "Specify Analysis Options"

Allow incomplete or partial allocation of structures (C)

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 option is available on the **Check Behavior** node in the **Configuration** pane.

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.

Settings

Default: Off

☑ 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

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

- The verification also allows partial allocation of structures when you select **Enable pointer arithmetic across fields** or **Precision** > **Retype variables of pointer types**.
- 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.

Command-Line Information

Parameter: -size-in-bytes
Default: Off

Example: `polyspace-code-prover-nodesktop -sources file_name -size-in-bytes`

See Also

Polyspace Analysis Options

"Enable pointer arithmetic across fields (C)"

Polyspace Results

Illegally dereferenced pointer

Related Examples

- "Specify Analysis Options"

Permissive function pointer calls (C)

Specify that the verification must allow function pointer calls where the type of the function pointer does not match the type of the function. This option is available on the **Check Behavior** node in the **Configuration** pane.

Settings

Default: Off

☑ 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*)`.

□ Off

The verification must require that the argument and return types of a function pointer and the function it calls are identical.

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.

Command-Line Information

Parameter: `-permissive-function-pointer`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-permissive-function-pointer`

Related Examples

- “Specify Analysis Options”

Detect uncalled functions (C/C++)

Detect functions that are not called directly or indirectly from `main` or another entry point during run-time. This option is available on the **Check Behavior** node in the **Configuration** pane.

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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-uncalled-function-checks all`

See Also

Polyspace Results

Function not called | Function not reachable

Related Examples

- "Specify Analysis Options"

- “Review Gray Checks”
- “Review and Fix Function Not Called Checks”
- “Review and Fix Function Not Reachable Checks”

Precision level (C/C++)

Specify the precision level that the verification must use. 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. This option is available on the **Precision** node in the **Configuration** pane.

Settings

Default: 2

0

This option corresponds to a static interval verification.

1

This option corresponds to a complex polyhedron model of domain values.

2

This option corresponds to more complex algorithms closely modelling domain values. The algorithms combine both complex polyhedrons and integer lattices.

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.

Command-Line Information

Parameter: `-O0 | -O1 | -O2 | -O3`
Default: `-O2`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-O1`

See Also

"Verification level (C)"

Related Examples

- “Specify Analysis Options”
- “Improve Verification Precision”

Verification level (C)

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. This option is available on the **Precision** node in the **Configuration** pane.

Settings

Default: `Software Safety Analysis level 2`

`C Source Compliance Checking`

Polyspace completes coding rules checking at the end of the compilation phase.

`Software Safety Analysis level 0`

The verification process runs once on your source code.

`Software Safety Analysis level 1`

The verification process runs twice on your source code.

`Software Safety Analysis level 2`

The verification process runs thrice on your source code. Use this option for most accurate results in reasonable time.

`Software Safety Analysis level 3`

The verification process runs four times on your source code.

`Software Safety Analysis level 4`

The verification process runs five times on your source code.

`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.

Difference between Level 0 and 1

The following example illustrates the difference between `Software Safety Analysis level 0` and `Software Safety Analysis level 1`:

<table>
<tr><th>Software Safety Analysis Level 0</th><th>Software Safety Analysis Level 1</th></tr>
<tr><td><pre>
#include <stdlib.h>

void ratio (float x, float *y)
{
 *y=(abs(x-*y))/(x+*y);
}

void level1 (float x,
     float y, float *t)
{ float v;
 v = y;
 ratio (x, &y);
 *t = 1.0/(v - 2.0 * x);
}

float level2(float v)
{
 float t;
 t = v;
 level1(0.0, 1.0, &t);
 return t;
}

void main(void)
{
 float r,d;
 d= level2(1.0);
 r = 1.0 / (2.0 - d);
}
</pre></td><td><pre>
#include <stdlib.h>

void ratio (float x, float *y)
{
 *y=(abs(x-*y))/(x+*y);
}

void level1 (float x,
     float y, float *t)
{ float v;
 v = y;
 ratio (x, &y);
 *t = 1.0/(v - 2.0 * x);
}

float level2(float v)
{
 float t;
 t = v;
 level1(0.0, 1.0, &t);
 return t;
}

void main(void)
{
 float r,d;
 d= level2(1.0);
 r = 1.0 / (2.0 - d);
}
</pre></td></tr>
</table>

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.

- 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.

Dependency

You cannot use the `C Source Compliance Checking` setting for remote verification.

Command-Line Information

Parameter: `-to`
Value: `c-compile | pass0 | pass1 | pass2 | pass3 | pass4 | other`
Default: `pass2`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-to pass2`

Related Examples

- "Specify Analysis Options"
- "Improve Verification Precision"

Verification time limit (C/C++)

Specify a time limit for the verification in hours. If the verification does not complete within that limit, it stops. This option is available on the **Precision** node in the **Configuration** pane.

Settings

Enter the time in hours. For fractions of an hour, specify decimal form.

Command-Line Information

Parameter: `-timeout`
Value: *`time`*
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-timeout 5.75`

Related Examples

- "Specify Analysis Options"
- "Improve Verification Precision"

Retype variables of pointer types (C)

Specify that the verification must allow pointers to be cast from one type to another. If you select this option, the verification replaces the original type of the pointer by its new type. This option is available on the **Precision** node in the **Configuration** pane.

Settings

Default: Off

☑ On

The verification allows pointers to be cast from one type to another. It replaces the original type of the pointer by its new type. For instance, using this option, the software produces a green check on the `assert` statement in the following code:

```
struct A {int a; char b;} s = {1,2};
char *tmp = (char *)&s;
struct A *pa = (struct A*)tmp;
assert((pa->a == 1) && (pa->b == 2));
```

□ Off

The verification retains the declaration type of a pointer even when it is recast.

Command-Line Information

Parameter: `-retype-pointer`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-retype-pointer`

See Also

"Enable pointer arithmetic across fields (C)" | "Allow incomplete or partial allocation of structures (C)"

Related Examples

- "Specify Analysis Options"

Retype symbols of integer types (C)

Specify that the verification must allow integers to be cast to pointers. This option is available on the **Precision** node in the **Configuration** pane.

Settings

Default: Off

☑ On

The verification allows integers to be cast to pointers. For instance, using this option, the software can prove the `assert` statements in the following code:

```
void function(void)
 {
  struct S1 {
   int x;
   int y;
   int z;
   char t;
  } s1 = {1,2,3,4};
 int addr;
 addr = (int)(&s1);
 assert(((struct S1 *)addr)->y == 2);
}
```

□ Off

The verification does not allow integers to be cast to pointers.

Dependencies

This option:

- Automatically enables **Check Behavior** > **Allow incomplete or partial allocation of structures**.
- Has no effect on global integers if you select the option **Verification Assumptions** > **Respect types in global variables**.
- Has no effect on integers that are structure fields if you select the option **Verification Assumptions** > **Respect types in fields**.

Tips

- Use this option for:
  - Code with memory mapping
  - Code close to the communication layer API – When your code contains low level drivers, it tends to perform generic pointer casts using `(void *)`.
- If you set this option:
  - Some of the `Illegally dereferenced pointer` checks can change
  - Some of the `Non-initialized variable` checks can change to `Non-initialized pointer` checks.

Command-Line Information

Parameter: `-retype-int-pointer`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-retype-int-pointer`

See Also

Illegally dereferenced pointer

Related Examples

- "Specify Analysis Options"

Sensitivity context (C/C++)

Specify that the software must store call context information during verification. If a function contains a red and green check in the same line for two different invocations, both checks will be displayed. This option is available on the **Precision** node in the **Configuration** pane.

Settings

Default: `none`

`none`

The software does not store call context information for functions.

`auto`

The software stores call context information for checks in the following functions:

- 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. Click ✚ to enter the name of a function.

Command-Line Information

Parameter: `-context-sensitivity`
Value: `*function1*[,*function2*,...]`
Default: `none`
Example: `polyspace-code-prover-nodesktop -sources *file_name* -context-sensitivity myFunc1,myFunc2`

To allow the software to decide which functions receive call context storage, use the option `-context-sensitivity-auto`.

Related Examples

- “Specify Analysis Options”

- “Identify Function Call Causing Orange Check”

Improve precision of interprocedural analysis (C/C++)

Use this option to propagate greater information about function arguments into the called function. This option is available on the **Precision** node in the **Configuration** pane.

Settings

Default: Off

Enter a positive integer to turn on this option. Entering a higher value leads to greater number of proven results, but also increases verification time.

Tips

Using this option, you can increase the verification time enormously within a certain pass. Therefore, use this option only when you have less than 1000 lines of code.

Command-Line Information

Parameter: `-path-sensitivity-delta`
Value: Positive integer
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-path-sensitivity-delta 1`

Related Examples

- “Specify Analysis Options”
- “Improve Verification Precision”

Specific precision (C)

Specify source files that you want to verify at a **Precision level** higher than that for the entire verification. This option is available on the **Precision** node in the **Configuration** pane.

Settings

Default: All files are verified with the precision you specified using **Precision** > **Precision level**.

Click ✚ to enter the name of a file and the corresponding precision level.

Command-Line Information

Parameter: `-modules-precision`
Value: *`file`*`:O0` | *`file`*`:O1` | *`file`*`:O2` | *`file`*`:O3`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-O1 -modules-precision My_File.c:O2`

See Also

"Precision level (C/C++)"

Related Examples

- "Specify Analysis Options"
- "Improve Verification Precision"

Optimize large static initializers (C)

Specify that the verification must approximate statically initialized `int`, `float` and `char` arrays if required. If you do not specify this option, for static initialization of large arrays, scaling problems can occur during the compilation phase. This option is available on the **Scaling** node in the **Configuration** pane.

Settings

Default: Off

☑ On

The verification approximates statically initialized `int`, `float` and `char` arrays if required. Using this option can speed up verification, but can decrease precision for some applications.

□ Off

The verification does not approximate statically initialized `int`, `float` and `char` arrays.

Command-Line Information

Parameter: `-no-fold`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-no-fold`

Related Examples

- "Specify Analysis Options"

Reduce task complexity (C)

Specify that the verification must use a slightly less precise model than default for interaction between tasks. This option is available on the **Scaling** node in the **Configuration** pane.

Settings

Default: Off

☑ On

The verification uses a slightly less precise model than default for interaction between tasks. Using this option, you can speed up verification, but have greater number of unproven results. There is also a loss of precision when variables shared between tasks are read through pointers.

□ Off

The verification uses the default model for interaction between tasks.

Command-Line Information

Parameter: `-lightweight-thread-model`
Default: Off

See Also

"Entry points (C/C++)"

Related Examples

- "Specify Analysis Options"
- "Reduce Verification Time"

Inline (C)

Specify the functions that the verification must clone for every function call. For instance, if you specify the function `func` for inlining and `func` is called twice, the software creates two copies of `func` for verification. The copies are named using the convention `func_pst_inlined_`*`ver`* where *`ver`* is the version number. This option is available on the **Scaling** node in the **Configuration** pane.

Settings

No Default

Click ➕ to enter function name.

Tips

- Use this option to identify the cause of a **Non-terminating call** error.
  - **Situation:** Sometimes, a red **Non-terminating call** check can appear on a function call though a red check does not appear in the function body. The function body represents all calls to the function. Therefore, if some calls to a function do not cause an error, an orange check appears in the function body.
  - **Action:** If you use this option, for every function call, there is a corresponding function body. Therefore, you can trace a red check on a function call to a red check in the function body.
- 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**.

Command-Line Information

Parameter: `-inline`
Value: `*function1*[,*function2*[,...]]`
No Default
Example: `polyspace-code-prover-nodesktop -sources *file_name* -inline func1,func2`

Related Examples

- "Specify Analysis Options"
- "Reduce Procedure Complexity"

Depth of verification inside structures (C/C++)

Specify a limit to the depth of analysis for nested structures. This option is available on the **Scaling** node in the **Configuration** pane.

Settings

Default: Full depth of nested structures is analysed.

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.

Command-Line Information

Parameter: `-k-limiting`
Value: *`positive integer`*
Default: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-k-limiting 1`

Related Examples

- "Specify Analysis Options"

Generate report (C/C++)

Specify whether to generate a report after the analysis. This option is available on the **Reporting** node in the **Configuration** pane.

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.

Settings

Default: Off

☑ On

Polyspace generates an analysis report using the template and format you specify.

□ Off

Polyspace does not generate an analysis report. You can still view your results in the Polyspace interface.

Tips

- To generate a report *after* an analysis is complete, select **Reporting** > **Run Report**. Alternatively, at the command line, use the command `polyspace-report-generator` with the options `-template` and `-format`.

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

"Report template (C/C++)" | "Output format (C/C++)"

Related Examples

- "Specify Analysis Options"

- “Generate Report”

Report template (C/C++)

Specify template for generating analysis report. This option is available on the **Reporting** node in the **Configuration** pane.

`.rpt` files for the report templates are available in the folder *`MATLAB_Install`*`\polyspace\toolbox\psrptgen\templates\`.

Settings

Default: `Developer`

`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.

`CodeMetrics`

The report contains a summary of code metrics, followed by the complete metrics for an application.

`CodingRules`

For C code, the report lists information about compliance with:

- MISRA C rules
- MISRA AC AGC rules
- Custom coding rules

For C++ code, the report lists information about compliance with:

- MISRA C++ rules
- JSF® C++ rules
- Custom coding rules

This report also contains the Polyspace configuration settings for the 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

The report also contains the Polyspace configuration settings for the analysis.

`DeveloperReview`

The report lists the same information as the `Developer` report. However, the reviewed results are sorted by review classification 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 for the analysis.

`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.

`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.

`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.

Dependencies

- This option is enabled only if you select the **Generate report** box.
- The templates `SoftwareQualityObjectives` and `SoftwareQualityObjectives_Summary` are available only if you generate a report from results downloaded from the Polyspace Metrics web dashboard. To generate these reports:

  1. Download results from the Polyspace Metrics dashboard.
  2. Select **Reporting** > **Run Report**.
  3. Select the template that you want.

Command-Line Information

Parameter: `-report-template`
Value: `Full path to` *`template`*`.rpt`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-report-template` *`matlabroot`*`\polyspace\toolbox\psrptgen\templates\Developer.rpt`

Here, *`matlabroot`* is the MATLAB® installation folder such as `C:\Program Files\MATLAB\R2015a`.

See Also

“Generate report (C/C++)” | “Output format (C/C++)”

Related Examples

- “Specify Analysis Options”
- “Generate Report”

- “Customize Report Templates”

Output format (C/C++)

Specify output format of generated report. This option is available on the **Reporting** node in the **Configuration** pane.

Settings

Default: `RTF`

`RTF`

Generate report in `.rtf` format

`HTML`

Generate report in `.html` format

`PDF`

Generate report in `.pdf` format

`Word`

Generate report in `.doc` format. Not available on UNIX® platforms.

`XML`

Generate report in `.xml` format.

Tips

- You must have Microsoft Office installed to view `.rtf` format reports containing graphics, such as the `Quality` report.
- If the table of contents or graphics in a `.doc` 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

This option is enabled only if you select the **Generate report** box.

Command-Line Information

Parameter: `-report-output-format`

Value: `RTF | HTML | PDF | Word | XML`
Default: `RTF`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-report-output-format pdf`

See Also

"Generate report (C/C++)" | "Report template (C/C++)"

Related Examples

- "Specify Analysis Options"
- "Generate Report"
- "Customize Report Templates"

Batch (C/C++)

Enable or disable batch remote analysis. This option is available on the **Distributed Computing** node in the **Configuration** pane.

For batch remote analysis, you need:

- Polyspace and MATLAB Distributed Computing Server™ on the cluster
- MATLAB, Polyspace and Parallel Computing Toolbox™ on your local computer

Settings

Default: Off

☑ 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**.
- On the DOS or UNIX command line, use the `polyspace-jobs-manager` command. For more information, see “Run Remote Analysis at Command Line”.
- On the MATLAB command line, use the polyspaceJobsManager function.

After the analysis, you might have to manually download the results from the cluster.

□ Off

Do not run batch analysis on a remote computer.

Dependency

You cannot use **Batch** mode with the **Verification Level** options `C source compliance checking` or `C++ source compliance checking`.

Command-Line Information

To run a remote verification 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: `polyspace-code-prover-nodesktop -batch -scheduler NodeHost`
`polyspace-code-prover-nodesktop -batch -scheduler MJSName@NodeHost`

See Also

"Add to results repository (C/C++)" on page 1-131 | `-scheduler`

Related Examples

- "Specify Analysis Options"
- "Set Up Server for Remote Verification and Analysis"
- "Run Remote Verification"
- "Run Remote Analysis at Command Line"

Add to results repository (C/C++)

Specify upload of analysis results to the Polyspace Metrics results repository, allowing Web-based reporting of results and code metrics. This option is available on the **Distributed Computing** node in the **Configuration** pane.

Settings

Default: Off

☑ 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.

□ Off

Analysis results are stored locally.

Dependency

This option is only available for remote verifications. For local verification, you can manually upload your results to Polyspace Metrics by right-clicking on your results file and selecting **Upload to Metrics**.

Command-Line Information

Parameter: `-add-to-results-repository`
Default: Off
Example: `polyspace-code-prover-nodesktop -batch -scheduler NodeHost -add-to-results-repository`

See Also

"Set Up Server for Remote Verification and Analysis" | "Set Up Polyspace Metrics" | "Generate Code Quality Metrics" | "Batch (C/C++)" on page 1-129

Related Examples

- "Run Remote Verification"
- "Generate Code Quality Metrics"

Command/script to apply after the end of the code verification (C/C++)

Specify a command or script to be executed after the verification. This option is available on the **Advanced Settings** node in the **Configuration** pane.

Settings

No Default

Enter full path to the command or script, or click 📁 to navigate to the location of the command or script. For example, you can enter the path to a script that sends an email. After the verification, this script will be executed.

Command-Line Information

Parameter: `-post-analysis-command`
Value: Path to executable file or command in quotes
No Default
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-post-analysis-command `pwd`/send_email`

Related Examples

- "Specify Analysis Options"

Automatic Orange Tester (C)

Specify that the Automatic Orange Tester must be executed at the end of the verification. This option is available on the **Advanced Settings** node in the **Configuration** pane.

You must select this option before verification if you want to run the Automatic Orange Tester after verification. During verification, Polyspace generates additional source code that tests each orange check for run-time errors. The software compiles this instrumented code. When you run the Automatic Orange Tester later, the software tests the resulting binary code.

Settings

Default: Off

☑ On

After verification, when you run the Automatic Orange Tester, Polyspace creates tests for unproven code and runs them.

□ Off

You cannot launch the Automatic Orange Tester after verification.

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 **Variable/function range setup** under **Inputs & Stubbing**.

Command-Line Information

Parameter: `-automatic-orange-tester`
Default: Off
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-automatic-orange-tester`

See Also

"Number of automatic tests (C)" | "Maximum loop iterations (C)" | "Maximum test time (C)"

Related Examples

- "Specify Analysis Options"
- "Test Orange Checks for Run-Time Errors"

More About

- "Limitations of Automatic Orange Tester"

Number of automatic tests (C)

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. This option is available on the **Advanced Settings** node in the **Configuration** pane.

Settings

Default: 500

Enter number of tests up to a maximum of 100,000.

Dependencies

This option is enabled only if you select the **Automatic Orange Tester** box.

Command-Line Information

Parameter: `-automatic-orange-tester-tests-number`
Value: *`positive integer`*
Default: 500
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-automatic-orange-tester -automatic-orange-tester-tests-number 500`

See Also

"Automatic Orange Tester (C)" | "Maximum loop iterations (C)" | "Maximum test time (C)"

Related Examples

- "Specify Analysis Options"
- "Test Orange Checks for Run-Time Errors"

Maximum loop iterations (C)

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. This option is available on the **Advanced Settings** node in the **Configuration** pane.

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 select the **Automatic Orange Tester** box.

Command-Line Information

Parameter: `-automatic-orange-tester-loop-max-iteration`
Value: *`positive integer`*
Default: 1000
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-automatic-orange-tester -automatic-orange-tester-loop-max-iteration 500`

See Also

"Automatic Orange Tester (C)" | "Number of automatic tests (C)" | "Maximum test time (C)"

Related Examples

- "Specify Analysis Options"
- "Test Orange Checks for Run-Time Errors"

Maximum test time (C)

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. This option is available on the **Advanced Settings** node in the **Configuration** pane.

Settings

Default: 5

Enter time in seconds. The maximum value that the software supports is 60.

Dependencies

This option is enabled only if you select the **Automatic Orange Tester** box.

Command-Line Information

Parameter: `-automatic-orange-tester-timeout`
Value: *`time`*
Default: 5
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-automatic-orange-tester -automatic-orange-tester-test-timeout 10`

See Also

"Automatic Orange Tester (C)" | "Number of automatic tests (C)" | "Maximum loop iterations (C)"

Related Examples

- "Specify Analysis Options"
- "Test Orange Checks for Run-Time Errors"

Other (C)

In this section...

Specify special options for C verification, which are provided by MathWorks if required. This option is available on the **Advanced Settings** node in the **Configuration** pane.

`-extra-flags`

No Default

Example:

```
 polyspace-code-prover-nodesktop -extra-flags -param1 -extra-flags -
param2 \

  -extra-flags 10 ...
```

`-c-extra-flags`

No Default

Example:

```
polyspace-code-prover-nodesktop -c-extra-flags -param1 -c-extra-
flags -param2 -c-extra-flags 10
```

`-cfe-extra-flags`

No Default

Example:

```
polyspace-code-prover-nodesktop -cfe-extra-flags -param1 -cfe-extra-
flags -param2
```

-il-extra-flags

No Default

Example:

```
polyspace-code-prover-nodesktop -il-extra-flags -param1 -il-extra-
flags -param2 -il-extra-flags 10
```

Option Descriptions specific to C++ Code

Target processor type (C++)

Specify the target processor type. This option is available on the **Target & Compiler** node in the **Configuration** pane.

Specifying the target processor type informs Polyspace of the size of fundamental data types and of the endianess of the target machine. You can analyze code intended for an unlisted processor type using one of the listed processor types, if they share common data properties.

Settings

Default: `i386`

You can modify some default attributes by selecting the browse button to the right of the **Target processor type** drop-down menu. The optional settings for each target are shown in [brackets] in the table.

Target	char	short	int	long	long long	float	double	long double	ptr	sign of char	endian	align
`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 / ColdFire`[a]	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
`x86_64`	8	16	32	64 [32][b]	64	32	64	128	64	signed	Little	64 [32]
`mcpu... (Advanced)`	8 [16]	8 [16]	16 [32]	32	32 [64]	32	32 [64]	32 [64]	16 [32]	signed	Little	32 [16, 8]

[a] The M68k family (68000, 68020, etc.) includes the “ColdFire” processor

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

[c] `mcpu` is a reconfigurable Micro Controller/Processor Unit target. You can use this type to configure one or more generic targets.

Tips

If your processor is not listed, use a similar processor that shares the same characteristics, or create an `mpcu` generic target processor. If your target processor does not match the characteristics of a processor described above, contact MathWorks technical support for advice.

Command-Line Information

Parameter: `-target`
Value: `i386 | m68k | powerpc | c-167 | x86_64 | mpcu`
Default: `i386`
Example: `polyspace-code-prover-nodesktop -lang cpp -target powerpc`

See Also

"Generic target options (C/C++)" on page 1-9

Related Examples

- "Specify Analysis Options"
- "Modify Predefined Target Processor Attributes"
- "Define Generic Target Processors"

Dialect (C++)

Allow syntax associated with C++ language extensions. This option is available on the **Target & Compiler** node in the **Configuration** pane.

Settings

Default: `none`

`none`

Analysis allows for ISO/IEC 14882:2003 C++ (C++ 2003) syntax.

If you want to allow ISO/IEC 14882:2011 C++ (C++ 2011) syntax, also select **C++ 11 extensions**.

`gnu3.4`

Analysis allows GCC 3.4 dialect syntax.

`gnu4.6`

Analysis allows GCC 4.6 dialect syntax.

`gnu4.7`

Analysis allows GCC 4.7 dialect syntax.

For more information, see “Limitations” on page 2-6.

`gnu4.8`

Analysis allows GCC 4.8 dialect syntax.

`iso`

Analysis allows for ISO/IEC 14882:2003 C++ (C++ 2003) syntax.

If you want to allow ISO/IEC 14882:2011 C++ (C++ 2011) syntax, also select **C++ 11 extensions**.

`cfront2`

Analysis allows for Cfront 2.0 language extensions.

`cfront3`

Analysis allows for Cfront 3.0 language extensions.

`visual`

Analysis allows Visual C++ .NET 2003 syntax.

`visual6`

Analysis allows Visual C++ 6.0 (VC6) syntax.

`visual7.0`

Analysis allows Visual C++ .NET 2002 syntax.

`visual7.1`

Analysis allows Visual C++ .NET 2003 syntax.

`visual8`

Analysis allows Visual C++ 2005 syntax.

`visual9.0`

Analysis allows Visual C++ 2008 syntax.

`visual10`

Analysis allows Visual C++ 2010 syntax.

This option automatically adds the option `-no-stl-stubs`.

`visual11.0`

Analysis allows Visual C++ 2012 syntax.

This option automatically adds the option `-no-stl-stubs`.

Dependencies

This parameter is dependent on the value of **Target operating system**. The dialect options work only with the applicable operating systems. You can use every dialect with the **Target operating system** option, `no-predefined-OS`.

If you enable **Check JSF C++ Rules** with a dialect other than `iso` or `none`, Polyspace cannot completely check some JSF coding rules. For example, AV Rule 8: “All code shall conform to ISO/IEC 14882:2002(E) standard C++.”

Limitations

Polyspace does not support certain aspects of the GNU 4.7 dialect. These limitations can cause compilation errors, incomplete results, or false positives.

- **Priority attributes** — Not supported, ignores priorities and uses standard initialization instead.

Example

```
#include <stdio.h>
struct A{
    int a;
    A():a(1) {
       fprintf(stderr, "A constructor\n");
    }
};

struct B{
    int b;

    B():b(1) {
       fprintf(stderr, "B constructor\n");
    }
};

A a __attribute__((init_priority (100)));
B b __attribute__((init_priority (50)));
```

The expected output from the above code is:

```
B constructor
A constructor
```

However, Polyspace preserves the standard initialization. So the actual output is:

```
A constructor
B constructor
```

Workaround: To use the desired priority, change the order of the declarations to match the desired order.

- **Vector types and attributes** — Not supported.
- **Visibility attributes** — Not supported, ignored.

  This limitation can cause C++ linkage problems in Polyspace Code Prover.

  Workaround: Remove all attributes during preprocessing,

  - At the command line, use the option `-D __attribute__(x)=`.
  - In the Polyspace environment, in **Macros** > **Preprocessor definitions**, add a row: `__attribute__(x)=`.

- **Complex types** — Only floating complex types supported, integral complex types cause an error.
- **Using built-in library function on complex types** — Not supported, stubbed during analysis. Calls to these functions will return variables with full ranges.

  Workaround: To make the analysis more precise, add an include file that defines the functions for complex variables.
- **Computed `goto`** — Not supported.

  This causes an error in Code Prover. To ignore the computed gotos, stub the functions containing the computed gotos:

  - At the command line, use the option `-functions-to-stub` *`funcList`* where *`funcList`* is the list of functions containing the computed gotos.
  - In the Polyspace environment, in the **Inputs & Stubbing** > **Functions to stub** table, use the ➕ button to add a row for each function containing the computed gotos.
- **Nested functions** — Not supported, causes an error.
- **Using built-in library functions on atomic operators** — Not supported, Polyspace stubs the functions. This limitation can cause imprecise results.
- **IEEE floating point library functions** — Limited support, can cause imprecise results.

  This limitation includes `isnan`, `isnanf`, `isnanl`, `isinf`, `isinff`, `isinfl`, `isnormal`, and `isfinite`.

  Workaround: In each of your source files, include a file containing the function definitions or declarations:

  - At the command line, use the option `-include` *`filename`*.
  - In the Polyspace environment, in **Environment Settings** > **Include**, use the ➕ button to add a row for your definition/declaration file.

Command-Line Information

Parameter: `-dialect`

Value: `none | gnu3.4 | gnu4.6 | gnu4.7 | iso | cfront2 | cfront3 | visual | visual6 | visual7.0 | visual7.1 | visual8 | visual9.0 | visual10 | visual11.0`
Default: `none`
Example: `polyspace-code-prover-nodesktop -lang cpp -sources "`*`file1.cpp,file2.cpp`*`" -OS-target Visual -dialect visual7.1`

See Also

"Target operating system (C/C++)" on page 1-4 | "Target processor type (C++)" on page 2-3 | "C++11 Extensions (C++)" on page 2-10 | "Block char16/32_t types (C++)" on page 2-11

Related Examples

- "Verify Keil or IAR Dialects"

More About

- "Supported C++ 2011 Standards"

C++11 Extensions (C++)

Allow for C++11 language extensions. This option is available on the **Target & Compiler** node in the **Configuration** pane.

If your code uses any C++11 language constructs, select this option to allow this syntax during your analysis.

Settings

Default: Off

☐ Off

The analysis does not allow C++11 syntax.

☑ On

The analysis allows C++11 syntax.

Dependencies

You can only select this option when the **Dialect** option is `none`, `gnu4.6`, or `gnu4.7`.

Command-Line Information

Parameter: `-cpp11-extension`
Default: off
Example: `polyspace-code-prover-nodesktop -lang cpp -cpp11-extension`

See Also

"Dialect (C++)" on page 2-5 | "Block char16/32_t types (C++)" on page 2-11

More About

- "Supported C++ 2011 Standards"

Block char16/32_t types (C++)

The analysis does not allow `char16_t` or `char32_t` types. This option is available on the **Target & Compiler** node in the **Configuration** pane.

If you have defined `char16_t` and/or `char32_t` through a `typedef` statement or using includes, this option allows you to turn off the standard Polyspace definition of `char16_t` and `char32_t`.

Settings

Default: Off

□ Off

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

☑ On

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

Dependencies

You can only select this option when the **Dialect** option is either `none` or a `gnu` dialect.

Command-Line Information

Parameter: `-no-uliterals`
Default: off
Example: `polyspace-code-prover-nodesktop -dialect gnu4.7 -lang cpp -cpp11-extension -no-uliterals`

See Also

"Dialect (C++)" on page 2-5 | "C++11 Extensions (C++)" on page 2-10

More About

- "Supported C++ 2011 Standards"

Enum type definition (C++)

Allow the analysis to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition. This option is available on the **Target & Compiler** node in the **Configuration** pane.

When using this option, each enum type is represented by the smallest integral type that can hold all its enumeration values.

Settings

Default: `auto-signed-int-first`

`auto-signed-int-first` On

Uses the first type that can hold all of the enumerator values from the following list:`signed int`, `unsigned int`, `signed long`, `unsigned long`, `signed long long`, `unsigned long long`

`auto-signed-first`

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

`auto-unsigned-first`

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

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

Command-Line Information

Parameter: `-enum-type-definition`
Value: `auto-signed-int-first | auto-signed-first | auto-unsigned-first`
Default: `auto-signed-int-first`
Example: `polyspace-code-prover-nodesktop -lang cpp -enum-type-definition auto-signed-first`

Pack alignment value (C++)

Specify the default packing alignment for an analysis. This option is available on the **Target & Compiler** node in the **Configuration** pane.

If an invalid value is given, analysis will halt and display an error message. with a bad value or if this option is used in non visual mode (**Target operating system** `Visual` or **Dialect** `visual*`).

Settings

Default: 8

- 1
- 2
- 4
- 8
- 16

Dependencies

This analysis option is available only when,

- **Target operating system** is set to `no-predefined-OS` or `Visual`.
- and **Dialect** is set to one of the `visual*` options.

Command-Line Information

Parameter: `-pack-alignment-value`
Value: `1 | 2 | 4 | 8 | 16`
Default: `8`
Example: `polyspace-code-prover-nodesktop -lang cpp -pack-alignment-value 4`

Ignore pragma pack directives (C++)

Specifies C++ `#pragma` packing alignment for structure, union, and class members. This option is available on the **Target & Compiler** node in the **Configuration** pane.

Settings

Default: Off

□ Off

Keeps C++ `#pragma` directives in the analysis

☑ On

Allows C++ `#pragma` directives to be ignored in order to prevent link errors

Analysis will halt and display an error message with a bad value or if this option is used in non visual mode (**Target operating system** `Visual` or **Dialect** `visual*`).

Dependencies

This analysis option is available only when,

- **Target operating system** is set to `no-predefined-OS` or `Visual`.
- and **Dialect** is set to one of the `visual*` options.

Command-Line Information

Parameter: `-ignore-pragma-pack`
Default: Off
Example: `polyspace-code-prover-nodesktop -lang cpp -ignore-pragma-pack`

Support managed extensions (C++)

Visual C++ /FX option allows the partial translation of sources making use of managed extensions to Visual C++ sources without managed extensions. This option is available on the **Target & Compiler** node in the **Configuration** pane.

Settings

Default: Off

□ Off

Do not support managed extensions

☑ On

Allows the analysis of a project containing translated sources obtained by compilation of a Visual project using the /FX Visual option.

Using /FX, the translated files are generated in place of the original ones in the project, but the names are changed from `foo.ext` to `foo.mrg.ext`.

These extensions are currently not taken into account by Polyspace analysis and can be considered as a limitation to analyze this kind of code. Managed files need to be located in the same folder as the original ones and Polyspace software will analyze managed files instead of the original ones without intrusion, and will permit you to remove part of the limitations due to specific extensions.

Dependencies

This analysis option is available only when,

- **Target operating system** is set to `no-predefined-OS` or `Visual`.
- and **Dialect** is set to one of the `visual*` options.

Command-Line Information

Parameter: `-support-FX-option-results`
Default: off
Example: `polyspace-code-prover-nodesktop -lang cpp -OS-target Visual -support-FX-option-results`

Import folder (C++)

Specifies a single directory to be included by *`#import`* directive. This option is available on the **Target & Compiler** node in the **Configuration** pane.

Settings

No default

Give the location of `*.tlh` files generated by a Visual Studio compiler when encountering `#import` directive on `*.tlb` files.

Dependencies

This analysis option is available only when,

- **Target operating system** is set to `no-predefined-OS` or `Visual`.
- and **Dialect** is set to one of the `visual*` options.

Command-Line Information

Parameter: `-import-dir`
Value: File location
Example: `polyspace-code-prover-nodesktop -OS-target Visual -dialect visual8 -import-dir /com1/inc`

Management of scope of 'for loop' variable index (C++)

Specify the scope of the index variable declared within a `for` loop. This option is available on the **Target & Compiler** node in the **Configuration** pane.

For example:

```
for (int index=0; ...){};
index++; // At this point, index variable is usable (out) or not (in)
```

This option allows the default behavior implied by the Polyspace `-dialect` option to be overridden.

This option is equivalent to the Visual C++ options `/Zc:forScope` and `Zc:forScope-`.

Settings

Default: `defined-by-dialect`

`defined-by-dialect`

Default behavior specified by selected dialect

`out`

The index variable is usable outside the scope of the for loop.

Default behavior for the dialect options `cfront2`, `crfront3`, `visual6`, `visual7` and `visual 7.1`

`in`

The index variable is not usable outside the scope of the for loop.

Default behavior for all other dialects, including `visual8`. The C++ standard specifies that the index is treated as `in`.

Command-Line Information

Parameter: `-for-loop-index-scope`
Value: `defined-by-dialect | out | in`
Default: `defined-by-dialect`
Example: `polyspace-code-prover-nodesktop -lang cpp -for-loop-index-scope in`

Management of wchar_t (C++)

Specify how to treat `wchar_t`. This option is available on the **Target & Compiler** node in the **Configuration** pane.

This option is equivalent to the Visual C++ options `/Zc:wchar` and `/Zc:wchar-`.

Settings

Default: `defined-by-dialect`

`defined-by-dialect`

Default behavior specified by selected dialect

`typedef`

Use according to `typedef` statement specified by Microsoft Visual C++ `6.0/7.0/7.1` dialects.

Default behavior for the dialect options `visual6`, `visual7.0` and `visual7.1`

`keyword`

Use as a keyword as given by the C++ standard

Default behavior for all other dialects, including `visual8`.

Command-Line Information

Parameter: `-wchar-t-is`
Value: `defined-by-dialect | typedef | keyword`
Default: `defined-by-dialect`
Example: `polyspace-code-prover-nodesktop -for-loop-index-scope keyword`

Set wchar_t to unsigned long (C++)

Specify the underlying type of `wchar_t` to be unsigned long. This option is available on the **Target & Compiler** node in the **Configuration** pane.

Settings

Default: Off

□ Off

Use the default underlying type of `wchar_t` as defined by the dialect or the **Management of wchar_t** option.

☑ On

Set the type of `size_t` to unsigned long, as defined in the C++ standard.

For example, `sizeof(L'W')` will have the value of `sizeof(unsigned long)` and the `wchar_t` field will be aligned in the same way as the unsigned long field.

Command-Line Information

Parameter: `-wchar-t-is-unsigned-long`
Default: off
Example: `polyspace-code-prover-nodesktop -lang cpp -wchar-t-is-unsigned-long`

Set size_t to unsigned long (C++)

Force the underlying type of `size_t` to be `unsigned long`. This option is available on the **Target & Compiler** node in the **Configuration** pane. If you use this option, you can only redefine `size_t` with a `typedef` statement to `unsigned long`.

For example, Polyspace applies the following `typedef` statement because the type is `unsigned long`:

```
typedef unsigned long size_t;
```

However, Polyspace ignores this `typedef` statement, because the **Set size_t to unsigned long** option allows only `unsigned long`.

```
typedef unsigned int size_t;
```

Settings

Default: Off

□ Off

Use the default underlying type of `size_t`, unsigned int

☑ On

Set the type of `size_t` to unsigned long

Command-Line Information

Parameter: `-size-t-is-unsigned-long`
Default: off
Example: `polyspace-code-prover-nodesktop -lang cpp -size-t-is-unsigned-long`

Ignore link errors (C++)

Ignore linkage errors. This option is available on the **Environment Settings** node in the **Configuration** pane.

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

Default: Off

□ Off

Stop analysis for linkage errors.

☑ On

Ignore the linkage errors if possible.

Command-Line Information

Parameter: `-no-extern-C`
Default: off
Example: `polyspace-code-prover-nodesktop -lang cpp -no-extern-C`

Check MISRA C++ rules

Specify whether to check for violation of MISRA C++ rules. Each value of the option corresponds to a subset of rules to check. This option is available on the **Coding Rules** node in the **Configuration** pane.

After analysis, the **Results Summary** pane lists the coding rule 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++)”

`custom`

Specify coding rules to check. Click Edit to create a coding rules file.

After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click Edit. Click 📁 to load the file.

Format of the custom file:

```
<rule number> off|on
```

Use # to enter comments in the file. For example:

```
9-5-1 off # rule 9-5-1: classes
15-0-2 on # rule 15-0-2: exception handling
```

Command-Line Information

Parameter: `-misra-cpp`
Value: `required-rules` | `all-rules` | `SQO-subset1` | `SQO-subset2` | *`file`*
Default: `required-rules`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-misra-cpp all-rules`

Related Examples

- “Specify Analysis Options”
- “Set Up Coding Rules Checking”

More About

- “Polyspace MISRA C++ Checker”
- “Software Quality Objective Subsets (C++)”
- “MISRA C++ Coding Rules”

Check JSF C++ rules

Specify whether to check for violation of JSF C++ rules (JSF++:2005). Each value of the option corresponds to a subset of rules to check. This option is available on the **Coding Rules** node in the **Configuration** pane.

After analysis, the **Results Summary** pane lists the coding rule 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.

`custom`

Specify coding rules to check. Click Edit to create a coding rules file.

After creating and saving the file, to reuse it for another project, do one of the following:

- Enter full path to the file in the space provided.
- Click Edit. Click 📁 to load the file.

Format of the custom file:

```
<rule number> off|on
```

Use # to enter comments in the file. For example:

```
67 off # rule 67: classes
```

```
    202 on # rule 202: expressions
```

Tips

- If your project uses a dialect other than ISO, some rules might not be completely checked. For example, AV Rule 8: “All code shall conform to ISO/IEC 14882:2002(E) standard C++.”

Command-Line Information

Parameter: `-jsf-coding-rules`
Value: `shall-rules` | `shall-will-rules` | `all-rules` | *`file`*
Default: `shall-rules`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-jsf-coding-rules all-rules`

Related Examples

- “Specify Analysis Options”
- “Set Up Coding Rules Checking”

More About

- “Polyspace JSF C++ Checker”
- “JSF C++ Coding Rules”

Files and folders to ignore (C++)

Specify files and folders to ignore during coding rules checking. This option is available on the **Inputs & Stubbing** node in the **Configuration** pane.

The files and folders are **not** ignored during Code Prover verification.

Settings

Default: `all-headers`

`all-headers`

Ignores `.h` or `.hpp` files

`all`

Ignores all files in include folders

`custom`

Ignore include files and folders that you specify in the **File/Folder** view. To add files to the custom **File/Folder** list, select 📁 to choose the files and folders to exclude. To remove a file or folder from the list of excluded files and folders, select the row. Then click ✖.

Dependencies

This option is enabled only if you select one of the options **Check MISRA C++ rules**, **Check JSF C++ rules** or **Check custom rules**.

Command-Line Information

Parameter: `-includes-to-ignore`
Value: `all-headers | all |` *`file1`*`[,`*`file2`*`[,...]] |` *`folder1`*`[,`*`folder2`*`[,...]]`
Default: `all-headers`
Example: `polyspace-code-prover-nodesktop -lang cpp -sources` *`file_name`* `-jsf-coding-rules required-rules -includes-to-ignore "C:\usr \include"`

See Also

“Check MISRA C++ rules” | “Check JSF C++ rules” | “Check custom rules (C/C++)”

Related Examples

- “Specify Analysis Options”
- “Set Up Coding Rules Checking”

Main entry point (C++)

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`. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

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 when you select:

- `Visual` for **Target & Compiler** > **Target operating system**
- **Code Prover Verification** > **Verify whole application**

Command-Line Information

Parameter: `-main`
Value: `_tmain | wmain | _tWinMain | wWinMain | WinMain | DllMain`

Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-OS-target visual -main _tmain`

See Also

"Verify module (C++)"

Related Examples

- "Specify Analysis Options"

Verify module (C++)

Specify that Polyspace must generate a `main` function during verification if it does not find one in the source files. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: On

● On

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**.
2. Calls functions specified by **Initialization functions** ahead of other functions.
3. Calls functions specified by **Functions to call** in arbitrary order.
4. Calls class methods specified by **Class** and **Functions to call within the specified classes**.

If you do not specify the above options explicitly, the generated `main`:

- Initializes all global variables except those declared with keywords `const` and `static`.
- Calls in arbitrary order all functions and class methods that are not called anywhere in the source files. Polyspace considers that global variables can be written between two consecutive function or methods calls. Therefore, in each called function or method, global variables initially have the full range of values allowed by their type.

○ Off

Polyspace stops verification if it does not find a `main` function in the source files.

Command-Line Information

Parameter: `-main-generator`
Default: Off

Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator ...`

See Also

"Variables to initialize (C++)" | "Functions to call (C++)" | "Initialization functions (C++)" on page 2-49 | "Class (C++)" | "Functions to call within the specified classes (C++)"

Related Examples

- "Specify Analysis Options"
- "Verify C++ Classes"
- "Configure Polyspace Analysis Options and Properties"

More About

- "Generate main Function"
- "Main Generation for Model Verification"

Class (C++)

Specify classes that Polyspace uses to generate a `main`. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

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

This option is enabled only if you select **Code Prover Verification** > **Verify module**.

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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2`

See Also

"Verify module (C++)" | "Functions to call within the specified classes (C++)" | "Analyze class contents only (C++)" | "Skip member initialization check (C++)"

Related Examples

- "Specify Analysis Options"
- "Verify C++ Classes"

Functions to call within the specified classes (C++)

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. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

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.

Dependencies

This option is enabled only if you select **Code Prover Verification** > **Verify module**.

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: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -class-analyzer custom=myClass1,myClass2 -class-analyzer-calls unused-public`

See Also

"Verify module (C++)" | "Class (C++)" | "Analyze class contents only (C++)" | "Skip member initialization check (C++)"

Related Examples

- "Specify Analysis Options"
- "Verify C++ Classes"

Analyze class contents only (C++)

Specify that Polyspace must verify only methods of classes that you specify using the **Class** option. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: Off

☑ On

Polyspace verifies the class methods only. It stubs functions out of class scope even if the functions are defined in your code.

□ Off

Polyspace verifies functions out of class scope in addition to class methods.

Dependencies

This option is enabled only if you select **Code Prover Verification** > **Verify module**. If you select this option, you must specify the classes using the **Class** 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

See Also

“Verify module (C++)” | “Class (C++)” | “Functions to call within the specified classes (C ++)” | “Skip member initialization check (C++)”

Related Examples

- “Specify Analysis Options”
- “Verify C++ Classes”

Skip member initialization check (C++)

Specify that Polyspace must not check whether each class constructor initializes all class members. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: Off

☑ On

Polyspace does not check whether each class constructor initializes all class members.

□ Off

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

This option is enabled only if you select **Code Prover Verification** > **Verify module**. If you select this option, you must specify the classes using the **Class** option.

Command-Line Information

Parameter: `-no-constructors-init-check`
Default: Off

See Also

“Verify module (C++)” | “Class (C++)”

Related Examples

- “Specify Analysis Options”

Functions to call (C++)

Specify functions that you want the generated `main` to call. You can use this option only to specify functions that are not members of a class. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: `unused`

`none`

The generated `main` does not call any function.

`unused`

The generated `main` calls only those functions that are not being 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. Click ✚ to enter function name.

Dependencies

This option is enabled only if you select **Verify module**.

Tips

- Select `unused` when you use the option **Run unit by unit verification**.
- If you want the generated `main` to call an inlined function, select `custom` and specify the function name.
- Select `none`:
  - If you do not want to verify uncalled functions. For applications that are not multitasking, Polyspace cannot verify a function unless it can be reached from `main`.
  - To verify a multitasking application without a main.

Command-Line Information

Parameter: `-main-generator-calls`
Value: `none | unused | all | custom=`*`function1`*`[,`*`function2`*`[,...]]`
Default: `unused`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -main-generator-calls unused`

Related Examples

- "Specify Analysis Options"

Variables to initialize (C++)

Specify global variables that you want the generated `main` to initialize. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

If you use the generated `main` to initialize a global variable, inside a function, before the first write operation on the variable, Polyspace considers it to have any value allowed by its type.

Settings

Default: `uninit`

`uninit`

> The generated `main` only initializes global variables that you have not initialized during declaration.

`none`

> The generated `main` does not initialize global variables.

`public`

> The generated `main` initializes all global variables except those declared with the keywords `static` and `const`.

`all`

> The generated `main` initializes all global variables except those declared with the keyword `const`.

`custom`

> The generated `main` only initializes global variables that you specify. Click ➕ to enter variable name.

Dependencies

This option is enabled only if you select **Verify module**.

Command-Line Information

Parameter: `-main-generator-writes-variables`
Value: `none | public | all | custom=`*`variable1`*`[,`*`variable2`*`[,...]]`

Default: `uninit`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -main-generator-writes-variables all`

Related Examples

- "Specify Analysis Options"

Initialization functions (C++)

Specify functions that you want the generated `main` to call ahead of other functions. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

No Default

Click ➕ to add a field.

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);
}
```

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.

Command-Line Information

Parameter: `-functions-called-before-main`
Value: *`function1`*`[,`*`function2`*`[,...]]`
No Default
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -functions-called-before-main myClass::init`

Dependencies

This option is enabled only if you select **Verify module**.

Related Examples

- "Specify Analysis Options"

Parameters (C++)

This option is available only for model-generated code.

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. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

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 class members, use the syntax `className::variableName`.

Command-Line Information

Parameter: `-variables-written-before-loop`
Value: `none | all | custom=`*`variable1`*`[,`*`variable2`*`[,...]]`
Default: `none`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -variables-written-before-loop all`

See Also

"Inputs (C++)" on page 2-47 | "Initialization functions (C++)" on page 2-49 | "Step functions (C++)" on page 2-50 | "Termination functions (C++)" on page 2-52

Related Examples

- "Specify Analysis Options"

- “Configure Polyspace Analysis Options and Properties”

More About

- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Inputs (C++)

This option is available only for model-generated code.

Specify variables that the generated `main` must write to, 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. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

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 class members, use the syntax `className::variableName`.

Command-Line Information

Parameter: `-variables-written-in-loop`
Value: `none | all | custom=`*`variable1`*`[,`*`variable2`*`[,...]]`
Default: `public`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -variables-written-in-loop all`

See Also

“Parameters (C++)” on page 2-45 | “Initialization functions (C++)” on page 2-49 | “Step functions (C++)” on page 2-50 | “Termination functions (C++)” on page 2-52

Related Examples

- “Specify Analysis Options”

- “Configure Polyspace Analysis Options and Properties”

More About

- “Recommended Polyspace options for Verifying Generated Code”
- “Main Generation for Model Verification”

Initialization functions (C++)

This option is available only for model-generated code.

Specify functions that the generated `main` must call before the cyclic code begins. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

No Default

Click ➕ to add a field. Enter function name. For class methods, use the syntax `className::functionName`.

Command-Line Information

Parameter: `-functions-called-before-loop`

No Default

Value: *`function1`*`[,`*`function2`*`[,...]]`

Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -functions-called-before-loop myfunc`

See Also

"Parameters (C++)" on page 2-45 | "Inputs (C++)" on page 2-47 | "Step functions (C++)" on page 2-50 | "Termination functions (C++)" on page 2-52

Related Examples

- "Specify Analysis Options"
- "Configure Polyspace Analysis Options and Properties"

More About

- "Recommended Polyspace options for Verifying Generated Code"
- "Main Generation for Model Verification"

Step functions (C++)

This option is available only for model-generated code.

Specify functions that the generated `main` must call in each cycle of the cyclic code. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

Default: `none`

`none`

The generated `main` does not call functions in the cyclic code.

`all`

The generated `main` calls all functions except inlined ones.

`custom`

The generated `main` calls functions that you specify. Click ✚ to add a field. Enter function name. For class methods, use the syntax `className::functionName`.

Tips

- If you specify a function for the option **Initialization functions** or **Termination functions**, you cannot specify it for **Step functions**.

Command-Line Information

Parameter: `-functions-called-in-loop`
Value: `none | all | custom=`*`function1`*`[,`*`function2`*`[,...]]`
Default: `none`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-main-generator -functions-called-in-loop all`

See Also

"Parameters (C++)" on page 2-45 | "Inputs (C++)" on page 2-47 | "Initialization functions (C++)" on page 2-49 | "Termination functions (C++)" on page 2-52

Related Examples

- "Specify Analysis Options"
- "Configure Polyspace Analysis Options and Properties"

More About

- "Recommended Polyspace options for Verifying Generated Code"
- "Main Generation for Model Verification"

Termination functions (C++)

This option is available only for model-generated code.

Specify functions that the generated `main` must call after the cyclic code loop. This option is available on the **Code Prover Verification** node in the **Configuration** pane.

Settings

No Default

Click ✚ to add a field. Enter function name. For class methods, use the syntax `className::functionName`.

Tips

- If you specify a function for the option **Initialization functions**, you cannot specify it for **Termination functions**.

Command-Line Information

Parameter: `-functions-called-after-loop`
No Default
Value: `*function1*[,*function2*[,...]]`
Example: `polyspace-code-prover-nodesktop -sources *file_name* -main-generator -functions-called-after-loop myfunc`

See Also

“Parameters (C++)” on page 2-45 | “Inputs (C++)” on page 2-47 | “Initialization functions (C++)” on page 2-49 | “Step functions (C++)” on page 2-50

Related Examples

- “Specify Analysis Options”
- “Configure Polyspace Analysis Options and Properties”

More About

- “Recommended Polyspace options for Verifying Generated Code”

- “Main Generation for Model Verification”

No STL stubs (C++)

Specify that the verification must not use Polyspace implementations of the Standard Template Library. This option is available on the **Inputs & Stubbing** node in the **Configuration** pane.

Settings

Default: Off

☑ On

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

□ Off

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

Tips

Use this option when Polyspace implementation of the Standard Template Library causes linking errors.

Command-Line Information

Parameter: `-no-stl-stubs`
Default: Off

Related Examples

- "Specify Analysis Options"

Functions to stub (C++)

Specify functions to stub during verification. This option is available on the **Inputs & Stubbing** node in the **Configuration** pane.

For these 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.

Settings

No Default

Click ➕ to enter function name.

When entering function names, use one of the following syntaxes:

- Basic syntax, with extensions for classes and templates:

Function Type	Syntax
Simple function	`test`
Class method	`A::test`
Template method	`A<T>::test`

- Syntax with function arguments, to differentiate overloaded functions. Function arguments are separated with semicolons:

Function Type	Syntax
Simple function	`test()`
Class method	`A::test(int;int)`
Template method	`A<T>::test<T>::test(T;T)`

Tips

If you do not want to review checks in a certain function, you can stub the function. However, Polyspace makes certain assumptions about the arguments and return values of stubbed functions. The assumptions can affect the number of checks in the rest of the code. For example, the software considers that the return values assume the full range allowed by the return type. For more information, see "Assumptions About Stubbed Functions".

You can specify external constraints on the arguments and return values of stubbed functions. See "Constrain Stubbed Functions".

Command-Line Information

Parameter: `-functions-to-stub`
No Default
Value: `*function1*[,*function2*[,...]]`
Example: `polyspace-code-prover-nodesktop -sources *file_name* -functions-to-stub function_1,function_2`

See Also

"No automatic stubbing (C/C++)" | "Variable/function range setup (C/C++)" | "Functions to stub (C)"

Related Examples

- "Specify Analysis Options"
- "Specify Functions to Stub Automatically"
- "Constrain Data with Stubbing"

More About

- "Stubbing Overview"
- "When to Provide Function Stubs"
- "Stubbing Examples"

Tuning Precision and Scaling Parameters

Precision versus Time of Verification

There is a compromise to be made to balance the time required to obtain results, and the precision of those results. Consequently, launching Polyspace verification with the following options will allow the time taken for verification to be reduced but will compromise the precision of the results. It is suggested that the parameters should be used in the sequence shown - that is, if the first suggestion does not increase the speed of verification sufficiently then introduce the second, and so on.

- switch from -O2 to a lower precision;
- set the “Respect types in global variables (C/C++)” and “Respect types in fields (C/C++)” options;
- set the option “Depth of verification inside structures (C/C++)” to 2, then 1, or 0;
- stub manually missing functions which write into their arguments.

Precision versus Code Size

Polyspace verification can make approximations when computing the possible values of the variables, at any point in the program. Such an approximation will use a superset of the actual possible values.

For instance, in a relatively small application, Polyspace verification might retain very detailed information about the data at a particular point in the code, so that for example the variable VAR can take the values { -2; 1; 2; 10; 15; 16; 17; 25 }. If VAR is used to divide, the division is green (because 0 is not a possible value). If the program being analyzed is large, Polyspace verification would simplify the internal data representation by using a less precise approximation, such as [-2; 2] U {10} U [15 ; 17] U {25} . Here, the same division appears as an orange check.

If the complexity of the internal data becomes even greater later in the verification, Polyspace verification might further simplify the VAR range to (say) [-2; 20].

This phenomenon leads to the increase or the number of orange warnings when the size of the program becomes large.

Note: The amount of simplification applied to the data representations also depends on the required precision level (O0, O2), Polyspace verification will adjust the level of simplification:

- `-O0`: shorter computation time. You only need to focus on red and gray checks.
- -`O2`: less orange warnings.
- `-O3`: less orange warnings and bigger computation time.

Verification level (C++)

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. This option is available on the **Precision** node in the **Configuration** pane.

Settings

Default: `Software Safety Analysis level 2`

`C++ source compliance checking`

Polyspace completes coding rules checking at the end of the compilation phase.

`Software Safety Analysis level 0`

The verification process runs once on your source code.

`Software Safety Analysis level 1`

The verification process runs twice on your source code.

`Software Safety Analysis level 2`

The verification process runs thrice on your source code. Use this option for most accurate results in reasonable time.

`Software Safety Analysis level 3`

The verification process runs four times on your source code.

`Software Safety Analysis level 4`

The verification process runs five times on your source code.

`other`

If you use this option, Polyspace verification will make 20 passes unless you stop it manually.

Tips

- 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.

Dependency

You cannot use the `C++ Source Compliance Checking` setting for remote verification.

Command-Line Information

Parameter: `-to`
Value: `cpp-compliance | pass0 | pass1 | pass2 | pass3 | pass4 | other`
Default: `pass2`
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-to pass2`

Related Examples

- "Specify Analysis Options"
- "Improve Verification Precision"

Inline (C++)

Specify functions that the verification must clone for every function call. For instance, if you specify the function `func` for inlining and `func` is called twice, the software creates two copies of `func` during verification. The copies are named using the convention `func`*`ver`*`_pst_cloned_`*`tot`* where *`ver`* is the version number and *`tot`* is the total number of copies. This option is available on the **Scaling** node in the **Configuration** pane.

Settings

No Default

Click ➕ to enter function name.

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`.
- This option applies to all overloaded methods of a class.

Command-Line Information

Parameter: `-inline`
Value: *`function1`*`[,`*`function2`*`[,...]]`
No Default
Example: `polyspace-code-prover-nodesktop -sources` *`file_name`* `-inline my_func`

Related Examples

- "Specify Analysis Options"
- "Reduce Procedure Complexity"

Other (C++)

In this section...
"-extra-flags" on page 2-62
"-cpp-extra-flags" on page 2-62
"-il-extra-flags" on page 2-62

Specify special options for C++ verification, which are provided by MathWorks if required. This option is available on the **Advanced Settings** node in the **Configuration** pane.

-extra-flags

No Default

Example Shell Script Entry:

```
polyspace-code-prover-nodesktop -extra-flags -param1 -extra-flags -
param2
```

-cpp-extra-flags

No Default

Example Shell Script Entry:

```
polyspace-code-prover-nodesktop -cpp-extra-flags -stubbed-new-may-
return-null
```

-il-extra-flags

No Default

Example Shell Script Entry:

```
polyspace-code-prover-nodesktop -il-extra-flags flag
```

3

Polyspace 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.

Mark the offending code block by two `#pragma` directives, one at the beginning of the asm 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`.

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:

```
polyspace-code-prover-nodesktop -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.

See Also

polyspaceCodeProver

-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`.

Note: In the Polyspace environment, 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.

```
polyspace-code-prover-nodesktop -author "John Smith"
```

See Also

`-date` | `-prog` | `polyspaceCodeProver`

-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.

```
polyspace-code-prover-nodesktop -date "15/03/2012"
```

See Also

`-author` | `-prog`

-detect-pointer-escape

Detect stack pointer dereference outside scope

Syntax

```
-detect-pointer-escape
```

Description

`-detect-pointer-escape` detects stack pointer dereferences outside scope. Such dereference can happen, for example, when a pointer to a variable that is local to a function is returned from the function. Because the scope of the variable is limited to the function, dereferencing the pointer outside the function can cause undefined behavior.

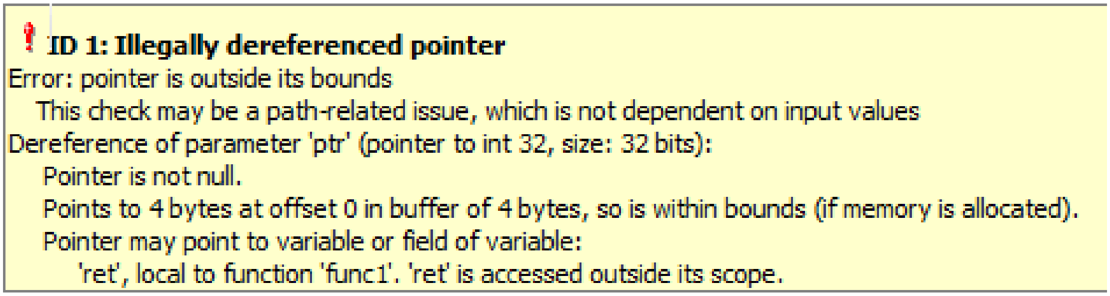
Examples

In the following code, if you use 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 **Check 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.

See Also

`polyspaceCodeProver`

Related Examples

- "Run Local Verification at Command Line"

Introduced in R2015a

-h[elp]

Display list of possible options

Syntax

```
-h
-help
```

Description

`-h` and `-help` display the list of possible options in the shell window and the argument syntax.

Examples

Display the command-line help.

```
polyspace-code-prover-nodesktop -h
polyspace-code-prover-nodesktop -help
```

See Also

`polyspaceCodeProver`

-I

Specify include folder for compilation

Syntax

```
-I folder
```

Description

`-I` *`folder`* specifies the name of a folder that you must include when compiling C sources. You can specify only one folder for each instance of `-I`. However, you can specify this option multiple times.

Polyspace software automatically includes the `./sources` folder (if it exists) after the include folders that you specify.

Examples

Include two folders with the analysis.

```
polyspace-code-prover-nodesktop -I /com1/inc -I /com1/sys/inc
```

Because `./sources` is included automatically, this Polyspace command is equivalent to:

```
polyspace-code-prover-nodesktop -I /com1/inc -I /com1/sys/inc
                                                -I ./sources
```

See Also

`polyspaceCodeProver`

-import-comments

Import comments and justifications from previous analysis

Syntax

`-import-comments` *`resultsFolder`*

Description

`-import-comments` *`resultsFolder`* imports the comments and justifications from a previous analysis, as specified by the results folder.

Examples

Increment your project's version number (`-version`) and import comments from the previous results.

```
polyspace-code-prover-nodesktop -version 1.3
        -import-comments C:\Results\myProj\1.2
```

See Also

`-version` | `polyspaceCodeProver`

-lang

Specify code language for the project

Syntax

`-lang *[c|cpp]*`

Description

`-lang` *`[c|cpp]`* specifies the code language for the project, either `c` for C code or `cpp` for C++ code.

If you do not specify a language, Polyspace tries to detect the language from the source files.

Note: In the Polyspace user interface, specify the project language when you create a new project. For more information, see “Create Project”.

Examples

Define the language of your Polyspace Project as C++.

```
polyspace-code-prover-nodesktop -lang cpp -sources...
```

See Also

`polyspaceCodeProver`

-max-processes

Specify the maximum number of processes that can run simultaneously on a multicore system.

Syntax

```
-max-processes num
```

Description

`-max-processes` *`num`* specifies the maximum number of processes that can run simultaneously on a multicore system. The valid range of *`num`* is 1 to 128. The default is 4.

Examples

Disable parallel processing during the analysis.

```
polyspace-code-prover-nodesktop -max-processes 1
```

See Also

`polyspaceCodeProver`

-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:

```
#These are the options for MyCodeProverProject
-lang c
-prog MyCodeProverProject
-author jsmith
-sources "mymain.c,funAlgebra.c,funGeometry.c"
-OS-target no-predefined-OS
-target x86_64
-dialect none
-dos
-misra2 required-rules
-includes-to-ignore all-headers
-main-generator
-results-dir C:\Polyspace\MyCodeProverProject
```

2 Run Polyspace using options in the file `listofoptions.txt`.

```
polyspace-code-prover-nodesktop -options-file listofoptions.txt
```

See Also

`polyspaceCodeProver` | `polyspaceConfigure`

-prog

Specify name of project

Syntax

```
-prog projectName
```

Description

`-prog` *`projectName`* specifies the name of your Polyspace project. This name must use only letters, numbers, underscores (_), dashes (-), or periods (.).

Examples

Assign a session name to your Polyspace Project.

```
polyspace-code-prover-nodesktop -prog MyApp
```

See Also

`-author` | `-date` | `polyspaceCodeProver`

-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`.

Examples

Specify the name of the analysis report.

```
polyspace-code-prover-nodesktop -report-template Developer
      -report-output-name Airbag_v3.rtf
```

See Also

“Output format (C/C++)” on page 1-127 | “Report template (C/C++)” | `polyspaceCodeProver`

-results-dir

Specify the results folder

Syntax

```
-results-dir
```

Description

`-results-dir` specifies where to save the analysis results. The default location at the command line is the current folder. In the user interface, the default location is `C:Polyspace_Results`.

Examples

Specify to store your results in the `RESULTS` folder.

```
polyspace-code-prover-nodesktop -results-dir RESULTS ...
   export RESULTS=results_'date + %d%B_%HH%M_%A'
polyspace-code-prover-nodesktop -results-dir 'pwd'/$RESULTS
```

See Also

`polyspaceCodeProver`

-scheduler

Specify cluster or job scheduler

Syntax

```
-scheduler schedulingOption
```

Description

`-scheduler` *`schedulingOption`* specifies the head node of the MDCS cluster or MATLAB job scheduler on the node host. Use this command to manage the cluster, or to specify where to run batch analyses.

Examples

Run a batch analysis on a remote server.

```
polyspace-code-prover-nodesktop -batch -scheduler NodeHost
polyspace-code-prover-nodesktop -batch -scheduler 192.168.1.124:12400
polyspace-code-prover-nodesktop -batch -scheduler MJSName@NodeHost
```

```
polyspace-job-manager listjobs -scheduler NodeHost
```

See Also

`polyspaceCodeProver` | `polyspaceJobsManager` | `polyspaceJobsManager`

-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. The list must be in quotations and separated by commas. 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`.

```
polyspace-code-prover-nodesktop -sources mymain.c
     -sources funAlgebra.c -sources funGeometry.c
```

See Also

`polyspaceCodeProver`

-sources-list-file

Specify file containing list of sources

Syntax

```
-sources-list-file "filename"
```

Description

`-sources-list-file "filename"` specifies 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 absolute path to a source file. For example:

```
C:\Sources\myfile.c
C:\Sources2\myfile2.c
```

This option is available only in batch analysis mode.

Examples

Run analysis on files listed in `files.txt`.

```
polyspace-code-prover-nodesktop -batch -scheduler NODEHOST
  -sources-list-file "C:\Analysis\files.txt
polyspace-code-prover-nodesktop -batch -scheduler NODEHOST
  -sources-list-file "/home/polyspace/files.txt"
```

See Also

`polyspaceCodeProver`

-tmp-dir-in-results-dir

Keep temporary files in results folder

Syntax

```
-tmp-dir-in-results-dir
```

Description

`-tmp-dir-in-results-dir` keeps temporary files in the results folder. By default, temporary files are stored in the standard `/temp` or `C:\Temp` folder. This option stores the temporary files in a subfolder of the results folder. Use this option only when the temporary folder partition does not have enough disk space. If the results folder is mounted on a network drive, this option can slow down your processor.

Examples

Store temporary files in the results folder.

```
polyspace-code-prover-nodesktop -tmp-dir-in-results-dir
```

See Also

`polyspaceCodeProver`

-v[ersion]

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.

```
polyspace-code-prover-nodesktop -v
```

See Also

`polyspaceCodeProver`

-verif-version

Assign a version identifier

Syntax

```
-verif-version id
```

Description

`-verif-version` *`id`* assigns a verification identifier, *`id`*, to identify the verification. You can use this identifier to refer to different verifications at the command line. For example, you can import comments from a previous verification using the verification identifier.

Examples

Assign a verification identifier.

```
polyspace-code-prover-nodesktop -verif-version 1.3
```

See Also

`polyspaceCodeProver`

Check Reference

Absolute address

Absolute address is assigned to pointer

Description

This check appears when an absolute address is assigned to a pointer.

This check is always orange because the software does not have information about the absolute address and cannot verify, for example, the validity of the address and the availability of memory.

To disable this check, specify the appropriate option. See "Green absolute address checks (C/C++)".

Diagnosing This Check

"Review and Fix Absolute Address Checks"

Examples

Absolute address assigned to pointer

```
void main() {
    int *p = (int *)0x32;
    int x = *p;
    p++;
    x = *p;
}
```

In this example, 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** check even though p still points to an absolute address.

Correction — Use Polyspace analysis option

You can use absolute addresses in your code and not produce an orange **Absolute address** error. To allow absolute addresses, on the **Configuration** pane, under **Verification Assumptions**, select **Green absolute address checks**.

```
void main() {
    int *p = (int *)0x32;
    int x = *p;
    p++;
    x = *p;
}
```

Check Information

Category: Static memory
Language: C | C++
Acronym: ABS_ADDR

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.

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() {
    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() {
    float val = 0.0;
    myFuncStruct.func(val);
}
```

In this example, because the global variable myFuncStruct is not initialized, the function pointer myFuncStruct.func contains NULL. Therefore, 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() {
    for (int i=1; i<=MAX_MEMSEG; i++)
        operation(i);
}
```

In this example, the function pointer operation is cast to the contents of a location in memory. Because Polyspace cannot determine whether the location contains a variable or a function code, the **Absolute address** check produces an orange error on the cast. Subsequently, when the pointer operation is dereferenced, the **Correctness condition** check produces a red error.

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 = &func;
    int arg = 0, result = funcPtr(&arg);
}
```

In this example, the function pointer `funcPtr` points to a function with argument type `complex*`. However, it is assigned the function `func` whose argument type is `int*`. Because of this type mismatch, the **Correctness condition** check produces a red error.

Function pointer points to function with wrong number of arguments

```
typedef int (*typeFuncPtr) (int, int);

int func(int);

void main() {
    typeFuncPtr funcPtr = &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 produces a red error.

Function pointer points to function with wrong return type

```
typedef double (*typeFuncPtr) (int);

int func(int);
```

```
void main() {
    typeFuncPtr funcPtr = &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 produces a red error.

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`.

Check Information

Category: Other
Language: C | C++
Acronym: COR

See Also

"Variable/function range setup (C/C++)"

More About

- “Constrain Global Variables”

C++ specific checks

C++ specific invalid operations occur

Description

This check 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.

Examples

Array size is 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 in two categories:

- `n = 0`: When the `new` operator uses this argument, the **C++ specific checks** produce an error.
- `n > 0`: When the `new` operator uses this argument, the **C++ specific checks** is green.

Combining the two categories of arguments, the **C++ specific checks** produce an orange error on the `new` operator.

`typeid` operator dereferences a `NULL` pointer

```
#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) {
        if(radiusVal > 0)
            radius = radiusVal;
        else
            radius = 0;
    }

    double area() {
        return (PI * radius * radius);
    }
};

class Square: public Shape {
    double side;
public:
    Square(double sideVal):Shape() {
        setVal(sideVal);
    }

    void setVal(double sideVal) {
        if(sideVal > 0)
            side = sideVal;
        else
            side = 0;
    }

    double area() {
        return (side * side);
    }
};

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 if(typeid(*shapePtr) == typeid(Square)) {
        std::cout<<"Enter side:";
        std::cin>>val;
        shapePtr -> setVal(val);
        std::cout<<"Area of square = "<<shapePtr -> area();
    }
    else {
        std::cout<<"No valid shape.";
    }

}
```

In this example, the `Shape*` pointer `shapePtr` returned by `getShapePtr()` function can be:

- `NULL`: When `shapePtr` is used with the `typeid` operator, the **C++ specific checks** produce an error.
- Not `NULL`: When `shapePtr` is used with the `typeid` operator, the **C++ specific checks** is green.

Combining these two cases, the **C++ specific checks** produce an orange error on the `typeid` operator in the first `if` statement branch in `main`.

Following this orange error, Polyspace considers that `shapePtr` is not `NULL`. Therefore, the **C++ specific checks** on the `typeid` operator in the second `if` statement branch is green.

Check Information

Category: C++
Language: C++
Acronym: CPP

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.

```
#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

```
#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.

```
#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>
#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;
        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>
#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;
        if(divisor < -0.00001 || divisor > 0.00001)
            printf(" .2f ", 1.0/divisor);
        else
            printf(" Infinite ");
    }
}
```

Check Information

Category: Numerical

Language: C | C++
Acronym: ZDV

Exception handling

Exception handling

Description

This check determines whether:

- A function call throws an exception.
- The exception is caught.

This check appears on both a function call as well as the function body. Use this check to follow the propagation of error from an entry-point function down the branches of the call tree.

Examples

Exception in calls to function

```
#include <vector>

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 appears as a red **Exception handling** error on both the function call and function body. A red **Exception handling** error also appears on `main` because a function call inside `main` throws an exception.

Exception handled through `try`/`catch` construct

```
class error {
    error() {  }
    error(const error&) { }
} ;


void funcNegative() {
    try {
        throw error() ;
    }
    catch (error NegativeError) {
    }
}

void funcPositive() {
    try {
    }
    catch (error PositiveError) {
    }
}
```

```
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. Therefore, the exception propagates to the corresponding `catch` block and does not continue further. The **Exception handling** check on the function body, function call, and the `main` function appears green.
- 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 calls to constructor

```
class error {
};

class X
{
public:
    X() {
        throw error();
    }
    ~X() {
        ;
    }
};

int main() {
    try {
        X * px = new X ;
        delete X;
    } catch (error) {
        assert(1) ;
    }
}
```

In this example, the new operator calls the constructor X::X(). The constructor throws an exception. The exception appears as a red **Exception handling** error on the constructor body and the new operator. The exception then propagates to the catch block and does not continue farther. Therefore the **Exception handling** check on the main function appears green.

The green assert statement shows that the exception has propagated to the catch block.

Exception in calls 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 **Exception handling** error is red.

Type mismatch between `throw` declaration and usage

```
#include <string>

class negativeBalance {
public:
    negativeBalance(const string & s): errorMessage( s) {}
    ~negativeBalance() {}
private:
    string errorMessage;
};

class Account {
```

```
public:
    Account(long initVal):balance(initVal) {}
    ~Account() {}
    void debitAccount(long debitAmount) throw (int, char);
private:
    long balance;
};

void Account::debitAccount(long debitAmount) throw (int, char) {
    if((balance - debitAmount) < 0 )
        throw negativeBalance("Negative balance");
    else
        balance -= debitAmount;
}

void main() {
    Account *myAccount = new Account(1000);
    try {
        myAccount -> debitAccount(2000);
    }
    catch(negativeBalance &) {
    }
    delete myAccount;
}
```

In this example, the arguments to `throw` in the `Account::debitAccount` method are declared to be either `int` or `char`. However, the method throws an exception with type `negativeBalance`. Therefore, the **Exception handling** check produces a red error on `throw`.

Check Information

Category: C++
Language: C++
Acronym: EXC

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 (C/C++)”.

Diagnosing This Check

“Review and Fix Function Not Reachable Checks”

Examples

Function Call from Unreachable Branch of Condition

```
#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 checkRatio(double ratio) {
    checkUpperBound(ratio);
    checkLowerBound(ratio);
}

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 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

Category: Data flow
Language: C | C++
Acronym: FNR

See Also

Polyspace Analysis Options

"Detect uncalled functions (C/C++)"

Polyspace Results

Function not called | Unreachable code

Function returns a 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.

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 returns a 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 returns a 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

Category: C++
Language: C++
Acronym: FRV

See Also

Polyspace Results

Initialized return value

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 (C/C++)".

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 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;
    }
}

void reset() {
    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 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();
    }
}

void reset() {
    var=0;
}
```

Function Called from Another Uncalled Function

```
#define max 100
int var;
int numberOfResets;
int getValue();
int getSaturation();
```

```
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;
        }
    }
}

void reset() {
    updateCounter();
    var=0;
}

void updateCounter() {
    numberOfResets++;
}
```

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 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();
    }
}

void reset() {
    updateCounter();
    var=0;
}

void updateCounter() {
    numberOfResets++;
}
```

Check Information

Category: Data flow
Language: C | C++
Acronym: FNC

See Also

Polyspace Analysis Options

"Detect uncalled functions (C/C++)"

Polyspace Results

Function not reachable

Illegally dereferenced pointer

Pointer is dereferenced outside bounds

Description

This check on a pointer dereference determines whether the pointer points outside its bounds.

When you assign an address to a pointer, a block of memory is associated with the pointer. You cannot access memory beyond that block using 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`

```
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 derferencing `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 = 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 dereference of `cuboidPtr` produce 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 = 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 = 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 = &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 = &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

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**.

```
#include<stdlib.h>
typedef struct {
    int length;
    int breadth;
} square;
```

```
void main() {
    square mySquare;
    char* squarePtr = &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. Polyspace Code Prover produces a red **Illegally dereferenced pointer** check on `*ptr`.

Check Information

Category: Static memory
Language: C | C++
Acronym: IDP

See Also

Polyspace Analysis Options

"Allow incomplete or partial allocation of structures (C)" | "Enable pointer arithmetic across fields (C)" | -detect-pointer-escape

Polyspace Results

Non-initialized pointer

Initialized return 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 every function call.

Diagnosing This Check

"Review and Fix Initialized Return Value 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 **Initialized return value** 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 **Initialized return value** 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

Category: Data flow
Language: C
Acronym: `IRV`

See Also

Polyspace Results

Function returns a value

Inspection points

Variable range information appears

Description

This user-specified check provides range information on specified variables. If you want to know the range of the variables var1, var2, ... at a certain point in the code, place the line #pragma var1 var2 ... at that point. After verification, to see the variable range, place your cursor on the variable name.

Note: The tooltip indicates the range that Polyspace considers, not the actual variable range. Because of approximations, the variable range that Polyspace considers can sometimes be a superset of the actual variable range. Use this check to help understand the cause of other Polyspace checks.

Examples

View range of variable

```
int input();

void main() {
    int num=input();
    int i;
    if(num>0 && num<10) {
        for(i=0; i<20; i++)
            num+=i;
#pragma Inspection_Point num
    }
#pragma Inspection_Point num
}
```

In this example, if you place your cursor on the variable num in the #pragma statements, you can view its range. In the first case, the tooltip shows the range [191 .. 199]. In the second case, the tooltip shows the range [-2^{31} .. 0] or [10 .. 2^{31} - 1]. The second range shows that Polyspace considers the return value of `input()` to be in the full range of type `int`.

Check Information

Category: Other
Language: C
Acronym: IPT

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.

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 HALF_PI 1.5707963267948966
#define LARGE_EXP 710

enum operation {
    ASIN,
    ACOS,
    TAN,
    SQRT,
    LOG,
    EXP,
    ACOSH,
    ATANH
};

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 TAN:
        assert( myVal == HALF_PI);
        res = tan(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;
    case ACOSH:
        assert(myVal < 1.0);
        res = acosh(myVal);
        break;
    case ATANH:
        assert(myVal <= -1.0 || myVal >= 1.0);
        res = atanh(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.

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;
}
```

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.

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;
}
```

Invalid use of standard library string routine

```
#include <stdio.h>

char* Copy_String(void)
{
    char *res;
    char gbuffer[5],text[20]="ABCDEFGHIJKL";
    res=strcpy(gbuffer,text);
    return(res);
}
```

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>

char* Copy_String(void)
{
    char *res;
    char gbuffer[20],text[20]="ABCDEFGHIJKL";
    res=strcpy(gbuffer,text);
    return(res);
}
```

Check Information

Category: Other
Language: C | C++
Acronym: STD_LIB

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 Summary** pane or **Source** pane.
2. View the message on the **Check Details** pane.

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;

  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 Summary** pane or **Source** pane.
2. View the message on the **Check Details** pane.

Check Information

Category: Data flow
Language: C | C++
Acronym: NIVL

See Also

Polyspace Results

Non-initialized pointer | Non-initialized variable

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 = 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 = 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

Category: Data flow
Language: C | C++
Acronym: NIP

See Also

Polyspace Results

Non-initialized local variable | Non-initialized variable

Non-initialized variable

Variable other than local variable is not initialized before being read

Description

For variables other than local variables, this check occurs on every variable read. It determines whether the variable being read is initialized.

By default, Polyspace considers that global variables are initialized according to ANSI C standards. For instance, the default initial value of an `int` variable is 0.

To prevent this default assumption during analysis, on the **Configuration** pane, select **Inputs & Stubbing**. Select **Ignore default initialization of global variables**. This option is not available for C++ code.

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.

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();
 if(val>=0 && val<= 100)
   globVar += val;
}
```

Check Information

Category: Data flow
Language: C | C++
Acronym: NIV

See Also

Polyspace Analysis Options

"Ignore default initialization of global variables (C)"

Polyspace Results

Non-initialized local variable | Non-initialized pointer

Non-null this-pointer in method

this pointer is null during member function call

Description

This check on a this pointer dereference determines whether the pointer is NULL.

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 **Non-null this-pointer in 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

Category: C++
Language: C++
Acronym: NNT

Non-terminating call

Called function does not return to calling context

Description

This check on a function call determines whether the called function returns to its calling context. A function does not return to its calling context if it contains a run-time error.

Depending on the context, a function call causing a definite error appears in the user interface in two different ways:

- A dashed red underline on the function call. The dashed red underline indicates that you can typically find a red error in the function body.

  - To find the source of error, on the **Source** pane, place your cursor on the function call and view the tooltip.
  - Navigate to the source of error. Right-click the function call and select **Go to Cause** if the option exists.

  The software does not display this kind of non-terminating call on the **Results Summary** pane. However, following the function call, like other red checks, Polyspace does not analyze the remaining code in the same scope as the check.
- A red error on the function call. The red indicates that there is at least one other call to the same function that does not produce a definite error. You find the error, which is orange, inside the function body.

  Even though a definite failure occurs in one function call, because the verification results in a loop body are aggregated over all function calls, the failure shows as an orange check in the function body. To indicate that a definite failure has occurred, a red **Non-terminating call** check is shown on the function call.

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

Category: Control flow
Language: C | C++
Acronym: NTC

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.

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

Category: Control flow
Language: C | C++
Acronym: NTL

Object oriented programming

Dynamic type of `this` pointer is incorrect

Description

This check on dereference of a `this` pointer or pointer to method determines whether the dereference is valid.

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 **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 **Object oriented programming** check produces a red error.

Check Information

Category: C++
Language: C++
Acronym: OOP

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.

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]);
   }
```

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]);
   }
```

Check Information

Category: Static memory
Language: C | C++
Acronym: OBAI

See Also

Polyspace Results

Illegally dereferenced pointer

Overflow

Arithmetic operation causes overflow

Description

This check on an arithmetic operation determines whether the result overflows. An overflow occurs when the value of a variable falls outside the range allowed by its type.

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}. Since the maximum value that the type `int` can hold on a 32–bit target is $2^{31}-1$, the **Overflow** check on the multiplication produces a red error.

Overflow due to left shift on signed integers

```
void main(void)
 {
  unsigned int i;

  i = 1090654225 << 1;
 }
```

In this example, an **Overflow** error occurs because of integer promotion.

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 can be represented by float on a 32-bit target. Therefore, the operation val * 2 results in an **Overflow** error.

Negative overflow

```
#define FLT_MAX 3.40282347e+38F
#define FLT_MIN 1.17549435e-38

int input();

void main() {
int choice=input();

if(choice==0)
  float_negative_overflow();
else
  int_negative_overflow();
}

void float_negative_overflow() {
 float zer_float = FLT_MIN;
 float min_float = -FLT_MAX;

 zer_float = zer_float * zer_float;
 min_float = -min_float * min_float;
}

void int_negative_overflow() {
 int min_int = -2147483648;
}
```

In this example:

- In `float_negative_overflow`, there are two cases of underflow:
  - In the first case, `zer_float` contains the closest possible number to zero that can be represented by the type `float`. Because the operation `zer_float * zer_float` produces a number that is even closer to zero, it cannot be represented by the type `float`. However, the **Overflow** check does not detect this kind of underflow.
  - In the second case, `min_float` contains the most negative number that can be represented by the type `float`. Because the operation `-min_float * min_float` produces a number that is further negative, it cannot be represented by the type `float`. Therefore, the **Overflow** check produces a red error.
- In `int_negative_overflow`, the variable `min_int` is assigned the value -2147483648. This assignment occurs in three steps:

  1. The value 2147483648 is assigned to an unsigned 32–bit integer.
  2. The unsigned integer is cast to a signed integer.
  3. The unary minus is performed on the signed integer.

  Since the maximum value that a signed integer can have is 2147483647, a overflow occurs in the second step. Therefore, even though the minimum value a signed integer can have is -2147483648, a red **Overflow** error appears on the operation `int min_int = -2147483648;` .

Overflows on constants

```
void main() {
 char x = 0xFFFF;
 x=x+1;
}
```

In this example, the constant `0xFFFF` is greater than the maximum value that can be represented by the type `char`. Therefore the **Overflow** check produces a red error.

The following table lists three kinds of constants with the corresponding data types. For each kind, the data type assigned to a constant is the first data type in the corresponding column that can hold the constant.

Decimal	`int`, `long`, `unsigned long`

Hexadecimal	int, unsigned int, long, unsigned long
Float	float, double

For example, (assuming a 16-bit target) the data types for the following values are listed in this table.

5.8	double
6	int
65536	long
0x6	int
0xFFFF	unsigned int
5.8F	float
65536U	unsigned int

To avoid red **Overflow** errors on constants, on the **Configuration** pane, use the analysis option **Check Behavior** > **Ignore overflowing computations on constants**.

Overflows on unsigned bit fields

```
#include <stdio.h>

struct
{
  unsigned int dayOfWeek : 2;
} Week;

void main()
{
  Week.dayOfWeek = 2;
  Week.dayOfWeek = 3;
  Week.dayOfWeek = 4;
}
```

In this example, `dayOfWeek` occupies 2 bits. Because it is an unsigned integer, it can take values in `[0,3]`. When you assign 4 to `dayOfWeek`, the **Overflow** check is red.

To detect overflows on signed and unsigned integers, on the **Configuration** pane, under **Check Behavior**, select `signed-and-unsigned` for **Detect overflows**.

Overflows on signed and `enum` bit fields

```
enum tBit {
  ZERO = 0x00,
  ONE = 0x01 ,
  TWO = 0x02
};

struct twoBit
{
  enum tBit myBit:2;
} myBitField;

void main()
{
  myBitField.myBit = ZERO;
  myBitField.myBit = ONE;
  myBitField.myBit = TWO;
}
```

In this example, because `myBit` is an `enum` variable, it is implemented through a signed integer according to the ANSI C90 standard. `myBit` occupies 2 bits. Because it is a signed integer, it can take values in `[-2,1]`. When you assign 2 to `myBit`, the **Overflow** check is red.

To detect overflows on signed integers alone, on the **Configuration** pane, under **Check Behavior**, select `signed` for **Detect overflows**.

Check Information

Category: Numerical
Language: C | C++
Acronym: OVFL

See Also

Polyspace Analysis Options

"Detect overflows (C/C++)" | "Ignore overflowing computations on constants (C/C++)" | "Overflow computation mode (C/C++)"

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 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 **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 **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 < svar1) {
        return 1;
    } else {
        return 0;
    }

}
```

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 **Shift operations** check, Polyspace assumes that `svar1 >= svar2`. The branch of the statement, `if(svar1 < svar2)`, is unreachable.

Check Information

Category: Numerical
Language: C | C++
Acronym: SHF

See Also

Polyspace Analysis Options

"Allow negative operand for left shifts (C/C++)"

Unreachable code

Code cannot be reached during execution

Description

This check determines whether a section of code can be reached during execution.

Examples of unreachable code include the following:

- 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

Category: Data flow
Language: C | C++
Acronym: UNR

See Also

Polyspace Results

Function not called | Function not reachable

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.

```
#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

```
#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>

double getNumber();
void squareRootOfDifference(double firstNumber, double 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

Category: Other
Language: C | C++
Acronym: ASRT

Approximations Used During Verification

Why Polyspace Verification Uses Approximations

In this section...

What is Static Verification

Polyspace software uses *static verification* to prove the absence of run-time errors. Static verification derives the dynamic properties of a program without actually executing it. This differs significantly from other techniques, such as run-time debugging, in that the verification it provides is not based on a given test case or set of test cases. The dynamic properties obtained through the Polyspace verification are true for executions of the software.

Polyspace verification works by approximating the software under verification, using representative approximations of software operations and data.

For example, 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`, a traditional approach would be to enumerate each possible value of `i`. One thousand checks would be required.

Using the static verification approach, the variable `i` is modelled by its variation domain. For instance the model of `i` is that it belongs to the [0..999] static interval. (Depending on the complexity of the data, convex polyhedrons, integer lattices and more elaborated models are also used for this purpose).

An approximation, by definition, leads to information loss. For instance, the information that `i` is incremented by one every cycle in the loop is lost. However, the important fact is that this information is not required to ensure that range errors will not occur; it is only necessary to prove that the variation domain of `i` is smaller than the range of `tab`. Only one check is required to establish that – and hence the gain in efficiency compared to traditional approaches.

Static code verification does have an exact solution, but that solution is generally not practical, as it would generally require the enumeration of all possible test cases. As a result, approximation is required.

Exhaustiveness

Nothing is lost in terms of exhaustiveness. The reason is that Polyspace verification works by performing upper approximations. In other words, the computed variation domain of a program variable is a superset of its actual variation domain. As a result, Polyspace verifies run-time error items that require checking.

Polyspace Assumptions About Certain Code Constructs

In this section...

Variable Ranges

For variables whose values cannot be determined from the code, Polyspace assumes full-range of values allowed by their type.

For instance, for a variable of integer type, the software uses the following criteria to determine the minimum and maximum value allowed:

- 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 (C)" or "Target processor type (C++)".

- In addition to the C standard, Polyspace assumes that your target uses the two's complement representation for signed integers. Therefore, 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. Therefore, 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, do the following:

1. Run verification on the following 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 and therefore `val` is full-range.

Initialization of Global Variables

Unless you use the option Ignore default initialization of global variables, Polyspace considers global variables to be initialized according to ANSI C standards. The default values are:

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

If you define global variables in your code, then the software uses the dummy function `_init_globals()` to initialize the global variables. The `_init_globals()` function is the first function called in the `main` function.

Consider the following code in the application `gv_example.c`.

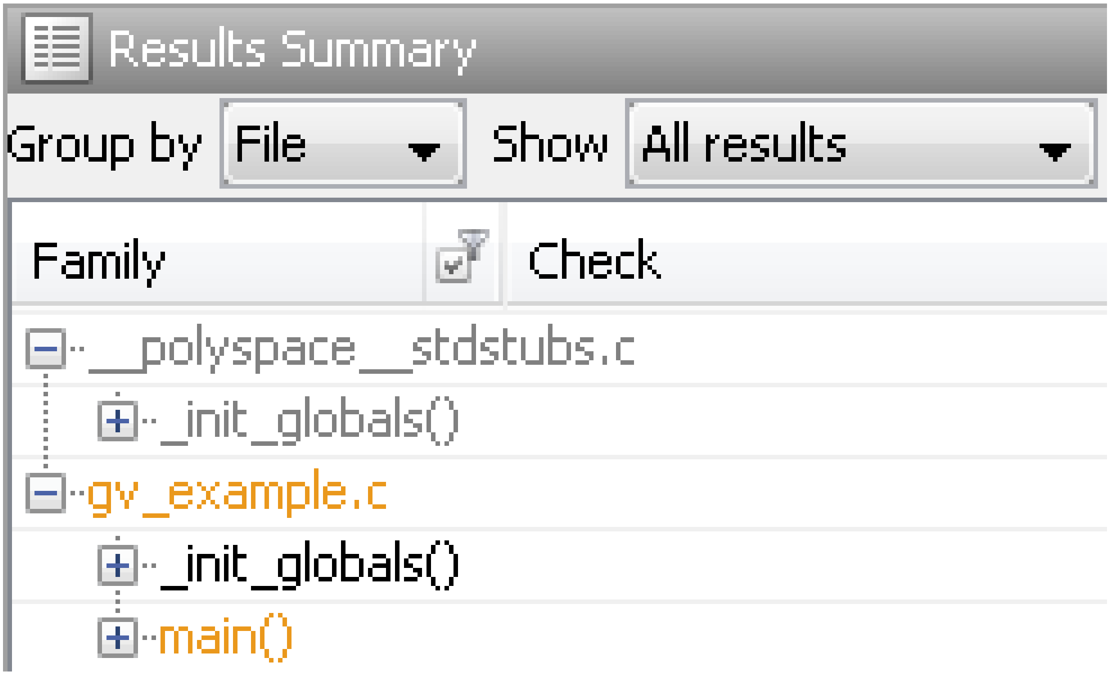
```
extern int func(int);  /* External function */

/* Global variables initialized in _init_globals() */
/* before the execution of main() procedure     */
int garray[3] = {1, 2, 3};
/* Initialized: written in __init_globals() */
int gvar = 12;
/* Initialized: written in __init_globals() */

int main(void) {
 int i, lvar = 0;
 for (i = 0; i < 3; i++)
  lvar += func(garray[i] + gvar);
 return lvar;
}
```

After verification, you see the following:

- On the **Results Summary** pane, if you select **Group by** > **File**, under the node `gv_example.c`, you see `_init_globals`.

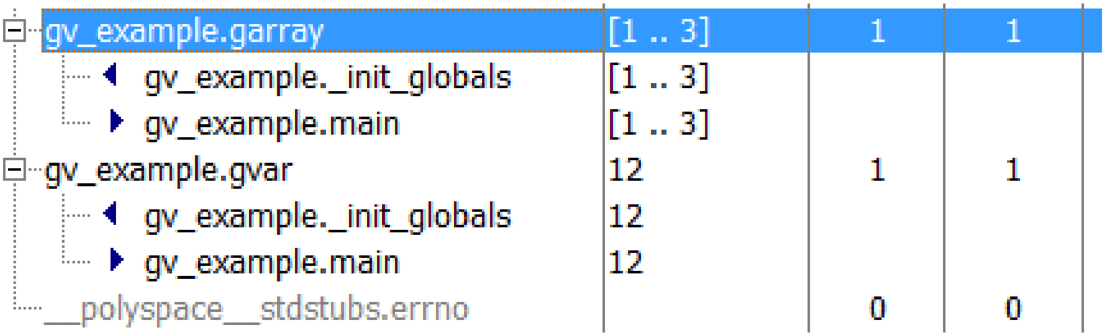


Group by File

Show All results

```
⊟ __polyspace__stdstubs.c
    ⊞ _init_globals()
⊟ gv_example.c
    ⊞ _init_globals()
    ⊞ main()
```

- On the **Variable Access** pane, `gv_example._init_globals` represents the first write operation on a global variable, for example, `garray`. In the **Values** column, the corresponding value represents the value of the global variable after initialization.



gv_example.garray	[1 .. 3]	1	1
gv_example._init_globals	[1 .. 3]		
gv_example.main	[1 .. 3]		
gv_example.gvar	12	1	1
gv_example._init_globals	12		
gv_example.main	12		
__polyspace__stdstubs.errno		0	0

Volatile Variables

The values of volatile variables can change without explicit write operations.

For local volatile variables:

- Polyspace assumes that the variable has the full range of values allowed for their type.
- Unless you explicitly initialize the variable, Polyspace produces an orange **Non-initialized local variable** check when the variable is read.

In the following example, Polyspace assumes that `val1` is potentially non-initialized but `val2` is initialized. However, because Polyspace assumes both variables to have full range of values, it considers that the + operation can cause an overflow.

```
int func (void)
{
    volatile int val1, val2=0;
    return( val1 + val2);
}
```

For global volatile variables:

- Polyspace assumes that the variable has full range of values allowed for their type.

- Even if you do not explicitly initialize the variable, Polyspace produces a green **Non-initialized variable** check when the variable is read.

If the root cause of an orange check is a volatile variable, you cannot override the default assumptions and constrain the values of the volatile variables. Instead 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. See “Constrain Stubbed Functions”.
- 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 Variables”.
- Replace the volatile variable with a stubbed function, but only for verification. Before verification, specify constraints on the stubbed functions. To do this replacement:

  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**. See “Command/script to apply to preprocessed files (C/C++)”.
  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**. See “Include (C/C++)”.

Structures with Volatile Fields

Polyspace treats the fields of a structure as volatile even if only one field is declared with the `volatile` keyword.

In the following example, Polyspace produces an orange **Non-initialized variable** error when `a` is read, even though `a` is not declared with the `volatile` keyword.

```
 typedef struct {
 int a;
  volatile int b;
} volStruct;

void func(int);

void main() {
  volStruct myStruct;
  func(myStruct.a);
  func(myStruct.b);
}
```

Absolute Addresses

Polyspace highlights absolute addresses in your code with an orange **Absolute address** check to indicate one of the following issues:

- The address might not be a valid address
- The address might not have sufficient memory available.

In the following example, `X` is a macro that refers to the content of an absolute address `0x20000`. Each time `X` is used, the software produces an orange **Absolute address** check to highlight the use of an absolute address.

```
#define X (* ((int *)0x20000))


void f1(void)  {
    int y;
    X = 100;
    y = 1 / X;
}
```

In the following example, the software produces an **Absolute address** check only when the address is assigned to a pointer `p`. Following the check, the software considers that the pointer p is initialized and can be dereferenced. Therefore, it produces green **Non-initialized pointer** and **Illegally dereferenced pointer** checks on `*p = 100`.

```
void f2(void) {
    int y;
    int *p = (int *)0x20000;
    *p  = 100;
    y  = 1/ *p;
}
```

For more information, see Absolute address.

External Variables

Polyspace verification works on the principle that a global or static external variable could take any value within the range of its type.

```
extern int x;
void f(void)
int y;
y = 1 / x;  // orange because x ~ [-2^31, 2^31-1]
y = 1 / x;  // green because x ~ [-2^31 -1] U [1, 2^31-1]
```

For more information on color propagation, refer to “Verification Following Red and Orange Checks”.

For external structures containing fields of type “pointer to function”, this principle leads to red errors in the verification results. In this case, the resulting default behavior is that these pointers do not point to any valid function. For meaningful results, you need to define these variables explicitly.

The excessive use of global variables can lead to problems in a design. It is not always apparent which global variables are produced by a given file, or which global variables are used by that file. For example, global variables result in:

- File APIs (or functions accessible from outside the file) without procedure parameters.
- The requirement for a formal list of variables which are produced and used, as well as the theoretical ranges they can take as input and/or output values.

Definitions and Declarations

The definition and declaration of a variable are two different but related operations.

Definition

- If a function is defined, it means that the body of the function has been written: `int f(void) { return 0; }`
- If a variable is defined, it means that a part of memory has been 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. For more information, see “External Variables”.

When a function is not defined, unless you choose the option `none` for **Functions to stub**, the software automatically stubs the function. For more information, see:

- C code: “Inputs & Stubbing”
- C++ code: “Inputs & Stubbing”

Declaration

- Function declaration: `int f(void);`
- Variable declaration: `extern int x;`

A declaration provides information about the type of the function or variable. If the function or variable is used in a file where it has not been declared, a compilation error results.

Types Promotion

- “Unsigned Integers Promoted to Signed Integers” on page 5-11
- “Promotions Rules in Operators” on page 5-12
- “Example” on page 5-13

Unsigned Integers Promoted to Signed Integers

You need to understand the circumstances under which signed integers are promoted to unsigned.

For example, the execution of the following code would produce an assertion failure and a core dump.

```
#include <assert.h>
```

```
int f1(void) {
 int x = -2;
 unsigned int y = 5;
 assert(x <= y);
}
```

Implicit promotion explains this behavior. In this example, `x <= y` is implicitly:

```
((unsigned int) x) <= y /* implicit promotion since y is unsigned */
```

A negative cast into unsigned gives a large value. This value can never be <= 5, so the assertion can never hold true.

In this second example, consider the range of possible values for x:

```
void f2(void)
volatile int random;
unsigned int y = 7;
int x = random;
assert ( x >= -7 && x <= y );

assert (x>=0 && x<=7);
```

The first assertion is orange; it may cause an assert failure. However, given that the range of x after the first assertion is **not [-7 .. 7]**, but rather **[0 .. 7]**, the second assertion would hold true.

Promotions Rules in Operators

Familiarity with the rules applying to the standard operators of the C language helps you to analyze those orange and **red** checks which relate to overflows on type operations. Those rules are:

- Unary operators operate on the type of the operand.
- Shifts operate on the type of the left operand.
- Boolean operators operate on Booleans.
- Other binary operators operate on a common type. If the types of the two operands are different, they are promoted to the first common type which can represent both of them.
- Be careful of constant types.
- Be careful when verifying a operation between variables of different types without an explicit cast.

Example

Consider the integer promotion aspect of the ANSI C standard (see 6.2.1 in ISO/IEC 9899:1990). On arithmetic operators like +, -, *, % and / , an integer promotion is applied on both operands. For verification, that can imply an OVFL or a UNFL orange check.

```
2 extern char random_char(void);
3 extern int random_int(void);
4
5 void main(void)
6 {
7  char c1 = random_char();
8  char c2 = random_char();
9  int i1 = random_int();
10  int i2 = random_int();
11
12  i1 = i1 + i2;   // A typical OVFL/UNFL on a + operator
13  c1 = c1 + c2;   // An OVFL/UNFL warning on the c1
14       // assignment [from int32 to int8]
15 }
```

Unlike the addition of two integers at line 12, an implicit promotion is used in the addition of the two chars at line 13. Consider this second “equivalence” example.

```
2 extern char random_char(void);
3
4 void main(void)
5 {
6  char c1 = random_char();
7  char c2 = random_char();
8
9  c1 = (char)((int)c1 + (int)c2);  // Warning OVFL: due to
10             // integer promotion
11 }
```

An orange check represents a warning of a potential overflow (OVFL), generated on the (char) cast [from int32 to int8]. A green check represents a verification that the + operator does not produce an overflow (OVFL).

Integer promotion requires that the abstract machine must promote the type of each variable to the integral target size before realizing the arithmetic operation and subsequently adjusting the assignment type. See the preceding equivalence example of a simple addition of two *char*.

Integer promotion respects the size hierarchy of basic types:

- *char (signed or not)* and *signed short* are promoted to *int*.
- *unsigned short* is promoted to *int* only if *int* can represent all possible values of an *unsigned short*. If that is not the case (because of a 16-bit target, for example) then *unsigned short* is promoted to *unsigned int*.
- Other types such as*(un)signed int*, *(un)signed long int*, and *(un)signed long long int* promote themselves.

Using `memset` and `memcpy`

- "Polyspace Specifications for `memcpy`" on page 5-14
- "Polyspace Specifications for `memset`" on page 5-16

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`.	In the following 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`.
	`#include <string.h>`
	`typedef struct {`
	`    char a;`
	`    int b;`
	` } S;`
	
	`void func(int);`

<table>
<tr><th>Specification</th><th>Example</th></tr>
<tr><td></td><td>

```
void main() {
  S s;
  int d;
  memcpy(&d, &s, sizeof(S));
}
```

</td></tr>
<tr><td>

Polyspace does not check if the memory that `sourcePtr` points to is itself initialized.

</td><td>

In the following code, Polyspace does not produce a red **Non-initialized local variable** error when the `memcpy` function copies `s` to `d`.

```
#include <string.h>
typedef struct {
    char a;
    int b;
 } S;

void func(int);

void main() {
  S s, d;
  memcpy(&d, &s, sizeof(S));
  func(d.b);
}
```

</td></tr>
</table>

<table>
<tr><th>Specification</th><th>Example</th></tr>
<tr><td>Following the use of memcpy, Polyspace considers that the variables that destinationPtr points to can have any value allowed by their type.</td><td>In the following code, Polyspace 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.

```
#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
}
```
</td></tr>
</table>

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`.	In the following 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)`.

Specification	Example

	• A memory block of size `sizeof(S)` is assigned to `&val`.
	
	`#include <string.h>`
	`typedef struct {`
	`    char a;`
	`    int b;`
	`} S;`
	
	`void main() {`
	` int val;`
	` memset(&val,0,sizeof(S));`
	`}`

If `value` is 0, following the use of `memset`, Polyspace considers that the variables that `ptr` points to have the value 0.	In the following code, Polyspace considers that following the use of `memset`, each field of `s` has 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`
	`}`

Specification	Example

If `value` is anything other than 0, following the use of `memset`, Polyspace considers that:
• The variables that `ptr` points to can be non-initialized.
• If initialized, the variables can have any value allowed by their type.
	In the following code, Polyspace considers that following the use of `memset`, each field of `s` has any value allowed by its type. For instance, `s.b` can have any value in the range allowed for an `int` variable.
	`#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`
	`}`

Shared Variables

At a minimum, the range of a shared variable is the union of all ranges of the variable in the application. At a maximum, the variable is full range.

```
void p_task1(void)
{
 begin_cs();
 X = 0;
 if (X) {
  Y = X;      // Verified NIV, although it should be gray
  assert (Y == 12);  // Warning assert, although it should be gray
 }
 end_cs();
}

void p_task2(void)
{
 begin_cs();
 X = 12;
 Y = X + 1;      // Polyspace considers [Y==1] or [Y==13]
 if (Y == 13)
  Y = 14;
 else
  Y = X - 1 ;    // this line should be gray
 end_cs();
}
```

Trigonometric Functions

With trigonometric functions, such as sines and cosines, verification sometimes assumes that the return value is bound between the limits of that function, regardless of the parameter passed to it. Consider the following example, which uses acos, sin and asin functions.

```
double res;

res = sin(3.141592654);
assert(res == 0.0); // Range is [-1..1]

res = acos(0.0);
assert(res == 0.0); // Range always in [0..pi]

res = asin(0.0);
assert(res == 0.0); // Always gives [0.0]
```

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 might be inserted at the end of an union.
- **Alignment** – Members of structures within a union might have different alignments.
- **Endianness** – Whether the most significant byte of a word could be stored at the lowest or highest memory address.
- **Bit-order** – Bits within bytes could have both different numbering and allocation to bit fields.

Because of these issues, Polyspace verification can lose precision when unions are used in your code.

For example, if you write a union member, but read back another union member, Polyspace considers that the latter member can have any value allowed by its type. In the following 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 an union using a static initializer, following ANSI C standard, Polyspace considers that the union member appearing first in the declaration list gets initialized.

Constant Pointer

To increase Polyspace precision where pointers are analyzed, replace

```
const int *p = &y;
```

with:

```
#define p (&y)
```

Variable Cast as Void Pointer

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 the following code:

```
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` should be assigned the value 5. However, the software treats `p->x1` as full-range.

In some cases, you can avoid this loss of precision by running your verification with the option `-retype-pointer`. For this example, if you specify `-retype-pointer`, the software assigns the value 5 to `x` in the function `g`.

Assembly Code

- “Ignored Inline Assemblers” on page 5-21
- “Single Function Containing Assembly Code” on page 5-24
- “Multiple Functions Containing Assembly Code” on page 5-25
- “Local Variables in Functions with Assembly Code” on page 5-26

Ignored Inline Assemblers

Polyspace recognizes the following inline assemblers as introduction of assembly code. During verification, it ignores the assembly code introduced by these assemblers.

- 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 automatically 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)
```

The following code provides 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

In functions containing assembly code, the software treats local variables that are not explicitly initialized as potentially initialized variables.

Consider the following function.

```
inline int f(void) {
  int r;
  asm("mov 4%0,%%eax"::"m"(r));
  return r;     // orange NIVL because r is not initialized
}
```

The software treats r as a potentially initialized variable. Verification generates an orange NIVL check for r.

Consider another function.

```
1  int dummy(void) {
2    int g,h;
3    h = g * 2;      // orange NIVL for g (red NIVL before 12a)
4    h = 2;           // h is assigned the value 2
5    asm("int $0x3");
6    asm("mov 4%0,%%eax"::"m"(g));
7    asm("movss 4%0,%%xmm1"::"m");
8    return h;       // value returned is 2
9  }
```

In line 3, the variable g is not initialized. Verification:

- Generates an orange NIVL check for g.
- Assigns a full-range value to g.

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:

```
MATLAB_Install\polyspace\verifier\code_prover
```

6

Examples

Scripts for Command-Line Verification

The following sections contain sample scripts that you can use to run a Polyspace verification from the DOS or UNIX command line. For information on the general workflow, see:

- "Run Local Verification at Command Line"
- "Run Remote Analysis at Command Line"

In this section...

Simple C Example

```
polyspace-code-prover-nodesktop \
 -prog myCproject \
 -O1 \
 -I /home/user/includes \
 -D SUN4 -D USE_FILES \
```

Apache Example

Here is a script for verifying the code for Apache (after formatting). The source code is in C and the compilation is for an Oracle® Sun™ Microsystems SPARC® processor.

Note: The use of O0 to reduce verification time.

```
polyspace-code-prover-nodesktop \ \
 -target sparc \
 -prog Apache \
 -keep-all-files \
 -continue-with-red-error \
 -O0 \
 -D PST \
 -D __GNUC_MINOR__=6 -D SOLARIS2=270 -D USE_EXPAT \
 -D NO_DL_NEEDED \
 -I sources \
 -I /usr/local/pst/include.sparc \
 -I /usr/include \
 -results-dir RESULTS
```

cxref Example

Here is another C launch command. The compilation is for Linux. Note the escape characters, allowing quoted strings to be used as compiler defines.

```
polyspace-code-prover-nodesktop \
 -OS-target linux \
 -prog cxref \
 -O0 \
 -I `pwd` \
 -I sources \
 -I <Polyspace_Install>/include/include.linux \
 -D CXREF_CPP='\"/usr/local/gcc/bin/cpp\"' \
 -D PAGE='\"A4\"' \
 -results-dir RESULTS
```

T31 Example

Another simple C launcher. There are a couple of tasks and compilation is for an m68k.

```
polyspace-code-prover-nodesktop \
 -target m68k \
 -entry-points task_callback_main,task_tcp_main,cdtask_depm_main,
task_receiver \
 -to pass1 \
 -prog T31 \
 -O0 \
 -results-dir `pwd`/RESULTS_31 \
```

Dishwasher1 Example

Another C example. This one is for the c-167 and has tasks protected by critical section.

```
polyspace-code-prover-nodesktop \
 -target c-167 \
 -entry-points periodic,pst_main \
 -D PST -D const= -D water= \
 -from scratch \
 -to pass4 \
 -critical-section-begin "critical_enter:cs1" \
 -critical-section-end "critical_exit:cs1" \
 -prog dishwasher1 \
 -I `pwd`/sources \
 -O0 \
 -results-dir RESULTS
```

Satellite Example

A C example with tasks and critical sections.

```
polyspace-code-prover-nodesktop
 -target c-167 \
 -entry-points ctask0,ctask1,ctask2,ctask3,interrupts \
 -O2 \
 -keep-all-files \
 -from scratch \
 -critical-section-begin "DisableInterrupts:sc1" \
 -critical-section-end "EnableInterrupts:sc1" \
 -ignore-constant-overflows \
 -include `pwd`/sources/options.h \
 -to pass4 \
 -prog satellite \
 -I `pwd`/sources \
 -results-dir RESULTS
```

7

Functions

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 priority classification, an action status, or other comments using Name,Value pairs.

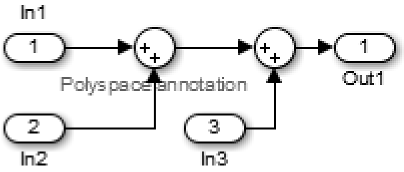
In the generated code associated with the annotated block, Polyspace adds code comments before and after the lines of code. Polyspace reads these comments and marks Polyspace results of the specified `kind` with the annotated information.

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:Fix>*/
  annotate_y.Out1=(annotate_u.In1+annotate_U.In2)+annotate_U.In3;
  /* polyspace:end<RTE:OVFL:Medium: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. If analysis results do not exist for systemName, Polyspace opens to the Project Browser pane.

`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.

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_v2, 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_v2';
open(model)
gcb = 'WhereAreTheErrors_v2/10* x // (x-y)/Divide';
pslinkfun('annotations','type','RTE','kind','ZDV','status',...
 'justify with annotation','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_v2 and open the advanced options window.

```
model = 'WhereAreTheErrors_v2';
```

```
load_system(model)
pslinkfun('advancedoptions')
```

In the **Distributed Computing** pane, select the options **Batch** and **Add to results repository**.

Set the configuration template for new Polyspace analyses to have these options.

```
pslinkfun('settemplate',fullfile(cd,'pslink_config',...
    'WhereAreTheErrors_v2_config.psprj'))
```

View the current Polyspace template.

```
template = pslinkfun('gettemplate')
```

```
template =
C:\ModelLinkDemo\pslink_config\WhereAreTheErrors_v2_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_v2`, create a Polyspace options object, set the verification mode, and open the advanced options window.

```
model = 'WhereAreTheErrors_v2';
load_system(model)
slbuild(model)
opts = pslinkoptions(model);
opts.VerificationMode = 'CodeProver';
pslinkfun('advancedoptions')
```

In the **Distributed Computing** pane, select the **Batch** and **Add to results repository** options.

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 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 Rules”.

type Value	kind Values
'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 "Supported MISRA C++ Coding 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_v2')`

`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(matlabroot,`
`'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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'block','MyModel\Sum', 'status','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'` — classification of the check

`'high'` | `'medium'` | `'low'` | `'not a defect'` | `'unset'`

Classification of the check specified as `high`, `medium`, `low`, `not a defect`, or `unset`.

Example: `'class','high'`

`'status'` — action status

`'undecided'` | `'investigate'` | `'fix'` | `'improve'` | `'restart with different options'` | `'justify with annotation'` | `'no action planned'` | `'other'`

Action status of the check specified as `undecided`, `investigate`, `fix`, `improve`, `restart with different options`, `justify with annotation`, `no action planned`, or `other`.

The statuses, `justify with annotation` and `no action planned`, also mark the result as justified.

Example: `'status','no action planned'`

`'comment'` — additional comments

string

Additional comments specified as a string. 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 options object to customize Polyspace runs from MATLAB command line

Syntax

```
opts = pslinkoptions(codegen)
opts = pslinkoptions(model)
```

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.

Examples

Use a Simulink model to create and edit an options objects

Load `psdemo_model_link_sl` and create a Polyspace options object from the model:

```
load_system('psdemo_model_link_sl_v2');
model_opt = pslinkoptions('psdemo_model_link_sl_v2')
```

```
model_opt =

                    ResultDir: 'results_$ModelName$'
         VerificationSettings: 'PrjConfig'
           OpenProjectManager: 0
         AddSuffixToResultDir: 0
     EnableAdditionalFileList: 0
           AdditionalFileList: {}
             VerificationMode: 'CodeProver'
          EnablePrjConfigFile: 0
                PrjConfigFile: ''
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
```

```
              OutputRangeMode: 'None'
           ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
```

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: ''
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
           ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
```

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: ''
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
           ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
      CxxVerificationSettings: 'PrjConfig'
    CheckConfigBeforeAnalysis: 'OnWarn'
```

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: ''
               InputRangeMode: 'DesignMinMax'
               ParamRangeMode: 'None'
              OutputRangeMode: 'None'
           ModelRefVerifDepth: 'Current model only'
      ModelRefByModelRefVerif: 0
      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: ''
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
                 AutoStubLUT: 0
```

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: ''
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
                 AutoStubLUT: 0
```

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 Properties.

Example: `ec_opt = pslinkoptions('ec')`

Example: `tl_opt = pslinkoptions('tl')`

Data Types: `char`

`model` — Simulink model

`model name`

Simulink model, specified by the model name. Creates a Polyspace options object with the configuration options of that model. If you do not set any options, the object has the default configuration options. If a code generator has been set, the object has the default options for that code generator.

For a description of all configuration options and their values, see pslinkoptions Properties.

Example: `model_opt = pslinkoptions('my_model')`

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 Properties.

Example:
```
opts= pslinkoptions('ec')
opts.VerificationSettings = 'Misra'
```

More About

- pslinkoptions Properties

See Also

`pslinkfun` | `pslinkrun`

pslinkrun

Run Polyspace analysis on generated code from MATLAB command line

Syntax

```
resultsFolder = pslinkrun
resultsFolder = pslinkrun(system)
resultsFolder = pslinkrun(system,opts)
resultsFolder = pslinkrun(system,opts,asModelRef)
```

Description

`resultsFolder = pslinkrun` on generated code from the current system and returns the location of the results folder. It uses the analysis options associated with the current system. The current system, or model, is the system returned by the command `bdroot`.

`resultsFolder = pslinkrun(system)` runs Polyspace on the code generated from the model or subsystem specified by system. It uses the analysis options associated with system.

`resultsFolder = pslinkrun(system,opts)` analyzes system using the analysis options from the options object opts.

`resultsFolder = pslinkrun(system,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 only a referenced model instead of an entire model hierarchy.

Examples

Run Polyspace from the Command Line

Use a Simulink model to generate code, set configuration options, and then run an analysis from the command line.

Load and build the model `WhereAreTheErrors_v2` to generate code.

```
model = 'WhereAreTheErrors_v2';
```

```
load_system(model)
slbuild(model)
```

Create a Polyspace options object from the model and change the configuration to run a Code Prover verification.

```
opts = pslinkoptions(model);
opts.VerificationMode = 'CodeProver';
```

Run Polyspace using your options object:

```
results = pslinkrun(model,opts)
```

The results are saved to the `results_WhereAreTheErrors_v2` folder, listed in the `results` variable.

Build and Analyze Referenced Model Code from the Command Line

Use a Simulink model to generate reference code, set configuration options, and then run an analysis from the command line.

Load and build the model `WhereAreTheErrors_v2` to generate code as if it is referenced by another model:

```
model = 'WhereAreTheErrors_v2';
load_system(model)
slbuild(model,'ModelReferenceRTWTargetOnly')
```

Create a Polyspace options object from the model and change the configuration to run a Code Prover verification.

```
opts = pslinkoptions(model);
opts.VerificationMode = 'CodeProver';
```

Run Polyspace using your options object:

```
results = pslinkrun(model,opts,true)
```

The results are saved to the `results_mr_WhereAreTheErrors_v2` folder, listed in the `results` variable.

Input Arguments

`system` — Model or system

`bdroot` (default) | model or system name

Model or system that you want to analyze, specified as a string, with the model or system name in single quotes. The default value is the system returned by `bdroot`.

Example: `resultsFolder = pslinkrun('demo')` where `demo` is the name of a model.

Data Types: `char`

`opts` — Analysis options

options associated with system (default) | Polyspace options object

Analysis options for the analysis, specified as an options object or the options already associated with the model or system. The function `pslinkoptions` creates an options object. You can customize the options object by changing the

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 code generated as standalone code. This option is equivalent to choosing **Verify Code Generated For** > **Model** in the Simulink Polyspace options.
- If asModelRef is true, Polyspace analyzes code generated as model referenced code. This option is equivalent to choosing **Verify Code Generated For** > **Referenced Model** in the Simulink Polyspace options.

Data Types: `logical`

Output Arguments

`resultsFolder` — Variable for location of the results folder

string

Variable for location of the results folder, specified as a string. The default value of this variable is `results_$ModelName$`. You can change this value in the configuration options using `pslinkoptions`.

Data Types: `char`

See Also

pslinkfun | pslinkoptions

polyspaceCodeProver

Run Polyspace Code Prover verification from MATLAB

Syntax

```
polyspaceCodeProver
```

```
polyspaceCodeProver(projectFile)
polyspaceCodeProver(resultsFile)
polyspaceCodeProver('-results-dir',resultsFolder)
```

```
polyspaceCodeProver('-help')
```

```
polyspaceCodeProver('-sources',sourceFiles)
polyspaceCodeProver('-sources',sourceFiles,Name,Value)
```

Description

`polyspaceCodeProver` opens Polyspace Code Prover.

`polyspaceCodeProver(projectFile)` opens a Polyspace project file in Polyspace Code Prover.

`polyspaceCodeProver(resultsFile)` opens a Polyspace results file in Polyspace Code Prover.

`polyspaceCodeProver('-results-dir',resultsFolder)` opens a Polyspace results file from resultsFolder in Polyspace Code Prover.

`polyspaceCodeProver('-help')` displays all options that can be supplied to the `polyspaceCodeProver` command to run a Polyspace Code Prover verification.

`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.

Examples

Open Polyspace Projects from MATLAB

This example shows how to open a Polyspace project file with extension `.psprj` from MATLAB. In this example, you open the project file `Demo_C.psprj` from the folder *`Matlab_Install`*`\polyspace\examples\cxx\Demo_C`.

Assign the full path to the project file to a MATLAB variable `prjFile`.

```
prjFile = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', ...
         'Demo_C', 'Demo_C.psprj');
```

Use `prjFile` to 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 *`Matlab_Install`*`\polyspace\examples\cxx\Demo_C\Module_1\Result_1`.

Assign the full path to the folder to a MATLAB variable `resFolder`.

```
resFolder = fullfile(matlabroot, 'polyspace', 'examples', ...
       'cxx', 'Demo_C', 'Module_1', 'Result_1');
```

Use `resFolder` to open the results.

```
polyspaceCodeProver('-results-dir',resFolder)
```

Run Polyspace Verification from MATLAB

This example shows how to run a Polyspace verification on a single source file from the MATLAB command-line. For this example:

- Save a C source file, `source.c`, in the folder `C:\Polyspace_Sources`.
- Save an include file in the folder `C:\Polyspace_Includes`.

Run the following command on the MATLAB command line.

```
polyspaceCodeProver('-sources','C:\Polyspace_Sources\source.c', ...
           '-I','C:\Polyspace_Includes', ...
           '-results-dir','C:\Polyspace_Results')
```

Polyspace runs on the file `C:\Polyspace_Sources\source.c` and stores the result in `C:\Polyspace_Results`.

To view the results from the MATLAB command line, enter:

```
polyspaceCodeProver('-results-dir','C:\')
```

Run Polyspace Verification with Coding Rules Checking

This example shows how to run a Polyspace verification with additional options. You can specify as many additional options as you want as "Name-Value Pair Arguments" on page 7-23. Here you specify:

- Checking of MISRA C coding rules using the option `-misra2`. For more information, see "Check MISRA C:2004".
- Excluding header files from coding rules checking using the option `-includes-to-ignore`. For more information, see "Files and folders to ignore (C)".
- Automatic generation of `main` function using the option `-main-generator`. For more information, see "Verify module (C)".

Assign the source file path to a MATLAB variable `sourceFileName`.

```
sourceFileName = fullfile(matlabroot, 'polyspace',...
'examples', 'cxx', 'Demo_C_Single-File','sources','example.c')
```

Assign the include file path to a MATLAB variable `includeFileName`.

```
includeFileName = fullfile(matlabroot, 'polyspace',...
'examples', 'cxx', 'Demo_C_Single-File','sources','include.h')
```

Assign the results folder path to a MATLAB variable `resFolder`.

```
resFolder = fullfile('C:\','Polyspace_Results')
```

Run Polyspace Code Prover verification with additional options `-misra2`, `-includes-to-ignore` and `-main-generator`.

```
polyspaceCodeProver('-sources',sourceFileName,...
     '-I',includeFileName, ...
     '-results-dir',resFolder,'-misra2','required-rules',...
```

```
    '-includes-to-ignore','all-headers','-main-generator')
```

Open the results file.

```
polyspaceCodeProver('-results-dir',resFolder)
```

Input Arguments

`projectFile` — Name of `.psprj` file

string

Name of project file with extension `.psprj`, specified as a string.

If the file is not in the current folder, `projectFile` must include a full or relative path. Use `pwd` to identify the current folder and `cd` to change the current folder.

Example: `'C:\Polyspace_Projects\myProject.psprj'`

`resultsFile` — Name of `.pscp` file

string

Name of results file with extension `.pscp`, specified as a string.

If the file is not in the current folder, `resultsFile` must include a full or relative path. Use `pwd` to identify the current folder and `cd` to change the current folder.

Example: `'myResults.psbf'`

`resultsFolder` — Name of result folder

string

Name of result folder, specified as a string. The folder must contain the results file with extension `.psbf`. 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. Use `pwd` to identify the current folder and `cd` to change the current folder.

Example: `'C:\Polyspace\Results\'`

`sourceFiles` — Comma-separated names of `.c` or `.cpp` files

string

Comma-separated source file names with extension .c or .cpp, specified as a single string.

If the files are not in the current folder, sourceFiles must include a full or relative path. Use pwd to identify the current folder and cd to change the current folder.

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 single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: '-OS-target','Linux','-dialect','gnu4.6' specifies that the source code is intended for the Linux operating system and contains non-ANSI C syntax for the GCC 4.6 dialect.

For the full list of analysis options, see "Analysis Options".

polyspaceConfigure

Create Polyspace project from your build system at the MATLAB command line

Syntax

`polyspaceConfigure buildCommand`

`polyspaceConfigure buildCommand -option value`

Description

`polyspaceConfigure buildCommand` traces your build system and creates a Polyspace project with information gathered from your build system.

`polyspaceConfigure buildCommand -option value` traces your build system and uses the flag `-option value` to modify the default operation of `polyspaceConfigure`.

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.

```
polyspaceConfigure  -prog myProject ...
         make -B targetName buildOptions
```

Open the Polyspace project in the **Project Browser**.

```
polyspaceCodeProver('myProject.psprj')
```

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 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 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  -no-project -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
```

Trace Incremental Makefile Builds

This example shows how to trace incremental makefile builds to keep your Polyspace project updated. If you use this approach, polyspaceConfigure does not have to trace the entire makefile every time you make a change to it.

Create a Polyspace project from your makefile using polyspaceConfigure. For this first project creation:

- Use the -B or -W *makefileName* option with make so that all prerequisite targets in the makefile are remade.

  For the list of options allowed with the GNU make, see make options.
- Use the -incremental option so that the build trace information is saved.

```
polyspaceConfigure -prog myProject ...
        -incremental make -B targetName buildOptions
```

After you add, remove or change source files, to keep your Polyspace project updated, rerun polyspaceConfigure with the same options. Do not use the -B or -W *makefileName* option with make.

```
polyspaceConfigure -prog myProject ...
        -incremental make targetName buildOptions
```

The polyspaceConfigure function uses the previous build trace information to incrementally add or remove the updated files to your Polyspace project. It does not trace the entire makefile.

- “Create Project Automatically”

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
`-author`	Author name	Name of project author. **Example:** `-author jsmith`
`-code-prover` (default) | `-bug-finder`	None	Option to create a Polyspace Bug Finder™ or Polyspace Code Prover project.
`-debug`	None	Option used by MathWorks technical support
`-help`	None	Option to display the full list of `polyspaceConfigure` commands
`-lang`	`auto`(default) |`c`|`cpp`	Option to specify source code language. By default,`polyspaceConfigure` detects the language. If it detects a mixture of languages in the compilation units, it assigns C++ as the project language. If it detects the use of C++11, it allows C++11 extensions.
`-output-options-file`	None	Option to create a Polyspace analysis options file. Use this file for command-line analysis using `polyspaceCodeProver`.
`-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`

Option	Argument	Description
`-prog`	Project name	Project name that appears in the Polyspace user interface. The default is `polyspace`. **Example:** `-prog myProject`
`-silent` (default) | `-verbose`	None	Option to suppress or display additional messages from running `polyspaceConfigure`.

Advanced Options

Option	Argument	Description
`-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 *`matlabroot`*`\polyspace\configure\compiler_configuration\`. For information on the contents of the file, see "Your Compiler Is Not Supported". **Example:** `-compiler-configuration myCompiler.xml`
`-incremental`	None	Option to save build trace information for reuse in incremental builds
`-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 `polyspaceConfigure` with the `-no-project` option. If you use this option, you do not need to specify the buildCommand argument.
`-no-project`	None	Option to trace your build system without creating a Polyspace project and save the build trace information.

Option	Argument	Description
		Use this option to save your build trace information for a later run of `polyspaceConfigure` with the `-no-build` option.
`-tmp-path`	Path	Location of folder where temporary files are stored.

Cache Control Options

Option	Argument	Description
`-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`
`-no-cache` | `-cache-sources` (default) | `-cache-all-files`	None	Option to perform one of the following: • Not create a cache • Cache only source and header files. • Cache all files including binaries.
`-cache-path`	Path	Location of folder where cache information is stored. **Example:** `-cache-path ../cache`

More About

- "Requirements for Project Creation from Build Systems"
- "Your Compiler Is Not Supported"

polyspaceJobsManager

Manage Polyspace jobs on MDCS 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,'-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,'-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 MJS scheduler to run Polyspace remotely and monitor your jobs through the queue.

Before performing this example, set up an MJS 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.

```
mkdir 'C:\psdemo\src'
demo = fullfile(matlabroot,'polyspace','examples','cxx',...
'Demo_C','sources');
copyfile(demo,'C:\psdemo\src\')
```

Submit two jobs to your scheduler.

```
polyspaceCodeProver -batch -scheduler myMJS@myCompany.com
    -sources C:\psdemo\src\*.c'
    -results-dir 'C:\psdemo\res1'
polyspaceCodeProver -batch -scheduler myMJS@myCompany.com
    -sources 'C:\psdemo\src\main.c'
    -results-dir 'C:\psdemo\res2'
    -add-to-results-repository
polyspaceJobsManager('listjobs','-scheduler','myMJS@myCompany.com')
```

```
ID AUTHOR APPLICATION LOCAL_RESULTS_DIR WORKER STATUS DATE LANG CLUSTER_MODE
...
19 user Polyspace C:\psdemo\res1  queued Wed Mar 16 16:48:38 EST 2014 C Batch
20 user Polyspace C:\psdemo\res2  queued Wed Mar 16 16:48:38 EST 2014 C Batch
```

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')
```

```
ID AUTHOR APPLICATION LOCAL_RESULTS_DIR WORKER STATUS DATE LANG CLUSTER_MODE
...
19 user Polyspace C:\psdemo\res1  cancelled Wed Mar 16 16:48:38 EST 2014 C Batch
20 user Polyspace C:\psdemo\res2  running Wed Mar 16 16:48:38 EST 2014 C Batch
```

Remove job 19.

```
polyspaceJobsManager('remove','-job','19','-scheduler',...
   'myMJS@myCompany.com')
polyspaceJobsManager('listjobs','-scheduler','myMJS@myCompany.com')
```

```
ID AUTHOR APPLICATION LOCAL_RESULTS_DIR WORKER STATUS DATE LANG CLUSTER_MODE
...
20 user Polyspace C:\psdemo\res2  completed Wed Mar 16 16:48:38 EST 2014 C Batch
```

Get the log for job 20.

```
polyspaceJobsManager('getlog','-job','20','-scheduler',...
   'myMJS@myCompany.com')
```

Download the information from job 20.

```
polyspaceJobsManager('download','-job','20','-results-folder', ...
   'C:\psdemo\res3','-scheduler','myCluster')
```

Input Arguments

`jobNumber` — Queued job number

string

Number of the queued job that you want to manage, specified as a string in single quotes.

Example: `'-job','10'`

`resultsFolder` — Path to results folder

string

Path to results folder specified as a string in single quotes. This folder stores the downloaded results files.

Example: `'-results-folder','C:\psdemo\myresults'`

`scheduler` — job scheduler

head node of your MDCS 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 MDCS cluster (*`NodeHost`*).
- Name of the MJS on the head node host (*`MJSName@NodeHost`*).
- Name of a MATLAB cluster profile (*`ClusterProfile`*).

Example: `'-scheduler','myscheduler@mycompany.com'`

More About

- “Clusters and Cluster Profiles”
- “Run Remote Analysis at Command Line”

See Also

`polyspaceCodeProver`

PolyspaceAnnotation

Annotate Simulink blocks with known Polyspace results

Compatibility

`PolyspaceAnnotation` will be removed in a future release. Use `pslinkfun('annotations',...)` instead.

Syntax

```
PolyspaceAnnotation('type',typeValue,'kind',kindValue,Name,Value)
```

Description

`PolyspaceAnnotation('type',typeValue,'kind',kindValue,Name,Value)`adds an annotation of type typeValue and kind kindValue to the currently selected block in the model. You can also specify a different block using a Name,Value pair argument. You can also add notes about a priority classification, an action status, or other comments using Name,Value pairs.

In the generated code associated with the annotated block, code comments are added before and after the lines of code. Polyspace reads these comments and marks Polyspace results of the specified `kind` with the annotated information.

When you add annotations, you can identify known errors and coding rule violations to focus on new results.

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.

At the MATLAB command line, load and open the example model WhereAreTheErrors_v2:

```
open(WhereAreTheErrors_v2)
```

Set the current block to the division block of the `10* x // (x-y)` subsystem:

```
gcb = 'WhereAreTheErrors_v2/10* x // (x-y)/Divide';
```

Add an annotation to the current block to mark division by zero (DIV) errors as justified with the annotation.

```
PolyspaceAnnotation('type','RTE','kind','ZDV','status',...
 'justify with annotation','comment','verified not an error')
```

In Simulink, the division block of the `10* x // (x-y)` subsystem now has a Polyspace annotation.

Back at the MATLAB command line, generate code for the model:

```
slbuild('WhereAreTheErrors_v2')
```

Run a Polyspace Code Prover verification on your model:

```
pslinkrun('WhereAreTheErrors_v2')
```

After the analysis has finished, open the result in the Polyspace environment:

```
PolySpaceViewer('WhereAreTheErrors_v2')
```

If you look at orange division by zero error, the check is justified and includes the status and comments from your annotation.

Annotate a Block and Run a Polyspace Bug Finder Analysis

Use the Polyspace annotation function to annotate a block and see the annotation in the analysis results.

At the MATLAB command line, load and open the example model WhereAreTheErrors_v2:

```
WhereAreTheErrors_v2
```

Add an annotation to the switch block to annotate violations to MISRA C rule 13.7. Also, add to the annotation a comment, a classification, and a status.

```
PolyspaceAnnotation('type','Misra-C', 'kind', '13.7','block',...
'WhereAreTheErrors_v2/Switch1','status','improve','comment','look into later');
```

In the WhereAreTheErrors_v2 model in Simulink, you can see a Polyspace annotation added to the switch block.

At the MATLAB command line, generate code for the model:

```
slbuild('WhereAreTheErrors_v2')
```

Run an analysis on your model:

```
pslinkrun('WhereAreTheErrors_v2')
```

After the analysis is finished, open the results in the Polyspace environment:

```
PolySpaceViewer('WhereAreTheErrors_v2')
```

Results 10–14 are MISRA C 13.7 rule violations. The annotation information that you added to the switch block appears in these four results, because all four results are from the switch block.

Input Arguments

`typeValue` — type of result

`'RTE'` | `'MISRA-C'` | `'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-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 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 “Supported MISRA C++ Coding 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: `PolyspaceAnnotation('type','MISRA-CPP','kind','1-2-3')`

Data Types: `char`

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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: ‘block’,’MyModel\Sum’, ‘status’,’fix’

`'block'` — block to be annotated

`gcb` (default) | block name

Block to be annotated 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'` — classification of the check

`'high'` | `'medium'` | `'low'` | `'not a defect'` | `'unset'`

Classification of the check specified as `high`, `medium`, `low`, `not a defect`, or `unset`.

Example: `'class','high'`

`'status'` — action status

`'undecided'` | `'investigate'` | `'fix'` | `'improve'` | `'restart with different options'` | `'justify with annotation'` | `'no action planned'` | `'other'`

Action status of the check specified as `undecided`, `investigate`, `fix`, `improve`, `restart with different options`, `justify with annotation`, `no action planned`, or `other`.

The statuses, `justify with annotation` and `no action planned`, also mark the result as justified.

Example: `'status','no action planned'`

`'comment'` — additional comments

string

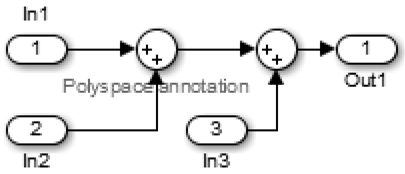
Additional comments specified as a string. The comments provide more information about why the results are justified.

Example: `'comment','defensive code'`

Limitations

- You can have only one annotation per block. If a block produces both a rule violation and an error, only one type can be annotation.
- Even though you apply annotations to individual blocks, the scope of the annotation may be larger. The generated code from one block can overlap with another causing the annotation to also overlap.

  For example, consider this model and its associated generated code.



```
/*
* polyspace:begin<RTE:OVFL:Medium:Fix>
*/
annotate_y.Out1 = (annotate_u.In1 + annotate_U.In2) + annotate_U.In3;

/* polyspace:end<RTE:OVFL:Medium:Fix> */
```

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. Therefore, the annotation justifies both summations

See Also

pslinkoptions | pslinkrun | PolySpaceViewer | gcb

PolySpaceViewer

Open analysis results in the Polyspace environment

Compatibility

`PolySpaceViewer` will be removed in a future release. Use `pslinkfun('openresults',...)` instead.

Syntax

```
PolySpaceViewer(system)
```

Description

`PolySpaceViewer(system)` opens the Polyspace results associated with the model or subsystem system in the Polyspace environment. If system has not been analyzed, Polyspace opens to the **Project Browser** pane.

Examples

Open Results in the Polyspace environment from the Command Line

Use the preconfigured model `WhereAreTheErrors_v2` to run a Polyspace analysis and open the results in the Polyspace environment.

Load the model `WhereAreTheErrors_v2`:

```
load_system('WhereAreTheErrors_v2')
```

Open the Polyspace Viewer:

```
PolySpaceViewer('WhereAreTheErrors_v2')
```

The Polyspace environment opens to the **Project Browser** pane because the model does not yet have Polyspace results.

Build the model to generate C code:

```
slbuild('WhereAreTheErrors_v2');
```

Create a Polyspace options object to set the configuration options:

```
config = pslinkoptions('WhereAreTheErrors_v2')
```

```
config =

                   ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 0
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'CodeProver'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
     CxxVerificationSettings: 'PrjConfig'
```

Change the analysis options to also check for MISRA coding rule violations:

```
config.VerificationSettings = 'PrjConfigAndMisra';
```

Run Polyspace on `WhereAreTheErrors_v2` using the configuration options object that you created:

```
pslinkrun('WhereAreTheErrors_v2', config);
```

Open the results in the Polyspace user interface:

```
PolySpaceViewer('WhereAreTheErrors_v2');
```

The analysis results of `WhereAreTheErrors_v2` appear in the Polyspace user interface.

Input Arguments

`system` — Simulink model

system | subsystem

Simulink model specified by the system or subsystem name.

Example: `PolySpaceViewer('myModel')`

See Also

`pslinkoptions` | `pslinkrun` | `PolyspaceAnnotation`

pslinkoptions Properties

Properties for the `pslinkoptions` object

Before running Polyspace from the command-line, use these properties to customize your analysis.

Analysis Configuration

`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'` – Use all options from the project configuration.
- `'PrjConfigAndMisraAGC'` – Use all options from the project configuration and enable MISRA AC AGC rule checking.
- `'PrjConfigAndMisra'` – Use all options from the project configuration and enable MISRA C:2004 rule checking.
- `'PrjConfigAndMisraC2012'` – Use all 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 `.prprj` 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 verification 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 results. 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 use 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'`

Depth of verification specified by the model reference level to which you want to analyze.

Only for Embedded Coder

Example: `opt.ModelRefVerifDepth = '3';`

`ModelRefByModelRefVerif` — Model reference analysis mode

`false` (default) | `true`

Model reference analysis mode specified as `false` to verify reference models within the model hierarchy, or `true` to verify referenced models individually.

Only for Embedded Coder

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 all options from project configuration and run complete analysis.
- `'PrjConfigAndMisraCxx'` – Inherit all options from project configuration, enable MISRA C++ rule checking, and run complete analysis.
- `'PrjConfigAndJSF'` – Inherit all 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`, to use Lookup Table code during the analysis, or `false`, to not.

Only for TargetLink

Example: `opts.AutoStubLUT = true;`

See Also

`pslinkoptions` | `pslinkrun`

8

MISRA C 2012

MISRA C:2012 Directive 4.1

Run-time failures shall be minimized

Description

Rule 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 Specification

This directive is checked through the Polyspace analysis.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

Run-time failures shall be minimized.

Check Information

Group: Code Design

Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Directive 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

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Directive 4.3

Assembly language shall be encapsulated and isolated

Description

Rule 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 Specification

Polyspace does not raise a warning on assembly language code encapsulated in `asm` functions or in `asm` pragmas.

Message in Report

Assembly language shall be encapsulated and isolated

Check Information

Group: Code Design
Category: Required

AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 1.2

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Directive 4.6

`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 Specification

Polyspace does not issue a warning for the `typedef` definition.

Message in Report

typedefs that indicate size and signedness should be used in place of the basic numerical types

Check Information

Group: Code Design
Category: Advisory
AGC Category: Advisory
Language: C90, C99

More About

- “Set Up Coding Rules Checking”

- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Directive 4.9

A function should be used in preference to a function-like macro where they are interchangeable

Description

Rule 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 Specification

Polyspace raises a warning on all function-like macro definitions.

Message in Report

A function should be used in preference to a function-like macro where they are interchangeable

Check Information

Group: Code Design
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 13.2 | MISRA C:2012 Rule 20.7

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Directive 4.10

Precautions shall be taken in order to prevent the contents of a header file being included more than once

Description

Rule 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. This situation can be a source of confusion. If this multiple inclusion leads to multiple or conflicting definitions, then your program can have undefined or erroneous behavior.

Polyspace Specification

Try to prevent multiple inclusions when a header file is formatted as:

```
#ifndef <control macro>
#define <control macro>
    contents
#endif
```

or

```
#ifdef <control macro>
#error ...
#else
#define <control macro>
    contents
#endif
```

Otherwise, Polyspace flags the inclusion as non-compliant.

Message in Report

Precautions shall be taken in order to prevent the contents of a header file being included more than once.

Check Information

Group: Code Design
Category: Required
AGC Category: Required
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Directive 4.11

The validity of values passed to library functions shall be checked

Description

Rule 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 Specification

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`

Message in Report

The validity of values passed to library functions shall be checked

Check Information

Group: Code Design
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Directive 4.1

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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 Specification

Standard compilation error messages do not lead to a violation of this MISRA rule.

Message in Report

- Too many nesting levels of #includes: N1. The limit is N0.
- Integer constant is too large.
- ANSI C does not allow '#XX'.
- Text following preprocessing directive violates ANSI standard.
- Too many macro definitions: N1. The limit is N0.
- Array of zero size should not be used.
- Integer constant does not fit within long int.
- Integer constant does not fit within unsigned long int.
- Too many nesting levels for control flow: N1. The limit is N0.
- Assembly language should not be used.
- Too many enumeration constants: N1. The limit is N0.

Check Information

Group: Standard C Environment

Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 1.2

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

All the supported extensions lead to a violation of this MISRA rule.

Message in Report

- ANSI C90 forbids hexadecimal floating-point constants.
- ANSI C90 forbids universal character names.
- ANSI C90 forbids mixed declarations and code.
- ANSI C90/C99 forbids case ranges.
- ANSI C90/C99 forbids local label declaration.
- ANSI C90 forbids mixed declarations and code.
- ANSI C90/C99 forbids typeof operator.
- ANSI C90/C99 forbids casts to union.
- ANSI C90 forbids compound literals.
- ANSI C90/C99 forbids statements and declarations in expressions.
- ANSI C90 forbids __func__ predefined identifier.

- ANSI C90 forbids keyword '_Bool'.
- ANSI C90 forbids 'long long int' type.
- ANSI C90 forbids long long integer constants.
- ANSI C90 forbids 'long double' type.
- ANSI C90/C99 forbids 'short long int' type.
- ANSI C90 forbids _Pragma preprocessing operator.
- ANSI C90 does not allow macros with variable arguments list.
- ANSI C90 forbids designated initializer.

  Keyword 'inline' should not be used.

Check Information

Group: Standard C Environment
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 1.1

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

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.

Check Information

Group: Standard C Environment
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Directive 4.1

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

A project shall not contain unreachable code.

Check Information

Group: Unused Code
Category: Required

AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 14.3 | MISRA C:2012 Rule 16.4

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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

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.

Polyspace Specification

Polyspace checks for useless writes during the Polyspace Code Prover verification.

Message in Report

There shall be no dead code.

Check Information

Group: Unused Code
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 17.7

More About

- “Set Up Coding Rules Checking”

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 2.3

A project should not contain unused type declarations

Description

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.

Message in Report

A project should not contain unused type declarations: type XX is not used.

Examples

Unused Local Type

```
int16_t unusedtype (void){

    typedef int16_t local_Type;

    return 67;
}
```

In this function, `typedef` defines `local_Type` as a new local type, but this type is never used in the function.

Correction — Use `local_Type`

One possible correction is to use `local_Type` as part of the function.

```
int16_t unusedtype (void){

    typedef int16_t local_Type;

    local_Type temp_var = 67;

    return temp_var;
}
```

Check Information

Group: Unused Code
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

MISRA C:2012 Rule 2.4

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 2.4

A project should not contain unused tag declarations

Description

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.

Message in Report

A project should not contain unused tag declarations: tag *`tag_name`* is not used.

Examples

Unused `struct` Tag

```
typedef struct record_t
{
    uint16_t key;
    uint16_t val;
} record1_t;
```

In this example, the tag `record_t` is used only in the `typedef` of `record1_t`, which is used in the rest of the translation unit whenever the type is needed.

Correction — Define `struct` Without a Tag

This typedef can be written in a compliant manner by omitting the tag.

```
typedef struct
```

```
{
    uint16_t key;
    uint16_t val;
} record1_t;
```

Check Information

Group: Unused Code
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

MISRA C:2012 Rule 2.3

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 2.5

A project should not contain unused macro declarations

Description

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.

Message in Report

A project should not contain unused macro declarations: macro *macro_name* is not used.

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 within this function.

Correction — Use DATA in a Function Call

One possible correction is to use DATA as part of a function call.

```
void use_macro (void)
{
    #define SIZE 4
    #define DATA 3

    use_int16(SIZE);
    use_int32(DATA);
}
```

Check Information

Group: Unused Code
Category: Advisory
AGC Category: Readability
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

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 Specification

You cannot annotate this rule in the source code.

For information on annotations, see "Add Review Comments to Code".

Message in Report

The character sequence `/*` shall not appear within a comment.

Check Information

Group: Comments
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

Line-splicing shall not be used in // comments.

Check Information

Group: Comments
Category: Required
AGC Category: Required
Language: C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Software Quality Objective Subsets (C:2012)"
- "Polyspace MISRA C:2012 Checker"

- “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.

Message in Report

Octal and hexadecimal escape sequences shall be terminated.

Check Information

Group: Character Sets and Lexical Conventions
Category: Required
AGC Category: Required
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"

- "Polyspace MISRA C:2012 Checker"
- "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 Specification

The Polyspace analysis converts trigraphs to the equivalent character for the run-time verification. However, Polyspace also raises a MISRA violation.

Message in Report

Trigraphs should not be used.

Check Information

Group: Character Sets and Lexical Conventions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

MISRA C:2012 Rule 5.1

External identifiers shall be distinct

Description

Rule Definition

External identifiers shall be distinct.

Rationale

External identifiers are ones declared with global scope or storage class `extern`.

Polyspace considers two names as distinct if there is a difference between their first 31 characters. If the difference between two names occurs only beyond the first 31 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 6 characters. To use the C90 rules checking, use the command-line option `-no-language-extensions`.

Message in Report

External `%s` `%s` conflicts with the external identifier XX in file YY.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 5.2 | MISRA C:2012 Rule 5.4 | MISRA C:2012 Rule 5.5

More About

- "Set Up Coding Rules Checking"

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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

Message in Report: Identifier XX has same significant characters as identifier YY.

Rule Definition

Identifiers declared in the same scope and name space shall be distinct.

Rationale

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the command-line option `-no-language-extensions`.

Message in Report

Identifier XX has same significant characters as identifier YY.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

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

More About

- “Set Up Coding Rules Checking”

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the command-line option `-no-language-extensions`.

Message in Report

Variable XX hides variable XX (FILE line LINE column COLUMN).

Check Information

Group: Identifiers
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 5.2 | MISRA C:2012 Rule 5.8

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the command-line option `-no-language-extensions`.

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.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 5.1 | MISRA C:2012 Rule 5.2 | MISRA C:2012 Rule 5.5

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the command-line option `-no-language-extensions`.

Message in Report

Identifier XX has same significant characters as macro YY.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 5.1 | MISRA C:2012 Rule 5.2 | MISRA C:2012 Rule 5.4

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”

- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 5.6

A typedef name shall be a unique identifier

Description

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.

Message in Report

XX conflicts with the typedef name YY.

Check Information

Group: Identifiers
Category:
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 5.7

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

XX conflicts with the tag name YY.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 5.6

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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

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.

Message in Report

- Object XX conflicts with the object name YY.
- Function XX conflicts with the function name YY.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 5.3

More About

- "Set Up Coding Rules Checking"

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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

Rule Definition

Identifiers that define objects or functions with internal linkage should be unique.

Polyspace Specification

This rule checker assumes that rule 5.8 is not violated.

Message in Report

- Object XX conflicts with the object name YY.
- Function XX conflicts with the function name YY.

Check Information

Group: Identifiers
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

MISRA C:2012 Rule 8.10

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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` 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.

Message in Report

Bit-fields shall only be declared with an appropriate type.

Check Information

Group: Types
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”

- “Polyspace MISRA C:2012 Checker”
- “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 Specification

This rule does not apply to unnamed bit fields because their values cannot be accessed.

Message in Report

Single-bit named bit fields shall not be of a signed type.

Check Information

Group: Types
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

Octal constants shall not be used.

Examples

Use of octal constants

```
#define CST     021
#define VALUE   010              /* Compliant - constant not used */
#if 010 == 01                    /* Non-Compliant - constant used */
#define CST 021                  /* 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) {          /* Non-Compliant - decimal 17 */
        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
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Readability
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

The lowercase character “l” shall not be used in a literal suffix.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Readability
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Examples

Incorrect Assignment of String Literal

```
char *str1 = "AccountHolderName";
const char *str2 = "AccountHolderName";

void checkAccount1(char*);              /* Non-Compliant */
void checkAccount2(const char*);        /* Compliant */

void main() {
 checkAccount1("AccountHolderName");    /* Non-Compliant */
 checkAccount2("AccountHolderName");    /* 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
Language: C90, C99

See Also

MISRA C:2012 Rule 11.4 | MISRA C:2012 Rule 11.8

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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

The C90 standard permits types to be omitted in some circumstances, in which case the `int` type is implicitly specified. Examples of potential circumstances in which you can use an implicit `int` are:

- Object declarations
- Parameter declarations
- Member declarations
- `typedef` declarations
- Function return types

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 `const char` might have been expected.

Message in Report

Types shall be explicitly specified.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required

Language: C90

See Also

MISRA C:2012 Rule 8.2

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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 mismatch between the number of arguments and parameters, their types, and the expected and actual return type of a function provides potential for undefined behavior. This rule also requires that you specify names 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.

Polyspace Specification

Polyspace also checks the function definition.

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.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 8.1 | MISRA C:2012 Rule 8.4 | MISRA C:2012 Rule 17.3

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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

Rule Definition

All declarations of an object or function shall use the same names and type qualifiers.

Rationale

Consistently using types and qualifiers across declarations of the same object or function encourages stronger typing. By specifying parameter names in function prototypes, Polyspace can check for interface consistency between the function definition and declarations.

Polyspace Specification

Polyspace generates some violations of this rule during the link phase.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

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.

Check Information

Group: Declarations and Definitions

Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 8.4

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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 for an object or function is visible when the object or function is defined, a compiler must check that the declaration and definition are compatible. In the presence of function prototypes, as required by rule 8.2, checking extends to the number and type of function parameters. A better way of implementing declarations of objects and functions with external linkage is to declare them in a header file. Then include the header file in all those code files that require them, including the one that defines them.

Message in Report

- Global definition of *`variable_name`* variable has no previous declaration.
- Function *`function_name`* has no visible compatible prototype at definition.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Advisory
Language: C90, C99

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

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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

Rule Definition

An external object or function shall be declared once in one and only one file.

Rationale

Typically, a single declaration is made in a header file that you include any in translation unit in which the identifier is defined or used. This inclusion ensures consistency between:

- The declaration and the definition
- The declarations in different translation units

Note: It is possible to have many header files in a project, but each external object or function is declared in only one header file.

Polyspace Specification

Polyspace checks only explicit `extern` declarations (tentative definitions are ignored).

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

- Object *`object_name`* has external declarations in multiples files.
- Function *`function_name`* has external declarations in multiples files.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 8.4

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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

Rule Definition

An identifier with external linkage shall have exactly one external definition.

Rationale

The behavior is undefined if you use an identifier for which multiple definitions exist (in different files) or no definition exists. Multiple definitions in different files are not permitted by this rule even if the definitions are the same. If the declarations are different, or initialize the identifier to different values, it is undefined behavior.

Polyspace Specification

Polyspace considers tentative definitions as definitions, but does not raise warnings on predefined symbols.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

- Forbidden multiple definitions for function *`function_name`*.
- Forbidden multiple tentative of definition for object *`object_name`*.
- Global variable *`variable_name`* multiply defined.
- Function *`function_name`* multiply defined.
- Global variable has multiple tentative of definitions.
- Undefined global variable *`variable_name`*.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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

Rule Definition

Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.

Rationale

Restricting or reducing the visibility of an object by giving it internal linkage or no linkage reduces the chance that it is accessed inadvertently. Compliance with this rule also avoids any possibility of confusion between your identifier and an identical identifier in another translation unit or a library.

Polyspace Specification

If your program does not use the externally defined function or object, Polyspace does not raise a warning.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

- Variable *`variable_name`* should have internal linkage.
- Function *`function_name`* should have internal linkage.

Check Information

Group: Declarations and Definitions
Category: Advisory

AGC Category: Advisory
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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 have an object or function declared with `extern`, and another declaration of the object or function is already visible, the linkage can be confusing. You expect that the `extern` storage class specifier creates external linkage. Apply the `static` storage class specifier to objects and functions with internal linking.

Message in Report

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

Examples

Internal and External Linkage Conflicts

```
static int foo = 0;
extern int foo;         /* Non-compliant */

extern int hhh;
static int hhh;         /* Non-compliant */
```

In this example, the first line defines `x` with internal linkage. Because the example uses the `static` keyword, the first line is compliant. However, the second line does not use `static` in the declaration, so the declaration is noncompliant. By comparison, the third line declares `y` with an `extern` keyword creating external linkage. The fourth line declares `y` with internal linkage, but this declaration conflicts with the first declaration of `y`.

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;
```

Internal linkage

```
static int fee(void);  /* Compliant - declaration: internal linkage */
int fee(void){         /* Non-compliant */
  return 1;
}

static int ggg(void);  /* Compliant - declaration: internal linkage */
extern int ggg(void){  /* 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 to be compliant with MISRA

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"

- “Polyspace MISRA C:2012 Checker”
- “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

Defining an object at block scope reduces the possibility that you inadvertently access the object . It ensures your program does not access the object elsewhere.

Polyspace Specification

Polyspace raises a warning only for static objects.

Message in Report

An object should be defined at block scope if its identifier only appears in a single function.

Check Information

Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

More About

- "Set Up Coding Rules Checking"

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 with external linkage, you can call the external definition of the function or the inline definition. This behavior can affect the execution time and therefore impact your program.

Tip To make an inline function available to several translation units, place its definition in a header file.

Message in Report

An inline function shall be declared with the static storage class.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C99

See Also

MISRA C:2012 Rule 5.9

More About

- “Set Up Coding Rules Checking”

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 incomplete type and access its elements, it is safer to state the size of the array explicitly. Providing size information for each declaration allows the software to check the declarations for consistency. It also allows a static checker to perform array bounds analysis without analyzing more than one unit.

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.

Examples

Array Declarations

```
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
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 1 greater than its predecessor. If the first enumeration constant is implicitly specified, then its value is 0. An explicitly specified enumeration constant has the value of the associate constant expression.

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.

Message in Report

The constant `constant1` has same value as the constant `constant2`.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”

- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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 Specification

Polyspace issues a warning if a non-`const` pointer parameter either:

- Does not modify the addressed object.
- Is passed to a call of a function that is declared with a `const` pointer parameter.

Message in Report

A pointer should point to a const-qualified type whenever possible.

Examples

Pointer Parameters

```
#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. Because s does not modify an object, this parameter is noncompliant. 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
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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, make sure that the memory areas operated on by two or more pointers do not overlap.

Message in Report

The restrict type qualifier shall not be used.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Advisory
Language: C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

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.

Message in Report

The value of an object with automatic storage duration shall not be read before it has been set.

Check Information

Group: Initialization
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

MISRA C:2012 Rule 15.1 | MISRA C:2012 Rule 15.3

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

The initializer for an aggregate or union shall be enclosed in braces.

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
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

Arrays shall not be partially initialized.

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
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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 an 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.

Message in Report

An element of an object shall not be initialized more than once.

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
Language: C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

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
Language: C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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`
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

Operation		Essential type category of arithmetic operand					
+ (unary)		3	4	5			
- (unary)		3	4	5		8	
+ -	either	3		5			
* /	either	3	4	5			
%	either	3	4	5			1
< > <= >=	either	3					
== !=	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.

Message in Report

The *`operand_name`* operand of the *`operator_name`* operator is of an inappropriate essential type category *`category_name`*.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 10.2

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

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 character data arithmetically because the data does not represent numeric values.

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.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 10.1

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

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.

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.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 10.4 | MISRA C:2012 Rule 10.5 | MISRA C:2012 Rule 10.6

More About

- "Set Up Coding Rules Checking"

- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

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.

Message in Report

Operands of `operator_name` operator shall have the same essential type category.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 10.3 | MISRA C:2012 Rule 10.7

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"

- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

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.

Message in Report

The value of an expression should not be cast to an inappropriate essential type.

Check Information

Group: The Essential Type Model
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 10.3 | MISRA C:2012 Rule 10.8

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

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.

Message in Report

The composite expression is assigned to an object with a wider essential type.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory

Language: C90, C99

See Also

MISRA C:2012 Rule 10.3 | MISRA C:2012 Rule 10.7

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

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.

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.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

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.

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.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 10.5

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

Polyspace considers both explicit and implicit casts when checking this rule. However, casts from `NULL` or `(void*)0` do not violate this rule.

Message in Report

Conversions shall not be performed between a pointer to a function and any other type.

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
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

Conversions shall not be performed between a pointer to an incomplete type and any other type.

Examples

Casts from incomplete type

```
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
Language: C90, C99

See Also

MISRA C:2012 Rule 11.5

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

A cast shall not be performed between a pointer to object type and a pointer to a different object type.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 11.4 | MISRA C:2012 Rule 11.5 | MISRA C:2012 Rule 11.8

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”

- “Polyspace MISRA C:2012 Checker”
- “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 Specification

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A conversion should not be performed between a pointer to object and an integer type.

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 “Files and folders to ignore (C)” or “Files and folders to ignore (C++)”.
- 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
Language: C90, C99

See Also

MISRA C:2012 Rule 11.3 | MISRA C:2012 Rule 11.7 | MISRA C:2012 Rule 11.9

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A conversion should not be performed from pointer to void into pointer to object.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 11.2 | MISRA C:2012 Rule 11.3

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A cast shall not be performed between pointer to void and an arithmetic type.

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
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

A cast shall not be performed between pointer to object and a non-integer arithmetic type.

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.

The rule is not violated when the pointer p is cast to long*.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 11.4

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

Polyspace flags both implicit and explicit conversions that violate this rule.

Message in Report

A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.

Check Information

Group: Pointer Type Conversions

Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 11.3

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 require 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.

Message in Report

The macro NULL shall be the only permitted form of integer null pointer constant.

Examples

Using 0 for Pointer Assignments and Comparisons

```
void main(void) {

    int *p1 = 0;               /* Non-compliant */
    int *p2 = ( void * ) 0;    /* Compliant     */

#define MY_NULL_1 0
#define MY_NULL_2 ( void * ) 0
```

```
    if ( p1 == MY_NULL_1 )    /* Non-compliant */
    { }
    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.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Readability
Language: C90, C99

See Also

MISRA C:2012 Rule 11.4

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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	++(post-increment) - -(post-decrement) & * + - ~ ! sizeof 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

Description	Operator and Operand	Precedence
Logical AND	&&	4
Logical OR	||	3
Conditional	?:	2
Assignment	= *= /= += -= <<= >>= &= ^= |=	1
Comma	,	0

Message in Report

Operand of logical %s is not a primary expression. The precedence of operators within expressions should be made explicit.

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

# if ! defined X && defined Y  /* Non-compliant - MISRA-12.1 */
# endif
```

In this example, two violations of MISRA rule 12.1 are shown in preprocessing code. In each 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

# if ! defined (X) && defined (Y)
# endif
```

Compliant Expressions Without Parentheses

```
int a, b, c, x;
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
Language: C90, C99

See Also

MISRA C:2012 Rule 12.2 | MISRA C:2012 Rule 12.3 | MISRA C:2012 Rule 12.4

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 the following 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 Specification

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.

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.

Check Information

Group: Expressions

Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 12.1

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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

Use of the comma operator is generally detrimental to the readability of code. The same code can usually be written in another form.

Message in Report

The comma operator should not be used.

Examples

Comma Usage in C Code

```
typedef signed int abc, xyz, jkl;

static void func1 ( abc, xyz, jkl );       /* Compliant */

int foo(void)
{
    volatile int rd = 1;                   /* Compliant */
    int var=0, foo=0, k=0, n=2, p, t[10];  /* Compliant */

    int abc = 0, xyz = abc + 1;            /* Compliant */
    int jkl = ( abc + xyz, abc + xyz );    /* Not compliant */

    var = 1, foo += var, kkk = 3;          /* Not compliant */
    var = (kkk = 1, foo = 2);             /* Not compliant */

    for ( var = 0, ptr = &t[ 0 ]; var < num; ++var, ++ptr){}
                                          /* Not compliant */

    if ((abc,xyz)<0) { return 1; }        /* Not compliant */
}
```

In this example, the code shows various uses of commas in C code. Using commas to call functions with variables is allowed (line 3). When using the comma for initialization, the variables and values must be clear (line 8 and 10). Line 11 is not compliant because it is unclear what `jkl` is initialized to. (For example, `abc+xyz`, `(abc+xyz)*(abc+xyz)`, `f((abc+xyz),(abc+xyz))`, etc.)

Line 13 and 14 are both assignment statements, but it is unclear which variables are getting assigned which values.

Line 16 violates multiple MISRA coding rules because the complex `for` statement makes it unclear which values control the loop.

Line 18 violates rule 12.3 because it is unclear if the `if` statement depends on `abc`, `xyz`, or both.

Check Information

Group: Expressions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 12.1

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Check Information

Group: Expressions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 12.1

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”

- "Software Quality Objective Subsets (C:2012)"

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.

Message in Report

Initializer lists shall not contain persistent side effects.

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
Language: C99

See Also

MISRA C:2012 Rule 13.2

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, this rule forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation.

Polyspace Specification

Rule 13.2 assumes that the comma operator is not used (rule 12.3).

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.

Examples

Variable Modified More Than Once in Expression

```
int a[10], b[10];
```

```
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++);       /* Non-compliant  */
}
```

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)`.

Check Information

Group: Side Effects
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Directive 4.9 | MISRA C:2012 Rule 13.1 | MISRA C:2012 Rule 13.3 | MISRA C:2012 Rule 13.4

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”

- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

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.

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++);
    if (choice == 0) {
        res = x++ + y++;
        return(res);           /* Non-compliant */
    }
    else if (choice == 1) {
        x++;                   /* Compliant */
        y++;                   /* Compliant */
        return (x+y);
    }
    else {
        res = operation(x++,y);
        return(res);           /* Non-compliant */
    }
}
```

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
Language: C90, C99

See Also

MISRA C:2012 Rule 13.2

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”

- “Polyspace MISRA C:2012 Checker”
- “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`.

Message in Report

The result of an assignment operator should not be used.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 13.2

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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 Specification

- For this rule, Polyspace considers that all function calls have a persistent side effect.
- If the right operand is a volatile variable, Polyspace does not flag this as a rule violation.

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.

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();

    if(mySwitch && check(val)) {   /* Non-compliant */
        }
}
```

In this example, the rule is violated because the right operand of the && operation is 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.

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
Language: C90, C99

More About

- "Set Up Coding Rules Checking"

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

The rule is not violated if the argument is a `volatile` variable.

Message in Report

The operand of the sizeof operator shall not contain any expression which has potential side effects.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 18.8

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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 Specification

If the `for` index is a variable symbol, Polyspace checks that it is not a float.

Message in Report

A loop counter shall not have essentially floating type.

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 = 1000 at the end of the loop */
        ++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){
        foo += 0.001f;  /* Non-compliant - foo used as a loop counter */
    }

    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
Language: C90, C99

See Also

MISRA C:2012 Rule 14.2

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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` statement provides a general-purpose looping facility. Using a restricted form of loop makes code easier to review and to analyze.

Polyspace Specification

Polyspace checks that:

- The `for` loop index (`V`) is a variable symbol.
- `V` is the last assigned variable in the first expression (if present).
- If the first expression exists, it contains an assignment of `V`.
- If the second expression exists, it is a comparison of `V`.
- If the third expression exists, it is an assignment of `V`.
- There are no direct assignments of the `for` loop index.

Message in Report

- 1st expression should be an assignment. The following kinds of for loops are allowed:
  - all three expressions shall be present;
  - the 2nd and 3rd expressions shall be present with prior initialization of the loop counter;
  - all three expressions shall be empty for a deliberate infinite loop.
- 3rd expression should be an assignment of a loop counter.

- 3rd expression : assigned variable should be the loop counter (*counter*).
- 3rd expression should be an assignment of loop counter (*counter*) only.
- 2nd expression should contain a comparison with loop counter (*counter*).
- Loop counter (*counter*) should not be modified in the body of the loop.
- Bad type for loop counter (*counter*).

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; i++) {} /* 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
Language: C90, C99

See Also

MISRA C:2012 Rule 14.1 | MISRA C:2012 Rule 14.3 | MISRA C:2012 Rule 14.4

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"

- “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 Specification

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.

Message in Report

- Boolean operations whose results are invariant shall not be permitted.
- Expression is always true.
- Boolean operations whose results are invariant shall not be permitted.
- Expression is always false.
- Controlling expressions shall not be invariant.

Check Information

Group: Control Statement Expressions

Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 2.1 | MISRA C:2012 Rule 14.2

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

Polyspace does not flag integer constants, for example `if(2)`.

If your configuration includes the option `-boolean-types`, the number of warnings can increase or decrease.

Message in Report

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 14.2 | MISRA C:2012 Rule 20.8

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

The goto statement should not be used.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 15.2 | MISRA C:2012 Rule 15.3 | MISRA C:2012 Rule 15.4

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

The goto statement shall jump to a label declared later in the same function.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 15.1 | MISRA C:2012 Rule 15.3 | MISRA C:2012 Rule 15.4

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.

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
Language: C90, C99

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

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

There should be no more than one break or goto statement used to terminate any iteration statement.

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++) {   /* Non-compliant  */
        if(sum >= sat)
            break;
        if(stop)
            goto L1;
        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++) {  /* Non-compliant */
        if(sum >= sat)
            break;
        for (j=0; j< i; j++) { /* Compliant */
            if(stop)
                goto L1;
            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
Language: C90, C99

See Also

MISRA C:2012 Rule 15.1 | MISRA C:2012 Rule 15.2 | MISRA C:2012 Rule 15.3

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

A function should have a single point of exit at the end.

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

Language: C90, C99

See Also

MISRA C:2012 Rule 17.4

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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

The rule applies to:

- Iteration statements such as `while`, `do ... while` or `for`.
- Selection statements such as `if ... else` or `switch`.

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.

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.

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

```
void f1(void) {
    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.

```
void f1(void) {
    if(flag_1) {                          /* Compliant */
        if(flag_2) {                        /* Compliant */
            action_1();
        }
    }
    else {                                /* Compliant */
        action_2();
    }
}
```

Spurious Semicolon After Iteration Statement

```
void f1(void) {
    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
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

All if … else if constructs shall be terminated with an else statement.

Examples

Missing `else` Block

```
int get_flag_1(void);
int get_flag_2(void);
void action_1(void);
void action_2(void);

void f1(void) {
    int flag_1 = get_flag_1(), flag_2 = get_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. The block can be empty.

```
int get_flag_1(void);
int get_flag_2(void);
void action_1(void);
void action_2(void);

void f1(void) {
    int flag_1 = get_flag_1(), flag_2 = get_flag_2();
    if(flag_1) {
        action_1();
    }
    else if(flag_2) {
        /* Non-compliant */
        action_2();
    }
    else {
        /* No statement required */
        /* ; is optional */
    }

}
```

Check Information

Group: Control Flow
Category: Required
AGC Category: Readability
Language: C90, C99

See Also

MISRA C:2012 Rule 16.5

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

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.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory
Language: C90, C99

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

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

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.

Check Information

Group: Switch Statements
Category: Required

AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 16.1

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

Polyspace raises a warning for each noncompliant `case` clause.

Message in Report

An unconditional break statement shall terminate every switch-clause.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 16.1

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

Every switch statement shall have a default label.

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){

    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
Language: C90, C99

See Also

MISRA C:2012 Rule 2.1 | MISRA C:2012 Rule 16.1

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

A default label shall appear as either the first or the last switch label of a switch statement.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 15.7 | MISRA C:2012 Rule 16.1

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

Every switch statement shall have at least two switch-clauses.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 16.1

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

If your configuration uses the `-boolean-types` option, the number of reported violations can increase.

Message in Report

A switch-expression shall not have essentially Boolean type.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory
Language: C90, C99

More About

- "Set Up Coding Rules Checking"

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

MISRA C:2012 Rule 17.1

The features of <starg.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`.

Message in Report

The features of <stdarg.h> shall not be used.

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
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

Message in Report: Function XX shall not call itself either directly or indirectly. Function XX is called indirectly by YY.

Examples

Direct and Indirect Recursion

```
void foo1( void ) {     /* Non-compliant - Indirect recursion foo1->foo2->foo1... */
    foo2();
    foo1();               /* Non-compliant - Direct recursion */
}

void foo2( void ) {
    foo1();
}
```

In this example, the rule is violated because of:

- Direct recursion `foo1` → `foo1`.
- Indirect recursion `foo1` → `foo2` → `foo1`.

Check Information

Group: Function
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

Function 'XX' has no complete visible prototype at call.

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 examples, 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
Language: C90

See Also

MISRA C:2012 Rule 8.2 | MISRA C:2012 Rule 8.4

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”

- “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.

Message in Report

Missing return value for non-void function 'XX'.

Examples

Missing Return Statement Along Certain Execution Paths

```
int absolute(int v) {
    if(v < 0) {
        return v;
    }
}
```

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;
    }
    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
Language: C90, C99

See Also

MISRA C:2012 Rule 15.5

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

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.

Message in Report

The declaration of an array parameter shall not contain the static keyword between the [ ].

Examples

Use of `static` Keyword Within `[ ]` in Array Parameter

```
extern int arr1[20];
extern int arr2[10];

/* Non-compliant: static keyword used in array declarator */
unsigned int total (unsigned int n, unsigned int arr[static 20]) {
    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);  /* Non-compliant - behavior not defined */
    res2 = total (20U, arr2); /* Non-compliant, even if behavior is defined */
}
```

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
Language: C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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`.

Message in Report

The value returned by a function having non-void return type shall be used.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 2.2

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

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 Specification

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`

Message in Report

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Directive 4.1 | MISRA C:2012 Rule 18.4

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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`. If `pointer_expression1` and `pointer_expression2` do not point to elements of the same array or the element beyond the end of that array, it is undefined behavior.

Message in Report

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Directive 4.1 | MISRA C:2012 Rule 18.4

More About

- “Set Up Coding Rules Checking”

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object.

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
Language: C90, C99

See Also

MISRA C:2012 Directive 4.1

More About

- “Set Up Coding Rules Checking”

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

Polyspace flags operations on pointers, for example, `Pointer + Integer`, `Integer + Pointer`, `Pointer - Integer`.

Message in Report

The +, -, += and -= operators should not be applied to an expression of pointer type.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 18.1 | MISRA C:2012 Rule 18.2

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

Declarations should contain no more than two levels of pointer nesting.

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
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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 Specification

Polyspace flags a violation when assigning an address to a global variable, returning a local variable address, or returning a parameter address.

Message in Report

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.

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 g(unsigned short *p){
    sp = p;  /* Non-compliant
              * the parameter u from f is copied to static sp */
}

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 `g` stores a copy of its pointer parameter `p`. If `p` always points to an object with static storage duration, then the code is compliant with this rule. However, in this example , `p` points to an object with automatic storage duration. In such a case, copying the parameter `p` is noncompliant.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

Flexible array members shall not be declared.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 21.3

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

Variable-length array types shall not be used.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required
Language: C99

See Also

MISRA C:2012 Rule 13.6

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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`.

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.

Examples

Assignment of Unions

```
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
Language: C90, C99

See Also

MISRA C:2012 Rule 19.2

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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-dependant.

Message in Report

The union keyword should not be used.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 19.1

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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 Specification

Polyspace flags text that precedes a `#include` directive. Polyspace ignores preprocessor directives, comments, spaces, or "new lines".

Message in Report

#include directives should only be preceded by preprocessor directives or comments.

Check Information

Group: Preprocessing Directives
Category: Advisory
AGC Category: Advisory
Language: C90, C99

More About

- “Set Up Coding Rules Checking”

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 `' " \ /* //` are used between `< >` delimiters in a header name preprocessing token.
- You use `' \ /* //` are used between `"` delimiters in a header name preprocessing token.

Although `\` results in undefined behavior, many implementations accept `/` in its place.

Polyspace Specification

Polyspace flags the characters `' \ " /*` between < and > in `#include <filename>`.

Polyspace flags the characters `' \ /*` between " and " in `#include <filename>`.

Message in Report

The ', "or \ characters and the /* or // character sequences shall not occur in a header file name.

Check Information

Group: Preprocessing Directives

Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 doe snot use one of the following forms:

- `#include <filename>`
- `#include "filename"`

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.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

More About

- "Set Up Coding Rules Checking"

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

- The macro *`macro_name`* shall not be redefined.
- The macro *`macro_name`* shall not be undefined.

Examples

Redefining `int` keyword

```
#define int some_other_type
          /* Non-compliant - int keyword behavior altered */
#include <stdlib.h>
...
```

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:

```
#define int_mine some_other_type
#include <stdlib.h>
...
```

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
```

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
Languages: C90, C99

See Also

`MISRA C:2012 Rule 21.1`

More About

- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

#undef shall not be used.

Check Information

Group: Preprocessing Directives
Category: Advisory
AGC Category: Readability
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

Polyspace looks for the # character in a macro arguments (outside a string or character constant).

Message in Report

Macro argument shall not look like a preprocessing directive.

Examples

Macro Expansion Causing 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
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

Expanded macro parameter *`param`* shall be enclosed in parentheses.

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
Language: C90, C99

See Also

MISRA C:2012 Directive 4.9

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 14.4

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

`Identifier` is not defined.

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
Language: C90, C99

See Also

MISRA C:2012 Directive 4.9

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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.

Message in Report

The # and ## preprocessor operators should not be used.

Check Information

Group: Preprocessing Directives
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 1.3 | MISRA C:2012 Rule 20.11

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”

- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

The ## preprocessor operator shall not follow a macro parameter following a # preprocessor operator.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 20.10

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

Message in Report

Expanded macro parameter *`param1`* is also an operand of *`op`* operator.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "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 can use a preprocessing directive to conditionally exclude source code until it encounters a corresponding `#else`, `#elif`, `#endif` directive. If your compiler does not detect a malformed or invalid preprocessing directive inside excluded source code, more code than you intended to excluded.

If all preprocessing directives are syntactically valid, even in excluded code, this unintended code exclusion cannot happen.

Message in Report

Directive is not syntactically meaningful.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”

- “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

Message in Report

- '#else' not within a conditional.
- '#elsif' not within a conditional.
- '#endif' not within a conditional. unterminated conditional directive.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

More About

- "Set Up Coding Rules Checking"

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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.

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.

Examples

Defining or Undefining Reserved Identifiers

```
#undef __LINE__               /* Non-compliant - begins with _ */
#define _Guard_H 1            /* Non-compliant - begins with _ */
#undef _ BUILTIN_squrt        /* 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
Languages: C90, C99

See Also

MISRA C:2012 Rule 20.4

More About

- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.2

A reserved identifier or macro name shall not be declared

Description

Rule Definition

A reserved identifier or 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 Specification

- 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.

Polyspace Specification

Message in Report

Identifier 'XX' shall not be reused.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

If you use names of dynamic heap memory allocation functions for macros, and you expand the macros in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro <name> shall not be used.
- Identifier XX should not be used.

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
Language: C90, C99

See Also

MISRA C:2012 Rule 18.7

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”

- “Polyspace MISRA C:2012 Checker”
- “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 Specification

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.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"

- “Polyspace MISRA C:2012 Checker”
- “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 Specification

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.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”

- “Polyspace MISRA C:2012 Checker”
- “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 Specification

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.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.7

The `atof`, `atoi`, `atol`, and `atoll` functions of `<stdlib.h>` shall not be used

Description

Rule Definition

The 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 Specification

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.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”

- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.8

The library functions abort, exit, getenv and system of <stdlib.h> shall not be used

Description

Rule Definition

The library functions abort, exit, getenv and system of <stdlib.h> shall not be used.

Rationale

Using these functions can cause undefined and implementation-defined behaviors.

Polyspace Specification

In case the abort, exit, getenv, and system functions are actually macros, and the macros are expanded in the code, this rule is detected as violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”

- “Polyspace MISRA C:2012 Checker”
- “Software Quality Objective Subsets (C:2012)”

Introduced in R2014b

MISRA C:2012 Rule 21.9

The 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 Specification

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.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required
Language: C90, C99

More About

- "Set Up Coding Rules Checking"

- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”
- “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 Specification

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.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required
Language: C90, C99

More About

- “Set Up Coding Rules Checking”
- “Review Coding Rule Violations”

- “Polyspace MISRA C:2012 Checker”
- “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 Specification

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.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

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
Language: C90, C99

More About

- "Set Up Coding Rules Checking"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

9

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.

Multi-line comments are counted as one comment. A statement typically ends with a semi-colon with some exceptions. Exceptions include semi-colons in `for` loops or structure field declarations.

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 “Review Code 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

Category: File
Acronym: `COMF`

Cyclomatic Complexity

Number of linearly independent paths through source code

Description

This metric specifies the number of linearly independent paths through the source code.

To calculate this metric, add 1 to the number of decision points in your code. A decision point is a statement that causes your program to branch into two paths. For example, at an `if` statement, your program can either enter the `if` branch or not.

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 “Review Code 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;
    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

Category: Function
Acronym: `VG`

Language Scope

Language scope

Description

This metric measures the cost of maintaining or changing a function. It is calculated as:

`(N1 + N2)/(n1 + n2)`
Here:

- `N1` is the number of occurrences of operators.
- `N2` is the number of occurrences of operands.
- `n1` is the number of distinct operators.
- `n2` is the number of distinct operands.

The recommended upper limit for this metric is 10. 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 “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Language Scope Calculation

```
int f(int i)
{
    if (i == 1)
        return i;
    else
        return i * g(i-1);
}
```

In this example:

- `N1` = 17.
- `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 (17 + 9) / (12 + 4) = 1.8.

Metric Information

Category: Function
Acronym: `VOCF`

Estimated Function Coupling

Measure of complexity between levels of call tree

Description

This metric is defined as (number of call occurrences – number of function definitions + 1). The metric provides an approximate measure of complexity between different levels of the call tree.

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.

Metric Information

Category: File
Acronym: `FCO`

See Also

Number of Call Occurrences

Number of Call Levels

Maximum depth of nesting of contol 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 with no 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 “Review Code 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

Category: Function
Acronym: `LEVEL`

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.

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>

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>

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.

Metric Information

Category: Function
Acronym: `NCALLS`

See Also

Number of Called Functions

Number of Called Functions

Number of callees of a function

Description

This metric specifies the number of callees of a function.

Calls through a function pointer are not counted. Calls in unreachable code and calls to standard library functions are counted.

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 “Review Code 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>
```

```
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.

Metric Information

Category: Function
Acronym: `CALLS`

See Also

Number of Call Occurrences | Number of Calling Functions

Number of Calling Functions

Number of distinct callers of a function

Description

This metric measures the number of distinct callers of a function.

Calls through a function pointer are not counted. Calls in unreachable code are counted. Even if a caller calls a function more than once, it is counted only once when this metric is calculated.

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 “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

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>

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 calling functions for `fibonacci` is 2. The calling functions are `main` and `fibonacci` itself.

Metric Information

Category: Function
Acronym: `CALLING`

See Also

Number of Called Functions

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 no indirect recursions 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”.

Note: This metric is available only in the Polyspace Metrics web interface.

Examples

Direct Recursion

```
int getVal(void);

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}

int sum(int val) {
    if(val<0)
        return 0;
    else
        return (val + sum(val-1));
}
```

In this example, the number of direct recursions is 1.

Metric Information

Category: Project
Acronym: `AP_CG_DIRECT_CYCLE`

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 without static initializers, 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.

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 9. The calculation excludes:

- The declaration `int sign;`.

- The comment `/* ... */`.
- The two lines with braces only.

Metric Information

Category: Function
Acronym: `FXLN`

See Also

Number of Lines Within Body | Number of Instructions

Number of Files

Number of source files

Description

This metric calculates the number of source files in your project.

Note: This metric is available only in the Polyspace Metrics web interface.

Metric Information

Category: Project
Acronym: `FILES`

See Also

Number of Header Files

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 “Review Code 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

Category: Function
Acronym: `PARAM`

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 “Review Code 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(char *);
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

Category: Function
Acronym: `GOTO`

Number of Header Files

Number of header files

Description

This metric measures the number of header files in the project. Both directly and indirectly included header files are counted. Polyspace internal header files and header files included by those files are also counted.

Note: This metric is available only in the Polyspace Metrics interface.

Metric Information

Category: Project
Acronym: `INCLUDES`

See Also

Number of Files

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 “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

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

Category: Function
Acronym: `STMT`

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.

Metric Information

Category: File
Acronym: `TOTAL_LINES`

See Also

Number of Lines Without Comment

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.

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

Category: Function
Acronym: `FLIN`

See Also

Number of Executable Lines

Number of Lines Without Comment

Number of lines of code excluding comments

Description

This metric calculates the number of lines in a file. When calculating the value of this metric, Polyspace excludes comments and blank lines.

Metric Information

Category: File
Acronym: `LINES_WITHOUT_CMT`

See Also

Number of Lines

Number of Paths

Estimated static path count

Description

This metric measures the number of paths through your source code.

If there are `goto` statements in your code, Polyspace cannot calculate the number of paths.

The recommended upper limit for this metric is 80. If the number of paths 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 “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Function with One Path

```
void func(int ch) {
    switch (ch)
    {
    case 1:
    case 2:
    case 3:
    case 4:
    default:
    }
}
```

In this example, `func` has 1 path.

Function with 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 5 paths. Apart from the path that goes through all the `case`s and `default`, each `break` causes the creation of a new path.

Metric Information

Category: Function
Acronym: PATH

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.

Note: This metric is available only in the Polyspace Metrics web interface. In the Polyspace user interface, each protected shared variable is reported separately. For more information, see Shared protected global variable.

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 options:

Option	Value
Entry points	`task` `interrupt_handler`
Temporally exclusive tasks	`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	
Entry points	`task` `interrupt_handler`	
Critical section details	**Starting procedure**	**Ending procedure**
	`take_semaphore`	`give_semaphore`

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

Category: Project
Acronym: `PSHV`

See Also

"Entry points (C/C++)" | "Critical section details (C/C++)" | "Temporally exclusive tasks (C/C++)"

Number of Recursions

Number of call graph cycles over one or more functions

Description

This metric specifies the number of recursions in your project. Even if more than one function is involved in one recursive cycle, the number of recursions is counted as one.

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. To detect use of recursions, check for violations of MISRA C:2012 Rule 17.2.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code Metrics”.
- In the Polyspace Metrics web interface, see “Compare Metrics Against Software Quality Objectives”.

Examples

Direct Recursion

```
int getVal(void);

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}

int sum(int val) {
    if(val<0)
        return 0;
    else
        return (val + sum(val-1));
}
```

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 operation1() {
    int stop = signal%2;
    if(!stop)
        operation2();
}

void operation2() {
    operation1();
}

void main() {
    operation1();
}
```

In this example, the number of recursions is 1. Although two functions `operation1` and `operation2` indirectly call themselves, they are involved in the same 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.

Indirect Recursion with Two Call Graph Cycles

```
volatile int signal;

void operation1() {
    int stop = signal%3;
    if(stop==1)
        operation2();
    else if(stop==2)
```

```
        operation3();
}

void operation2() {
    operation1();
}

void operation3() {
    operation3();
}

void main() {
    operation1();
}
```

In this example, the number of recursions is 2.

There are two call graph cycles:

- `operation1` → `operation2` → `operation1`
- `operation1` → `operation3` → `operation1`

Same Function Called in Direct and Indirect Recursion

```
volatile int signal;

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 call graph cycles is 1.

If the same function calls itself both directly and indirectly, the two cycles are counted as 1.

Metric Information

Category: Project
Acronym: `AP_CG_CYCLE`

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 there is one return statement, when reading the code, you can easily identify what the function returns.

To enforce limits on metrics:

- In the Polyspace user interface, see “Review Code 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

Category: Function
Acronym: `RETURN`

Number of 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.

Note: This metric is available only in the Polyspace Metrics web interface. In the Polyspace user interface, each unprotected shared variable is reported separately. For more information, see Shared unprotected global variable.

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 `task` and `interrupt_handler` as entry points and do not specify any protection mechanisms.

The operation `shared_var = INT_MAX` can interrupt the other operations on `shared_var` and cause unpredictable behavior.

Metric Information

Category: Project
Acronym: `UNPSHV`

10

Custom Coding Rules

Group 1: Files

Number	Rule Applied	Message generated if rule is violated	Other details
1.1	All source file names must follow the specified pattern.	The source file name “file_name” does not match 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.	The source dir name “dir_name” does not match 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.	The include file name “file_name” does not match 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.	The include dir name “dir_name” does not match the specified pattern.	Only the folder name is checked. An include file is a file that is included.

Group 2: Preprocessing

Number	Rule Applied	Message generated if rule is violated	Other details
2.1	All macros must follow the specified pattern.	The macro “macro_name” does not match the specified pattern.	Macro names are checked before preprocessing.
2.2	All macro parameters must follow the specified pattern.	The macro parameter “param_name” does not match the specified pattern.	Macro parameters are checked before preprocessing.

Group 3: Type definitions

Number	Rule Applied	Message generated if rule is violated	Other details
3.1	All integer types must follow the specified pattern.	The integer type "type_name" does not match the specified pattern.	Applies to integer 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.	The float type "type_name" does not match the specified pattern.	Applies to float types specified by `typedef` statements. For example: `typedef float f32_t;`
3.3	All pointer types must follow the specified pattern.	The pointer type "type_name" does not match 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.	The array type "type_name" does not match the specified pattern.	Applies to array types specified by `typedef` statements. For example: `typedef int[3] a_int_3;`
3.5	All function pointer types must follow the specified pattern.	The function pointer type "type_name" does not match the specified pattern.	Applies to function pointer types specified by `typedef` statements. For example: `typedef void (*pf_callback)(int);`

Group 4: Structures

Number	Rule Applied	Message generated if rule is violated	Other details
4.1	All `struct` tags must follow the specified pattern.	The struct tag "tag_name" does not match the specified pattern.	
4.2	All `struct` types must follow the specified pattern.	The struct type "type_name" does not match the specified pattern.	This is the `typedef` name.
4.3	All `struct` fields must follow the specified pattern.	The struct field "field_name" does not match the specified pattern.	
4.4	All `struct` bit fields must follow the specified pattern.	The struct bit field "field_name" does not match the specified pattern.	

Group 5: Classes (C++)

Number	Rule Applied	Message generated if rule is violated	Other details
5.1	All class names must follow the specified pattern.	The class tag "tag_name" does not match the specified pattern.	
5.2	All class types must follow the specified pattern.	The class type "type_name" does not match the specified pattern.	This is the `typedef` name.
5.3	All data members must follow the specified pattern.	The data member "member_name" does not match the specified pattern.	
5.4	All function members must follow the specified pattern.	The function member "member_name" does not match the specified pattern.	
5.5	All static data members must follow the specified pattern.	The static data member "member_name" does not match the specified pattern.	
5.6	All static function members must follow the specified pattern.	The static function member "member_name" does not match the specified pattern.	
5.7	All bitfield members must follow the specified pattern.	The bitfield "member_name" does not match the specified pattern.	

Group 6: Enumerations

Number	Rule Applied	Message generated if rule is violated	Other details
6.1	All enumeration tags must follow the specified pattern.	The enumeration tag “tag_name” does not match the specified pattern.	
6.2	All enumeration types must follow the specified pattern.	The enumeration type “type_name” does not match the specified pattern.	This is the typedef name.
6.3	All enumeration constants must follow the specified pattern.	The enumeration constant “constant_name” does not match the specified pattern.	

Group 7: Functions

Number	Rule Applied	Message generated if rule is violated	Other details
7.1	All global functions must follow the specified pattern.	The global function “function_name” does not match the specified pattern.	A global function is a function with external linkage.
7.2	All static functions must follow the specified pattern.	The static function “function_name” does not match the specified pattern.	A static function is a function with internal linkage.
7.3	All function parameters must follow the specified pattern.	The function parameter “param_name” does not match the specified pattern.	In C++, applies to non-member functions.

Group 8: Constants

Number	Rule Applied	Message generated if rule is violated	Other details
8.1	All global constants must follow the specified pattern.	The global constant “constant_name” does not match the specified pattern.	A global constant is a constant with external linkage.
8.2	All static constants must follow the specified pattern.	The static constant “constant_name” does not match the specified pattern.	A static constant is a constant with internal linkage.
8.3	All local constants must follow the specified pattern.	The local constant “constant_name” does not match the specified pattern.	A local constant is a constant without linkage.
8.4	All static local constants must follow the specified pattern.	The static local constant “constant_name” does not match the specified pattern.	A static local constant is a constant declared static in a function.

Group 9: Variables

Number	Rule Applied	Message generated if rule is violated	Other details
9.1	All global variables must follow the specified pattern.	The global variable “var_name” does not match the specified pattern.	A global variable is a variable with external linkage.
9.2	All static variables must follow the specified pattern.	The static variable “var_name” does not match the specified pattern.	A static variable is a variable with internal linkage.
9.3	All local variables must follow the specified pattern.	The local variable “var_name” does not match the specified pattern.	A local variable is a variable without linkage.
9.4	All static local variables must follow the specified pattern.	The static local variable “var_name” does not match the specified pattern.	A static local variable is a variable declared static in a function.

Group 10: Name spaces (C++)

Number	Rule Applied	Message generated if rule is violated	Other details
10.1	All names paces must follow the specified pattern.	The name space “name space_name” does not match the specified pattern.	

Group 11: Class templates (C++)

Number	Rule Applied	Message generated if rule is violated	Other details
11.1	All class templates must follow the specified pattern.	The class template “template_name” does not match the specified pattern.	
11.2	All class template parameters must follow the specified pattern.	The class template parameter “param_name” does not match the specified pattern.	

Group 12: Function templates (C++)

Number	Rule Applied	Message generated if rule is violated	Other details
12.1	All function templates must follow the specified pattern.	The function template "template_name" does not match the specified pattern.	Applies to non-member functions.
12.2	All function template parameters must follow the specified pattern.	The function template parameter "param_name" does not match the specified pattern.	Applies to non-member functions.
12.3	All function template members must follow the specified pattern.	The function template member "member_name" does not match the specified pattern.	

11

Global Variables

Shared protected global 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 Summary** 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 options:

Option	Value
Entry points	`task` `interrupt_handler`
Temporally exclusive tasks	`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	
Entry points	`task` `interrupt_handler`	
Critical section details	**Starting procedure**	**Ending procedure**

Option	Value	
	`take_semaphore`	`give_semaphore`

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.

Check Information

Language: C | C++

See Also

Polyspace Analysis Options

"Entry points (C/C++)" | "Critical section details (C/C++)" | "Temporally exclusive tasks (C/C++)"

Polyspace Results

Shared unprotected global variable | Non-shared used global variable | Non-shared unused global variable

More About

- "Multitasking"

Shared unprotected global 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.
- 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 Summary** 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 `task` and `interrupt_handler` as entry points and do not specify any protection mechanisms.

The operation `shared_var = INT_MAX` can interrupt the other operations on `shared_var` and cause unpredictable behavior.

Check Information

Language: C | C++

See Also

Polyspace Analysis Options

"Entry points (C/C++)" | "Critical section details (C/C++)" | "Temporally exclusive tasks (C/C++)"

Polyspace Results

Shared protected global variable | Non-shared used global variable | Non-shared unused global variable

More About

- "Multitasking"

Non-shared used global 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 Summary** 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` and `var3` as non-shared unused variables, and `var1` and `var4` as non-shared used variables.

`var3` is used in code that is deactivated by a condition that is always false and is therefore unused.

Note: In a C++ project, the software lists `var3` as an used variable and does not list the unused variable `var2`.

Check Information

Language: C | C++

See Also

Shared protected global variable | Shared unprotected global variable | Non-shared unused global variable

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 Summary** and **Variable Access** panes. On the **Source** pane, the coloring is applied to the variable only during declaration.

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` and `var3` as non-shared unused variables, and `var1` and `var4` as non-shared used variables.

`var3` is used in code that is deactivated by a condition that is always false and is therefore unused.

Note: In a C++ project, the software lists `var3` as an used variable and does not list the unused variable `var2`.

Check Information

Language: C | C++

See Also

Shared protected global variable | Shared unprotected global variable | Non-shared used global variable