Polyspace® Bug Finder™

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® Bug Finder™ Reference

Trademarks

Patents

Revision History

September 2013	Online only	New for Version 1.0 (Release 2013b)
March 2014	Online only	Revised for Version 1.1 (Release 2014a)
October 2014	Online only	Revised for Version 1.2 (Release 2014b)
March 2015	Online only	Revised for Version 1.3 (Release 2015a)

1 Option Descriptions

2 Option Descriptions for C++ Code

8 Code Metrics

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-bug-finder-nodesktop -os-target Linux`

See Also

"Target processor type (C)" on page 1-5 | "Dialect (C)" on page 1-12 | "Dialect (C ++)" on page 2-4

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-bug-finder-nodesktop -lang c -target m68k`

See Also

"Generic target options (C/C++)" on page 1-8

Related Examples

- "Specify Analysis Options"

- “Modify Predefined Target Processor Attributes”
- “Specify 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-bug-finder-nodesktop -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-bug-finder-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-bug-finder-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-bug-finder-nodesktop -target mcpu -char-is-16bits`

Option	Description	Available With...	Example
`-short-is-8bits`	Define `short` as 8 bits, regardless of sign	`mcpu`	`polyspace-bug-finder-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-bug-finder-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-bug-finder-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-bug-finder-nodesktop -target mcpu -double-is-64bits`

`-pointer-is-32bits`	Define pointer as 32 bits, regardless of sign. Alignment is also 32 bits.	`mcpu`	`polyspace-bug-finder-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-bug-finder-nodesktop -target mcpu -align 16`

See Also

"Target processor type (C)" on page 1-5 | "Target processor type (C++)" on page 2-2

Related Examples

- "Specify 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 1-13.

`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.

  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 is ignored by Bug Finder.
- **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-bug-finder-nodesktop -lang c -sources "`*`file1.c,file2.c`*`" -OS-target Linux -dialect gnu4.6`

See Also

“Target operating system (C/C++)” on page 1-3 | “Target processor type (C)” on page 1-5

Related Examples

- “Analyze 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-bug-finder-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-bug-finder-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-bug-finder-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-bug-finder-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-bug-finder-nodesktop -D HAVE_MYLIB -D int32_t=int`

See Also

"Disabled preprocessor definitions (C/C++)" on page 1-21

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-bug-finder-nodesktop -U HAVE_MYLIB -U USE_COM1`

See Also

“Preprocessor definitions (C/C++)” on page 1-19

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-bug-finder-nodesktop -dos -I ./my_copied_include_dir -D test=1`

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-bug-finder-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-bug-finder-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

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-bug-finder-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-bug-finder-nodesktop -I /com1/inc -I /com1/sys/inc`

See Also

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

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 1-28 | "Ignore default initialization of global variables (C)"

Related Examples

- "Specify Analysis Options"
- "Specify Constraints"

More About

- "Constraints"

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.

Command-Line Information

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

Related Examples

- "Specify Analysis Options"

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.

□ Off

The code is not intended for a multitasking 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

- “Set Up Multitasking Analysis”
- “Review Concurrency Defects”

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.

Tips

- 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-bug-finder-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”
- “Set Up Multitasking Analysis”
- “Review Concurrency Defects”

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-bug_finder-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

Data race | Data race including atomic operations

Related Examples

- “Specify Analysis Options”
- “Set Up Multitasking Analysis”
- “Review Concurrency Defects”

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-bug-finder-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

Data race | Data race including atomic operations

Related Examples

- “Specify Analysis Options”
- “Set Up Multitasking Analysis”
- “Review Concurrency Defects”

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-bug-finder-nodesktop -sources` *`file_name`* `-misra2 all-rules`

See Also

"Files and folders to ignore (C)"

Related Examples

- "Specify Analysis Options"
- "Activate Coding Rules Checker"
- "Select Specific MISRA or JSF Coding Rules"

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-bug-finder-nodesktop -sources` *`file_name`* `-misra-ac-agc all-rules`

Related Examples

- "Specify Analysis Options"
- "Activate Coding Rules Checker"
- "Select Specific MISRA or JSF Coding Rules"

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-bug-finder-nodesktop -lang c -sources *file_name* -misra3 mandatory-required

See Also

"Files and folders to ignore (C)"

Related Examples

- "Specify Analysis Options"
- "Activate Coding Rules Checker"
- "Select Specific MISRA or JSF Coding Rules"

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-bug-finder-nodesktop -sources` *`file_name`* `-misra3 all -misra3-agc-mode`

See Also

"Files and folders to ignore (C)" | "Check MISRA C:2012" on page 1-40

Related Examples

- "Specify Analysis Options"
- "Activate Coding Rules Checker"

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-bug-finder-nodesktop -sources` *`file_name`* `-custom-rules "C:\Rules\myrules.txt"`

Related Examples

- "Specify Analysis Options"
- "Activate Coding Rules Checker"
- "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 and during Bug Finder defect checking. This option is available on the **Inputs & Stubbing** node in the **Configuration** pane.

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 ✖.

Command-Line Information

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

Related Examples

- "Specify Analysis Options"
- "Activate Coding Rules Checker"

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-bug-finder-nodesktop -sources *filename* -misra2 required-rules -boolean-types boolean1_t,boolean2_t`

Related Examples

- “Activate Coding Rules Checker”
- “Specify 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-bug-finder-nodesktop -sources` *`filename`* `-misra2 required-rules -allowed-pragmas pragma_01,pragma_02`

Related Examples

- “Activate Coding Rules Checker”

More About

- “MISRA C:2004 Coding Rules”

Find defects (C/C++)

Enable or disable defect checking. Activate different defect checkers. This option is available on the **Bug Finder Analysis** node in the **Configuration** pane.

Settings

Default: `default`

`default`

A list of default defects defined by the software. For information on which defects are default, refer to the individual defect reference pages.

`all`

All defects.

`custom`

Choose the defects you want to find by selecting categories of checkers or specific defects.

Command-Line Information

Regardless of order, the shell script processes the `-checkers` option, and then `-disable-checkers` option.

Refer to the individual defect reference pages for the command-line parameters values.
Parameter: `-checkers`
Value: `default` | `all` | category | defect parameter
Default: `default`
Parameter: `-disable-checkers`
Value: category | defect parameter
Example: `polyspace-bug-finder-nodesktop -sources` *`filename`* `-checkers numerical,dataflow -disable-checkers FLOAT_ZERO_DIV`
Example: `polyspace-bug-finder-nodesktop -sources` *`filename`* `-checkers default -disable-checkers concurrency,dead_code`

See Also

"Numerical Defects" | "Static Memory Defects" | "Dynamic Memory Defects" | "Programming Defects" | "Data-flow Defects" | "Other Defects"

Related Examples

- "Specify Analysis Options"

More About

- "Bug Finder Defect Categories"

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 Reports”

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 *`MATLAB_Install`*`\polyspace\toolbox\psrptgen\templates\bug_finder`.

Settings

Default: `BugFinderSummary`

`BugFinderSummary`

The report lists:

- **Polyspace Bug Finder Summary**: Number of result sets and number of defects in the source code.
- **Code Metrics**: Various complexity metrics. For more information, see “Code Metrics”.
- **Defect Summary**: Defects that Polyspace Bug Finder™ looks for. For each defect, the report lists the:
  - Category of the defect.
  - Defect name.
  - Number of instances of the defect found in the source code.

`BugFinder`

The report lists:

- **Polyspace Bug Finder Summary**: Number of result sets and number of defects in the source code.
- **Code Metrics**: Various quantities related to the source code. For more information, see “Code Metrics”.
- **Defects**: Defects found in the source code. For each defect, the report lists the:
  - Function containing the defect.
  - Defect information on the **Check Details** pane.
  - Review information, such as **Classification**, **Status** and **Comment**.

- **Configuration Settings**: List of analysis options that Polyspace uses for analysis. For more information, see “Analysis Options for C” or “Analysis Options for C++”.

`BugFinder_CWE`

The report contains the same information as the `BugFinder` report. However, in the **Defects** chapter, an additional column lists the CWE™ identifiers for each defect.

`CodeMetrics`

The report lists the following:

- **Code Metrics Summary**: Various quantities related to the source code. For more information, see “Code Metrics”.
- **Code Metrics Details**: Various quantities related to the source code with the information broken down by file and function.

Dependencies

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

Command-Line Information

Parameter: `-report-template`
Value: Name of template with extension `.rpt`
Example: `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-report-template BugFinder.rpt`

See Also

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

Related Examples

- “Generate Reports”

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-bug-finder-nodesktop -sources` *`file_name`* `-report-output-format pdf`

See Also

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

Related Examples

- "Specify Analysis Options"
- "Generate Reports"

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.

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-bug-finder-nodesktop -batch -scheduler NodeHost`
`polyspace-bug-finder-nodesktop -batch -scheduler MJSName@NodeHost`

See Also

"Add to results repository (C/C++)" on page 1-61 | `-scheduler`

Related Examples

- "Specify Analysis Options"
- "Set Up Server for Remote Verification and Analysis"

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-bug-finder-nodesktop -batch -scheduler NodeHost -add-to-results-repository`

See Also

"Set Up Server for Remote Verification and Analysis" | "Set Up Polyspace Metrics" | "Batch (C/C++)" on page 1-59

Related Examples

- "Run Remote Batch Analysis"

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-bug-finder-nodesktop -sources` *`file_name`* `-code-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-bug-finder-nodesktop -sources` *`file_name`* ``-post-analysis-command `pwd`/send_email``

Related Examples

- "Specify Analysis Options"

Other (C)

In this section...
"`-extra-flags`" on page 1-64
"`-c-extra-flags`" on page 1-64
"`-cfe-extra-flags`" on page 1-65
"`-il-extra-flags`" on page 1-65

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

-extra-flags

This dialog box is for adding nonofficial or expert options to the analyzer. Each word of the option (even the parameters) must be preceded by *-extra-flags*.

These flags will be given to you by MathWorks if required.

No Default

Example Shell Script Entry:

```
 polyspace-bug-finder-nodesktop -extra-flags -param1 -extra-flags -
param2 \

  -extra-flags 10 ...
```

-c-extra-flags

This option is used to specify an expert option to be added to an analysis. Each word of the option (even the parameters) must be preceded by *-c-extra-flags*.

These flags will be given to you by MathWorks if required.

No Default

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -c-extra-flags -param1 -c-extra-flags
-param2 -c-extra-flags 10
```

-cfe-extra-flags

This option is used to specify an expert option for an analysis.

These flags will be given to you by MathWorks if required.

No Default

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -cfe-extra-flags -param1 -cfe-extra-
flags -param2
```

-il-extra-flags

This option is used to specify an expert option to be added to an analysis. Each word of the option (even the parameters) must be preceded by *-il-extra-flags*.

These flags will be given to you by MathWorks if required.

No Default

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -il-extra-flags -param1 -il-extra-
flags -param2 -il-extra-flags 10
```

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 **Main Generator** 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-bug-finder-nodesktop -sources *file_name* -main-generator -functions-called-after-loop myfunc`

See Also

"Parameters (C)" on page 1-70 | "Inputs (C)" on page 1-72 | "Initialization functions (C)" on page 1-67 | "Step functions (C)" on page 1-68

Related Examples

- "Specify Analysis Options"
- "Configure Simulink Model"

More About

- "Recommended Polyspace Bug Finder Options for Analyzing Generated Code"
- "Main Generation for Model Analysis"

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 **Main Generator** 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-bug-finder-nodesktop -sources *file_name* -main-generator -functions-called-before-loop myfunc`

See Also

“Parameters (C)” on page 1-70 | “Inputs (C)” on page 1-72 | “Step functions (C)” on page 1-68 | “Termination functions (C)” on page 1-66

Related Examples

- “Specify Analysis Options”
- “Configure Simulink Model”

More About

- “Recommended Polyspace Bug Finder Options for Analyzing Generated Code”
- “Main Generation for Model Analysis”

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 **Main Generator** 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-bug-finder-nodesktop -sources` *`file_name`* `-main-generator -functions-called-in-loop all`

See Also

"Parameters (C)" on page 1-70 | "Inputs (C)" on page 1-72 | "Initialization functions (C)" on page 1-67 | "Step functions (C)" on page 1-68 | "Termination functions (C)" on page 1-66

Related Examples

- "Specify Analysis Options"
- "Configure Simulink Model"

More About

- "Recommended Polyspace Bug Finder Options for Analyzing Generated Code"
- "Main Generation for Model Analysis"

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 **Main Generator** 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-bug-finder-nodesktop -sources` *`file_name`* `-main-generator -variables-written-before-loop all`

See Also

"Inputs (C)" on page 1-72 | "Initialization functions (C)" on page 1-67 | "Step functions (C)" on page 1-68 | "Termination functions (C)" on page 1-66

Related Examples

- “Specify Analysis Options”
- “Configure Simulink Model”

More About

- “Recommended Polyspace Bug Finder Options for Analyzing Generated Code”
- “Main Generation for Model Analysis”

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 **Main Generator** 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-bug-finder-nodesktop -sources` *`file_name`* `-main-generator -variables-written-in-loop all`

See Also

“Parameters (C)” on page 1-70 | “Initialization functions (C)” on page 1-67 | “Step functions (C)” on page 1-68 | “Termination functions (C)” on page 1-66

Related Examples

- “Specify Analysis Options”
- “Configure Simulink Model”

More About

- “Recommended Polyspace Bug Finder Options for Analyzing Generated Code”
- “Main Generation for Model Analysis”

Verify module (C)

This option is available only for model-generated code.

Specify that Polyspace must generate a `main` function if it does not find one in the source files. This option is available on the **Main Generator** 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-bug-finder-nodesktop -sources` *`file_name`* `-main-generator ...`

See Also

“Parameters (C)” on page 1-70 | “Inputs (C)” on page 1-72 | “Initialization functions (C)” on page 1-67 | “Step functions (C)” on page 1-68 | “Termination functions (C)” on page 1-66

Related Examples

- “Specify Analysis Options”
- “Configure Simulink Model”

More About

- “Main Generation for Model Analysis”

2

Option Descriptions for 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-bug-finder-nodesktop -lang cpp -target powerpc`

See Also

"Generic target options (C/C++)" on page 1-8

Related Examples

- "Specify Analysis Options"
- "Modify Predefined Target Processor Attributes"
- "Specify 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 2-5.

`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** — Limited support.

  If you encounter 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.

  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 is ignored by Bug Finder.
- **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`.

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-bug-finder-nodesktop -lang cpp -sources "`*`file1.cpp,file2.cpp`*`" -OS-target Visual -dialect visual7.1`

See Also

“Target operating system (C/C++)” on page 1-3 | “Target processor type (C++)” on page 2-2 | “C++11 Extensions (C++)” on page 2-9 | “Block char16/32_t types (C++)” on page 2-10

Related Examples

- “Analyze 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-bug-finder-nodesktop -lang cpp -cpp11-extension`

See Also

“Dialect (C++)” on page 2-4 | “Block char16/32_t types (C++)” on page 2-10

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-bug-finder-nodesktop -lang cpp -dialect gnu4.7 -cpp11-extension -no-uliterals`

See Also

"Dialect (C++)" on page 2-4 | "C++11 Extensions (C++)" on page 2-9

More About

- "Supported C++ 2011 Standards"

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-bug-finder-nodesktop -lang cpp -pack-alignment-value 4`

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-bug-finder-nodesktop -OS-target Visual -dialect visual8 -import-dir /com1/inc`

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-bug-finder-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-bug-finder-nodesktop -lang cpp -OS-target Visual -support-FX-option-results`

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-bug-finder-nodesktop -lang cpp -enum-type-definition auto-signed-first`

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-bug-finder-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-bug-finder-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-bug-finder-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-bug-finder-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-bug-finder-nodesktop -lang cpp -no-extern-C`

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)`

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`

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-bug-finder-nodesktop -sources` *`file_name`* `-misra-cpp all-rules`

Related Examples

- "Specify Analysis Options"
- "Activate Coding Rules Checker"
- "Select Specific MISRA or JSF Coding Rules"

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-bug-finder-nodesktop -sources `*`file_name`*` -jsf-coding-rules all-rules`

Related Examples

- “Specify Analysis Options”
- “Activate Coding Rules Checker”
- “Select Specific MISRA or JSF Coding Rules”

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 and during Bug Finder defect checking. This option is available on the **Inputs & Stubbing** node in the **Configuration** pane.

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 ✖.

Command-Line Information

Parameter: `-includes-to-ignore`
Value: `all-headers | all |` *`file1`*`[,`*`file2`*`[,...]] |` *`folder1`*`[,`*`folder2`*`[,...]]`
Default: `all-headers`
Example: `polyspace-bug-finder-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”

Related Examples

- “Specify Analysis Options”
- “Activate Coding Rules Checker”

Other (C++)

This option is for adding nonofficial or expert options to the analyzer. This option is available on the **Advanced Settings** node in the **Configuration** pane. Each word of the option (even the parameters) must be preceded by *`-extra-flags`*.

These flags will be given to you by MathWorks if required.

No Default

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -extra-flags -param1 -extra-flags -
param2
```

-cpp-extra-flags flag

It specifies an expert option to be added to a C++ analysis. Each word of the option (even the parameters) must be preceded by *`-cpp-extra-flags`*.

These flags will be given to you by MathWorks if required.

No Default

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -cpp-extra-flags -stubbed-new-may-
return-null
```

-il-extra-flags flag

It specifies an expert option to be added to a C++ analysis. Each word of the option (even the parameters) must be preceded by *`-il-extra-flags`*.

These flags will be given to you by MathWorks if required.

No Default

Example Shell Script Entry:

```
polyspace-bug-finder-nodesktop -il-extra-flags flag
```

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 **Main Generator** 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-bug-finder-nodesktop -sources *file_name* -main-generator -functions-called-after-loop myfunc`

See Also

"Parameters (C++)" on page 2-34 | "Inputs (C++)" on page 2-36 | "Initialization functions (C++)" on page 2-31 | "Step functions (C++)" on page 2-32

Related Examples

- "Specify Analysis Options"
- "Configure Simulink Model"

More About

- "Recommended Polyspace Bug Finder Options for Analyzing Generated Code"

- “Main Generation for Model Analysis”

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 **Main Generator** 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-bug-finder-nodesktop -sources` *`file_name`* `-main-generator -functions-called-before-loop myfunc`

See Also

“Parameters (C++)” on page 2-34 | “Inputs (C++)” on page 2-36 | “Step functions (C++)” on page 2-32 | “Termination functions (C++)” on page 2-29

Related Examples

- “Specify Analysis Options”
- “Configure Simulink Model”

More About

- “Recommended Polyspace Bug Finder Options for Analyzing Generated Code”
- “Main Generation for Model Analysis”

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 **Main Generator** 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-bug-finder-nodesktop -sources` *`file_name`* `-main-generator -functions-called-in-loop all`

See Also

"Parameters (C++)" on page 2-34 | "Inputs (C++)" on page 2-36 | "Initialization functions (C++)" on page 2-31 | "Termination functions (C++)" on page 2-29

Related Examples

- "Specify Analysis Options"
- "Configure Simulink Model"

More About

- "Recommended Polyspace Bug Finder Options for Analyzing Generated Code"
- "Main Generation for Model Analysis"

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 **Main Generator** 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-bug-finder-nodesktop -sources` *`file_name`* `-main-generator -variables-written-before-loop all`

See Also

“Inputs (C++)” on page 2-36 | “Initialization functions (C++)” on page 2-31 | “Step functions (C++)” on page 2-32 | “Termination functions (C++)” on page 2-29

Related Examples

- “Specify Analysis Options”

- “Configure Simulink Model”

More About

- “Recommended Polyspace Bug Finder Options for Analyzing Generated Code”
- “Main Generation for Model Analysis”

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 **Main Generator** 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-bug-finder-nodesktop -sources` *`file_name`* `-main-generator -variables-written-in-loop all`

See Also

"Parameters (C++)" on page 2-34 | "Initialization functions (C++)" on page 2-31 | "Step functions (C++)" on page 2-32 | "Termination functions (C++)" on page 2-29

Related Examples

- "Specify Analysis Options"

- “Configure Simulink Model”

More About

- “Recommended Polyspace Bug Finder Options for Analyzing Generated Code”
- “Main Generation for Model Analysis”

Verify module (C++)

This option is available only for model-generated code.

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 **Main Generator** 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-bug-finder-nodesktop -sources` *`file_name`* `-main-generator ...`

See Also

"Initialization functions (C++)" on page 2-31

Related Examples

- "Specify Analysis Options"
- "Configure Simulink Model"

More About

- "Main Generation for Model Analysis"

Polyspace Command-Line Options

-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-bug-finder-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

`polyspaceBugFinder`

-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-bug-finder-nodesktop -author "John Smith"
```

See Also

`-date` | `-prog` | `polyspaceBugFinder`

-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-bug-finder-nodesktop -date "15/03/2012"
```

See Also

`-author` | `-prog` | `polyspaceBugFinder` | `polyspaceCodeProver`

-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-bug-finder-nodesktop -h
polyspace-bug-finder-nodesktop -help
```

See Also

`polyspaceBugFinder`

-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-bug-finder-nodesktop -I /com1/inc -I /com1/sys/inc
```

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

```
polyspace-bug-finder-nodesktop -I /com1/inc -I /com1/sys/inc
                                                     -I ./sources
```

See Also

`polyspaceBugFinder`

-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-bug-finder-nodesktop -version 1.3
       -import-comments C:\Results\myProj\1.2
```

See Also

`-version` | `polyspaceBugFinder`

-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 New Project”.

Examples

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

```
polyspace-bug-finder-nodesktop -lang cpp -sources...
```

See Also

`polyspaceBugFinder`

-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 the maximum number of available CPUs.

Examples

Disable parallel processing during the analysis.

```
polyspace-bug-finder-nodesktop -max-processes 1
```

See Also

`polyspaceBugFinder`

-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 MyBugFinderProject
-lang c
-prog MyBugFinderProject
-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
-checkers default
-disable-checkers concurrency
-results-dir C:\Polyspace\MyBugFinderProject
```

2 Run Polyspace using options in the file `listofoptions.txt`.

```
polyspace-bug-finder-nodesktop -options-file listofoptions.txt
```

See Also

`polyspaceBugFinder` | `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-bug-finder-nodesktop -prog MyApp
```

See Also

`-author` | `-date` | `polyspaceBugFinder`

-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-bug-finder-nodesktop -report-template Developer
      -report-output-name Airbag_v3.rtf
```

See Also

"Output format (C/C++)" on page 1-57 | "Report template (C/C++)" | `polyspaceBugFinder`

-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-bug-finder-nodesktop -results-dir RESULTS ...
   export RESULTS=results_'date + %d%B_%HH%M_%A'
polyspace-bug-finder-nodesktop -results-dir 'pwd'/$RESULTS
```

See Also

`polyspaceBugFinder`

-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-bug-finder-nodesktop -batch -scheduler NodeHost
polyspace-bug-finder-nodesktop -batch -scheduler 192.168.1.124:12400
polyspace-bug-finder-nodesktop -batch -scheduler MJSName@NodeHost
```

```
polyspace-job-manager listjobs -scheduler NodeHost
```

See Also

`polyspaceBugFinder` | `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-bug-finder-nodesktop -sources mymain.c
     -sources funAlgebra.c -sources funGeometry.c
```

See Also

`polyspaceBugFinder`

-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-bug-finder-nodesktop -batch -scheduler NODEHOST
  -sources-list-file "C:\Analysis\files.txt
polyspace-bug-finder-nodesktop -batch -scheduler NODEHOST
  -sources-list-file "/home/polyspace/files.txt"
```

See Also

`polyspaceBugFinder`

-termination-functions

Specify process termination functions

Syntax

```
-termination-functions function1[,function2[,...]]
```

Description

`-termination-functions` *`function1`*`[,`*`function2`*`[,...]]` specifies functions that behave like the exit function and terminate your program.

Use this option to specify program termination functions that are declared but not defined in your code.

Examples

Polyspace detects an **Integer division by zero** defect in the following code because it does not recognize that `my_exit` terminates the program.

```
void my_exit();

double reciprocal(int val) {
  if(val==0)
    my_exit();
  return (1/val);
}
```

To prevent Polyspace from flagging the division operation, use the `-termination-functions` option:

```
polyspace-bug-finder-nodesktop -termination-functions my_exit
```

See Also

`polyspaceBugFinder`

-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-bug-finder-nodesktop -tmp-dir-in-results-dir
```

See Also

`polyspaceBugFinder`

-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-bug-finder-nodesktop -v
```

See Also

`polyspaceBugFinder`

4

Checks

Arithmetic operation with NULL pointer

Arithmetic operation performed on NULL pointer

Description

Arithmetic operation with NULL pointer occurs when an arithmetic operation involves a pointer whose value is NULL.

Examples

Arithmetic Operation with NULL Pointer Error

```
#include<stdlib.h>

int Check_Next_Value(int *loc, int val)
 {
  int *ptr= *loc, found = 0;

  if (ptr==NULL)
   {
      ptr++;
      /* Defect: NULL pointer shifted */

      if (*ptr==val) found=1;
   }

  return(found);
 }
```

When ptr is a NULL pointer, the code enters the if statement body. Therefore, a NULL pointer is shifted in the statement ptr++.

Correction — Avoid NULL Pointer Arithmetic

One possible correction is to perform the arithmetic operation when ptr is not NULL.

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

```
int Check_Next_Value(int *loc, int val)
 {
  int *ptr= *loc, found = 0;

  /* Fix: Perform operation when ptr is not NULL */
  if (ptr!=NULL)
   {
      ptr++;

      if (*ptr==val) found=1;
   }

  return(found);
 }
```

Check Information

Category: Static memory
Language: C | C++
Default: off
Command-Line Syntax: `null_ptr_arith`

See Also

Null pointer | "Find defects (C/C++)"

More About

- "Static Memory Defects"
- "Review and Comment Results"

Array access out of bounds

Array index outside bounds during array access

Description

Array access out of bounds occurs when an array index falls outside the range `[0...array_size-1]` during array access.

Examples

Array Access Out of Bounds Error

```
#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]);
    /* Defect: Value of i is greater than allowed value of 9 */
}
```

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, the `printf` statement attempts to access `fib[10]` through `i`.

Correction — Keep Array Index Within Array Bounds

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];
    }

    /* Fix: Print fib[9] instead of fib[10] */
    printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}
```

The `printf` statement accesses `fib[9]` instead of `fib[10]`.

Check Information

Category: Static memory
Language: C | C++
Default: on
Command-Line Syntax: `out_bound_array`

See Also

Pointer access out of bounds | "Find defects (C/C++)"

More About

- "Static Memory Defects"
- "Review and Comment Results"

Assertion

Failed assertion statement

Description

Assertion occurs when you use an `assert`, and the asserted expression is or could be false.

Examples

Check Assertion on Unsigned Integer

```
void asserting_x(unsigned int theta) {

    theta =+ 5;
    assert(theta < 0);
}
```

In this example, the `assert` function checks if the input variable, `theta`, is less than or equal to zero. The assertion fails because `theta` is an unsigned integer, so the value at the beginning of the function is at least zero. This positive value is increased by five. Therefore, the range of `theta` is `[5..MAX_INT]`. `theta` is always greater than zero.

Correction — Change Assert Expression

One possible correction is to change the assertion expression. By changing the *less-than-or-equal-to* sign to a *greater-than-or-equal-to* sign, the assertion does not fail.

```
void asserting_x(unsigned int theta) {

    theta =+ 5;
    assert(theta > 0);
}
```

Correction — Fix Code

One possible correction is to fix the code related to the assertion expression. If the assertion expression is true, fix your code so the assertion passes.

```
void asserting_x(int theta) {

    theta = -abs(theta);
    assert(theta < 0);
}
```

Check Information

Category: Other
Language: C | C++
Default: on
Command-Line Syntax: `assert`

See Also

"Find defects (C/C++)"

More About

- "Other Defects"
- "Review and Comment Results"

Code deactivated by constant false condition

Code segment deactivated by #if 0 directive or if(0) condition

Description

Code deactivated by constant false condition occurs when a block of code is deactivated using a #if 0 directive or if(0) condition.

Examples

Code Deactivated by Constant False Condition Error

```
#include<stdio.h>
int Trim_Value(int* Arr,int Size,int Cutoff)
{
    int Count=0;

    for(int i=0;i < Size;i++){
        if(Arr[i]>Cutoff){
            Arr[i]=Cutoff;
            Count++;
        }
    }

    #if 0
    /* Defect: Code Segment Deactivated */

    if(Count==0){
        printf("Values less than cutoff.");
    }
     #endif

    return Count;
}
```

In the preceding code, the printf statement is placed within a #if #endif directive. The software treats the portion within the directive as code comments and not compiled.

Correction — Change `#if 0` to `#if 1`

Unless you intended to deactivate the `printf` statement, one possible correction is to reactivate the block of code in the `#if #endif` directive. To reactivate the block, change `#if 0` to `#if 1`.

```
#include<stdio.h>
int Trim_Value(int* Arr,int Size,int Cutoff)
{
 int Count=0;

 for(int i=0;i < Size;i++)
     {
      if(Arr[i]>Cutoff)
            {
             Arr[i]=Cutoff;
             Count++;
            }
     }


 /* Fix: Replace #if 0 by #if 1 */
 #if 1
      if(Count==0)
           {
            printf("Values less than cutoff.");
           }
 #endif

 return Count;
}
```

Check Information

Category: Data-flow
Language: C | C++
Default: off
Command-Line Syntax: `deactivated_code`

See Also

"Find defects (C/C++)" | `Dead code` | `Unreachable code` | `Useless if`

More About

- “Data-flow Defects”
- “Review and Comment Results”

Data race

Multiple tasks perform unprotected non-atomic operations on shared variables

Description

Data race occurs when:

- Multiple tasks perform unprotected operations on a shared variable.
- At least one task performs a read operation and another task performs a write operation.
- At least one operation is non-atomic. For data race on both atomic and non-atomic operations, see Data race including atomic operations.

A non-atomic operation can translate into more than one machine instruction. For instance:

- The operation can involve both a read and write operation. For example, `var++` involves reading the value of `var`, increasing the value by one and writing the increased value back to `var`.
- The operation can involve a 64-bit variable on a 32-bit target. For example, the operation

  ```
  long long var1, var2;
  var1=var2;
  ```
  involves two steps in copying the content of `var2` to `var1` on certain targets.

  Polyspace uses the **Pointer** size for your **Target processor type** as the threshold to compute atomicity. For instance, if you use `i386` as your **Target processor type**, the **Pointer** size is 32 bits, and **Long long** and **Double** sizes are both 64 bits. Therefore, Polyspace considers copying one `long long` or `double` variable to another as non-atomic.
- The operation can involve writing the return value of a function call to a shared variable. For example, the operation `x=func()` involves calling `func` and writing the return value of `func` to `x`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Examples

Multiple Tasks Call the Same Function

```
int var;
void begin_critical_section();
void end_critical_section();

void increment(void) {
    var++;
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Value	
Entry points	task1 task2 task3	
Critical section details	**Starting procedure**	**Ending procedure**
	begin_critical_section	end_critical_section

In this example, the tasks task1, task2, and task3 call the function increment. increment contains the operation var++ that can involve multiple machine instructions including:

- Reading var.
- Writing an increased value to var.

These machine instructions, when executed from task1 and task2, can occur concurrently in an unpredictable sequence. For example, reading var from task1 can occur either before or after writing to var from task2. Therefore the value of var can be unpredictable.

Correction — Place Critical Section Inside Function Call

One possible correction is to place the operation var++ in a critical section. When task1 enters its critical section, the other tasks cannot enter their critical sections until task1 leaves its critical section. The operation var++ from the three tasks cannot interfere with each other.

To implement the critical section, in the function increment, place the operation var++ between calls to begin_critical_section and end_critical_section.

```
int var;
void begin_critical_section();
void end_critical_section();
void increment(void) {
      begin_critical_section();
      var++;
      end_critical_section();
}

void task1(void)  {
      increment();
}

void task2(void)  {
      increment();
}

void task3(void)  {
      increment();
}
```

Correction — Place Critical Section Outside Function Call

Another possible correction is to call increment in the same critical section in the three tasks. When task1 enters its critical section, the other tasks cannot enter their critical sections until task1 leaves its critical section. The calls to increment from the three tasks cannot interfere with each other.

To implement the critical section, in each of the three tasks, call increment between calls to begin_critical_section and end_critical_section.

```
int var;
void begin_critical_section();
void end_critical_section();
void increment(void) {
      var++;
}

void task1(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task2(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}

void task3(void)  {
     begin_critical_section();
     increment();
     end_critical_section();
}
```

Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks task1 and task2 temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane:

1. Select **Multitasking**.
2. For **Temporally exclusive tasks**, enter task1 task2.

Check Information

Category: Other

Language: C | C++
Default: On
Command-Line Syntax: `data_race`

See Also

“Find defects (C/C++)” | Data race including atomic operations | Deadlock | Double lock | Double unlock | Missing lock | Missing unlock | “Target processor type (C)” | “Target processor type (C++)”

More About

- “Set Up Multitasking Analysis”
- “Review Concurrency Defects”

Data race including atomic operations

Multiple tasks perform unprotected operations on shared variables

Description

Data race occurs when:

- Multiple tasks perform unprotected operations on a shared variable.
- At least one task performs a read operation and another task performs a write operation.

The operations can be either atomic or non-atomic. For data race on non-atomic operations alone, see Data race.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Examples

Data Race on Atomic Access

```
#include<stdio.h>

int var;
void begin_critical_section();
void end_critical_section();

void task1(void) {
    var = 1;
}

void task2(void) {
    int local_var;
    local_var = var;
    printf("%d", local_var);
}

void task3(void) {
    begin_critical_section();
    var++;
    end_critical_section();
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Value	

Entry points	`task1`
	`task2`
	`task3`	

Critical section details	**Starting procedure**	**Ending procedure**
	`begin_critical_section`	`end_critical_section`

In this example, the write operation `var=1;` in task `task1` executes concurrently with the read operation `local_var=var;` in task `task2`.

Correction — Use Critical Sections

One possible correction is to place the operations

- `var=1;` in `task1`
- `local_var=var;` in `task2`

in the same critical section. When `task1` enters its critical section, the other tasks cannot enter their critical sections until `task1` leaves its critical section. Therefore, the two operations cannot execute concurrently.

To implement the critical section, place the two operations between calls to `begin_critical_section` and `end_critical_section`.

```
#include<stdio.h>

int var;
void begin_critical_section();
void end_critical_section();

void task1(void) {
    begin_critical_section();
    var = 1;
    end_critical_section();
}

void task2(void) {
    int local_var;
    begin_critical_section();
    local_var = var;
    end_critical_section();
    printf("%d", local_var);
}

void task3(void) {
    begin_critical_section();
    var++;
    end_critical_section();
}
```

Correction — Make Tasks Temporally Exclusive

Another possible correction is to make the tasks `task1` and `task2` temporally exclusive. Temporally exclusive tasks cannot execute concurrently.

On the **Configuration** pane:

1. Select **Multitasking**.
2. For **Temporally exclusive tasks**, enter `task1 task2`.

Check Information

Category: Other
Language: C | C++
Default: Off
Command-Line Syntax: `data_race_all`

See Also

"Find defects (C/C++)" | Data race | Deadlock | Double lock | Double unlock | Missing lock | Missing unlock

More About

- "Set Up Multitasking Analysis"

- “Review Concurrency Defects”

Deadlock

Call sequence to lock functions cause two tasks to block each other

Description

Deadlock occurs when multiple tasks are stuck in their critical sections (CS) because:

- Each CS waits for another CS to end.
- The critical sections (CS) form a closed cycle. For example:
  - CS #1 waits for CS #2 to end, and CS #2 waits for CS #1 to end.
  - CS #1 waits for CS #2 to end, CS #2 waits for CS #3 to end and CS #3 waits for CS #1 to end.

Polyspace expects critical sections of code to follow a specific format. A critical section lies between a call to a lock function and a call to an unlock function. 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. Both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Examples

Deadlock with Two Tasks

```
void task1(void);
void task2(void);

int var;
void perform_task_cycle(void) {
 var++;
}

void begin_critical_section_1(void);
void end_critical_section_1(void);
void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_2();
    begin_critical_section_1();
    perform_task_cycle();
    end_critical_section_1();
    end_critical_section_2();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Value	
Entry points	task1 task2	
Critical section details	**Starting procedure**	**Ending procedure**
	begin_critical_section_	end_critical_section_1
	begin_critical_section_	end_critical_section_2

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. task1 calls begin_critical_section_1.
2. task2 calls begin_critical_section_2.
3. task1 reaches the instruction begin_critical_section_2();. Since task2 has already called begin_critical_section_2, task1 waits for task2 to call end_critical_section_2.

4 task2 reaches the instruction begin_critical_section_1();. Since task1 has already called begin_critical_section_1, task2 waits for task1 to call end_critical_section_1.

Correction-Follow Same Locking Sequence in Both Tasks

One possible correction is to follow the same sequence of calls to lock and unlock functions in both task1 and task2.

```
void task1(void);
void task2(void);
void perform_task_cycle(void);

void begin_critical_section_1(void);
void end_critical_section_1(void);

void begin_critical_section_2(void);
void end_critical_section_2(void);

void task1() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}

void task2() {
 while(1) {
    begin_critical_section_1();
    begin_critical_section_2();
    perform_task_cycle();
    end_critical_section_2();
    end_critical_section_1();
 }
}
```

Deadlock with More Than Two Tasks

```
int var;
void performTaskCycle() {
 var++;
}

void lock1(void);
void lock2(void);
void lock3(void);


void unlock1(void);
void unlock2(void);
void unlock3(void);

void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock3();
    lock1();
    performTaskCycle();
    unlock1();
    unlock3();
 }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Value	

Entry points	`task1`
	`task2`
	`task3`	

Critical section details	**Starting procedure**	**Ending procedure**
	`lock1`	`unlock1`
	`lock2`	`unlock2`
	`lock3`	`unlock3`

A **Deadlock** occurs because the instructions can execute in the following sequence:

1. `task1` calls `lock1`.
2. `task2` calls `lock2`.
3. `task3` calls `lock3`.
4. `task1` reaches the instruction `lock2();`. Since `task2` has already called `lock2`, `task1` waits for call to `unlock2`.
5. `task2` reaches the instruction `lock3();`. Since `task3` has already called `lock3`, `task2` waits for call to `unlock3`.
6. `task3` reaches the instruction `lock1();`. Since `task1` has already called `lock1`, `task3` waits for call to `unlock1`.

Correction — Break Cyclic Order

To break the cyclic order between critical sections, note every lock function in your code in a certain sequence, for example:

1. `lock1`
2. `lock2`
3. `lock3`

If you use more than one lock function in a task, use them in the order in which they appear in the sequence. For example, you can use `lock1` followed by `lock2` but not `lock2` followed by `lock1`.

```
void performTaskCycle();
void task1() {
 while(1) {
    lock1();
    lock2();
    performTaskCycle();
    unlock2();
    unlock1();
 }
}

void task2() {
 while(1) {
    lock2();
    lock3();
    performTaskCycle();
    unlock3();
    unlock2();
 }
}

void task3() {
 while(1) {
    lock1();
    lock3();
    performTaskCycle();
    unlock3();
    unlock1();
 }
}
```

Check Information

Category: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: `deadlock`

See Also

"Find defects (C/C++)" | Data race including atomic operations | Data race | Double lock | Double unlock | Missing lock | Missing unlock

More About

- “Set Up Multitasking Analysis”
- “Review Concurrency Defects”

Dead code

Code does not execute

Description

Dead code occurs when a block of code cannot be reached via any execution path. This defect excludes:

- `Code deactivated by constant false condition`, which checks for directives such as `#if 0`.
- `Unreachable code`, which checks for code after a control escape such as goto, break, or return.
- `Useless if`, which checks for if statements that are always true.

Examples

Dead Code from `if`-Statement

```
#include <stdio.h>

int Return_From_Table(int ch){

    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    if(table[ch]>100){
         return 0;  /*Defect: Condition always false */
    }
    return table[ch];
}
```

The maximum value in the array `table` is `4^2+4+1=21`, so the test expression `table[ch]>100` always evaluates to false. The `return 0` in the `if` statement is not executed.

Correction — Remove Dead Code

One possible correction is to remove the if condition from the code.

```
#include <stdio.h>

int Return_From_Table(int ch){

    int table[5];

    /* Create a table */
    for(int i=0;i<=4;i++){
        table[i]=i^2+i+1;
    }

    return table[ch];
}
```

Dead Code for if with Enumerated Type

```
typedef enum _suit {UNKNOWN, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > 7) {
        do_something(card);
    }
}
```

The type suit is enumerated with five options. However, the conditional expression card > 7 always evaluates to false because card can be at most 5. The content in the if statement is not executed.

Correction — Change Condition

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to HEART to relate directly to the type of card.

```
typedef enum _suit {UNKNOWN, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS))
        card = UNKNOWN_SUIT;

    if (card > HEARTS) {
        do_something(card);
    }
}
```

Check Information

Category: Data-flow
Language: C | C++
Default: on
Command-Line Syntax: `dead_code`

See Also

"Find defects (C/C++)" | `Code deactivated by constant false condition` | `Unreachable code` | `Useless if`

More About

- "Data-flow Defects"
- "Review and Comment Results"

Deallocation of previously deallocated pointer

Memory freed more than once without allocation

Description

Deallocation of previously deallocated pointer occurs when a block of memory is freed more than once using the `free` function without an intermediate allocation.

Examples

Deallocation of Previously Deallocated Pointer Error

```
#include <stdlib.h>

void allocate_and_free(void)
{

    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    free (pi);
    /* Defect: pi has already been freed */
}
```

The first `free` statement releases the block of memory that `pi` refers to. The second `free` statement on `pi` releases a block of memory that has been freed already.

Correction — Remove Duplicate Deallocation

One possible correction is to remove the second `free` statement.

```
#include <stdlib.h>

void allocate_and_free(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return;

    *pi = 2;
    free(pi);
    /* Fix: remove second deallocation */
}
```

Check Information

Category: Dynamic memory
Language: C | C++
Default: on
Command-Line Syntax: `double_deallocation`

See Also

Use of previously freed pointer | "Find defects (C/C++)"

More About

- "Dynamic Memory Defects"
- "Review and Comment Results"

Declaration mismatch

Mismatch between function or variable declarations

Description

Declaration mismatch occurs when a function or variable declaration does not match other instances of the function or variable.

Examples

Inconsistent Declarations in Two Files

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
double foo(void);

int bar(void) {
    return (int)foo();
}
```

In this example, *file1.c* declares `foo()` as returning an integer. In *file2.c*,`foo()` is declared as returning a double. This difference raises a defect on the second instance of `foo` in *file2*.

Correction — Align the Function Return Values

One possible correction is to change the function declarations so that they match. In this example, by changing the declaration of `foo` in *file2.c* to match *file1.c*, the defect is fixed.

file1.c

```
int foo(void) {
    return 1;
}
```

file2.c

```
int foo(void);

int bar(void) {
    return foo();
}
```

Inconsistent Structure Alignment

test1.c

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

test2.c

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

circle.h

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;
```

square.h

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

In this example, a declaration mismatch defect is raised on `square` in *square.h* because Polyspace infers that *square.h* does not have the same alignment as `square` in *test2.c*. This error occurs because the `#pragma pack(1)` statement in *circle.h* declares specific alignment. In *test2.c*, *circle.h* is included before *square.h*. Therefore, the `#pragma pack(1)` statement from *circle.h* is not reset to the default alignment after the `aCircle` structure. Because of this omission, *test2.c* infers that the `aSquare square` structure also has an alignment of `1` byte.

Correction — Close Packing Statements

One possible correction is to reset the structure alignment after the `aCircle` struct declaration. For the GNU or Microsoft Visual compilers, fix the defect by adding a `#pragma pack()` statement at the end of *circle.h*.

test1.c

```
#include "square.h"
#include "circle.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

test2.c

```
#include "circle.h"
#include "square.h"
struct aCircle circle;
struct aSquare square;

int main(){
    square.side=1;
    circle.radius=1;
    return 0;
}
```

circle.h

```
#pragma pack(1)

extern struct aCircle{
    int radius;
} circle;

#pragma pack()
```

square.h

```
extern struct aSquare {
    unsigned int side:1;
} square;
```

Other compilers require different `#pragma pack` syntax. For your syntax, see the documentation for your compiler.

Correction — Use the `Ignore pragma pack directives` Option

One possible correction is to add the `Ignore pragma pack directives` option to your Bug Finder analysis. If you want the structure alignment to change for each structure, and you do not want to see this **Declaration mismatch** defect, use this correction.

1. On the Configuration pane, select the **Advanced Settings** pane.
2. In the **Other** box, enter `-ignore-pragma-pack`.
3. Rerun your analysis.

   The **Declaration mismatch** defect is resolved.

Check Information

Category: Programming
Language: C | C++

Default: on
Command-Line Syntax: `decl_mismatch`

See Also

"Find defects (C/C++)" | "Ignore pragma pack directives (C++)" on page 2-13

More About

- "Programming Defects"
- "Review and Comment Results"

Double lock

Lock function is called twice in a task without an intermediate call to unlock function

Description

Double lock occurs when:

- A task calls a lock function `my_lock`.
- The task calls `my_lock` again before calling the corresponding unlock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. 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. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Examples

Double Lock

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void my_task(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Value	
Entry points	`my_task`	
Critical section details	**Starting procedure**	**Ending procedure**
	`BEGIN_CRITICAL_SECTION`	`END_CRITICAL_SECTION`

`my_task` enters a critical section through the call `BEGIN_CRITICAL_SECTION();`. `my_task` calls `BEGIN_CRITICAL_SECTION` again before it leaves the critical section through the call `END_CRITICAL_SECTION();`.

Correction — Remove First Lock

If you want the first `global_var+=1;` to be outside the critical section, one possible correction is to remove the first call to `BEGIN_CRITICAL_SECTION`. However, if other tasks are using `global_var`, this code can produce a Data race error.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void my_task(void)
{
    global_var += 1;
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

Correction — Remove Second Lock

If you want the first `global_var+=1;` to be inside the critical section, one possible correction is to remove the second call to `BEGIN_CRITICAL_SECTION`.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void my_task(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

Correction — Add Another Unlock

If you want the second global_var+=1; to be inside a critical section, another possible correction is to add another call to END_CRITICAL_SECTION.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void my_task(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

Double Lock with Function Call

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void performOperation(void) {
  BEGIN_CRITICAL_SECTION();
  global_var++;
}

void my_task(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    performOperation();
    END_CRITICAL_SECTION();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Value	
Entry points	my_task	
Critical section details	**Starting procedure**	**Ending procedure**
	BEGIN_CRITICAL_SECTION	END_CRITICAL_SECTION

my_task enters a critical section through the call BEGIN_CRITICAL_SECTION();. my_task calls the function performOperation. In performOperation, BEGIN_CRITICAL_SECTION is called again even though my_task has not left the critical section through the call END_CRITICAL_SECTION();.

Correction — Remove Second Lock

One possible correction is to remove the call to BEGIN_CRITICAL_SECTION in my_task.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void performOperation(void) {
  global_var++;
}

void my_task(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    performOperation();
    END_CRITICAL_SECTION();
}
```

Check Information

Category: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: double_lock

See Also

“Find defects (C/C++)” | Data race including atomic operations | Data race | Deadlock | Double unlock | Missing lock | Missing unlock

More About

- “Set Up Multitasking Analysis”
- “Review Concurrency Defects”

Double unlock

Unlock function is called twice in a task without an intermediate call to lock function

Description

Double unlock occurs when:

- A task calls a lock function `my_lock`.
- The task calls the corresponding unlock function `my_unlock`.
- The task calls `my_unlock` again. The task does not call `my_lock` a second time between the two calls to `my_unlock`.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. 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. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Examples

Double Unlock

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void my_task(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Value	
Entry points	`my_task`	
Critical section details	**Starting procedure**	**Ending procedure**
	`BEGIN_CRITICAL_SECTION`	`END_CRITICAL_SECTION`

`my_task` enters a critical section through the call `BEGIN_CRITICAL_SECTION();`. `my_task` leaves the critical section through the call `END_CRITICAL_SECTION();`. `my_task` calls `END_CRITICAL_SECTION` again without an intermediate call to `BEGIN_CRITICAL_SECTION`.

Correction — Remove Second Unlock

If you want the second `global_var+=1;` to be outside the critical section, one possible correction is to remove the second call to `END_CRITICAL_SECTION`. However, if other tasks are using `global_var`, this code can produce a Data race error.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void my_task(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    global_var += 1;
}
```

Correction — Remove First Unlock

If you want the second `global_var+=1;` to be inside the critical section, one possible correction is to remove the first call to `END_CRITICAL_SECTION`.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);
void my_task(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

Correction — Add Another Lock

If you want the second `global_var+=1;` to be inside a critical section, another possible correction is to add another call to `BEGIN_CRITICAL_SECTION`.

```
int global_var;

void BEGIN_CRITICAL_SECTION(void);
void END_CRITICAL_SECTION(void);

void my_task(void)
{
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
    BEGIN_CRITICAL_SECTION();
    global_var += 1;
    END_CRITICAL_SECTION();
}
```

Check Information

Category: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: `double_unlock`

See Also

“Find defects (C/C++)” | Data race including atomic operations | Data race | Deadlock | Double lock | Missing lock | Missing unlock

More About

- “Set Up Multitasking Analysis”

- “Review Concurrency Defects”

Float overflow

Overflow from operation between floating points

Description

Float overflow occurs when an operation on floating point variables exceeds the space available to represent the resulting value.

The exact storage allocation for different floating point types depends on your processor. See “Target processor type (C)” on page 1-5 or “Target processor type (C++)” on page 2-2.

Examples

Multiplication of Floats

```
float square(void) {

   float val = FLT_MAX;
   return val * val;
}
```

In the return statement, the variable `val` is multiplied by itself. The square of the maximum float value cannot be represented by a float (the return type for this function) because the value of `val` is the maximum float value.

Correction — Different Storage Type

One possible correction is to store the result of the operation in a larger data type. In this example, by returning a `double` instead of a `float`, the overflow defect is fixed.

```
double square(void) {
    float val = FLT_MAX;

    return val * val;
}
```

Check Information

Category: Numerical

Language: C | C++
Default: off
Command-Line Syntax: `float_ovfl`

See Also

Integer overflow | Unsigned integer overflow | “Find defects (C/C++)”

More About

- “Numerical Defects”
- “Review and Comment Results”

Float conversion overflow

Overflow when converting between floating point data types

Description

Float conversion overflow occurs when converting a floating point number to a smaller floating point data type. If the variable does not have enough memory to represent the original number, the conversion overflows.

The exact storage allocation for different floating point types depends on your processor. See “Target processor type (C)” on page 1-5 or “Target processor type (C++)” on page 2-2.

Examples

Converting from `double` to `float`

```
float convert(void) {

    double diam = 1e100;
    return (float)diam;
}
```

In the return statement, the variable `diam` of type double (64 bits) is converted to a variable of type float (32 bits). However, the value 1^100 requires more than 32 bits to be precisely represented.

Check Information

Category: Numerical
Language: C | C++
Default: on
Command-Line Syntax: `float_conv_ovfl`

See Also

Integer conversion overflow | Unsigned integer conversion overflow | Sign change integer conversion overflow | “Find defects (C/C++)”

More About

- “Numerical Defects”
- “Review and Comment Results”

Float division by zero

Dividing floating point number by zero

Description

Float division by zero occurs when the denominator of a division operation is a zero and a floating point number.

Examples

Dividing a Floating Point Number by Zero

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

Correction — Check Before Division

```
float fraction(float num)
{
    float denom = 0.0;
    float result = 0.0;

    if( ((int)denom) != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
float fraction(float num)
{
    float denom = 2.0;
    float result = 0.0;

    result = num/denom;

    return result;
}
```

Check Information

Category: Numerical
Language: C | C++
Default: on
Command-Line Syntax: `float_zero_div`

See Also

Integer division by zero | "Find defects (C/C++)"

More About

- "Numerical Defects"
- "Review and Comment Results"

Format string specifiers and arguments mismatch

String specifiers do not match corresponding arguments

Description

Format string specifiers and arguments mismatch occurs when the parameters in the format specification do not match their corresponding arguments. For example, an argument of type `unsigned long` must have a format specification of `%lu`.

Examples

Printing a Float

```
void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", fst);
}
```

In the `printf` statement, the format specifier, `%d`, does not match the data type of `fst`.

Correction — Use an Unsigned Long Format Specifier

One possible correction is to use the `%lu` format specifier. This specifier matches the `unsigned` integer type and `long` size of `fst`.

```
void string_format(void) {

    unsigned long fst = 1;

    printf("%lu\n", fst);
}
```

Correction — Use an Integer Argument

One possible correction is to change the argument to match the format specifier. Convert `fst` to an integer to match the format specifier and print the value `1`.

```
void string_format(void) {

    unsigned long fst = 1;

    printf("%d\n", (int)fst);
}
```

Check Information

Category: Other
Language: C | C++
Default: on
Command-Line Syntax: `string_format`

See Also

Invalid use of standard library string routine | “Find defects (C/C++)”

More About

- “Other Defects”
- “Review and Comment Results”

External Web Sites

- Standard library output functions

Integer overflow

Overflow from operation between integers

Description

Integer overflow occurs when an operation on integer variables exceeds the space available to represent the resulting value.

The exact storage allocation for different integer types depends on your processor. See "Target processor type (C)" on page 1-5 or "Target processor type (C++)" on page 2-2.

Examples

Addition of Maximum Integer

```
int plusplus(void) {

    int var = INT_MAX;
    var++;
    return var;
}
```

In the third statement of this function, the variable `var` is increased by one. But the value of `var` is the maximum integer value, so an `int` cannot represent one plus the maximum integer value.

Correction — Different Storage Type

One possible correction is to change data types. Store the result of the operation in a larger data type. In this example, by returning a `long` instead of an `int`, the overflow error is fixed.

```
long plusplus(void) {

    long lvar = INT_MAX;
    lvar++;
    return lvar;
}
```

Check Information

Category: Numerical
Language: C | C++
Default: off
Command-Line Syntax: `int_ovfl`

See Also

Unsigned integer overflow | Float overflow | "Find defects (C/C++)"

More About

- "Numerical Defects"
- "Review and Comment Results"

Integer conversion overflow

Overflow when converting between integer types

Description

Integer conversion overflow occurs when converting an integer to a smaller integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different integer types depends on your processor. See "Target processor type (C)" on page 1-5 or "Target processor type (C++)" on page 2-2.

Examples

Converting from `int` to `char`

```
char convert(void) {

    int num = 1000000;

    return (char)num;
}
```

In the return statement, the integer variable `num` is converted to a `char`. However, an 8-bit or 16-bit character cannot represent 1000000 because it requires at least 20 bits. So the conversion operation overflows.

Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number.

```
long convert(void) {

    int num = 1000000;

    return (long)num;
}
```

Check Information

Category: Numerical
Language: C | C++
Default: on
Command-Line Syntax: `int_conv_ovfl`

See Also

Float conversion overflow | Unsigned integer conversion overflow | Sign change integer conversion overflow | “Find defects (C/C++)”

More About

- “Numerical Defects”
- “Review and Comment Results”

Invalid deletion of pointer

Pointer deallocation using `delete` without corresponding allocation using `new`

Description

Invalid deletion of pointer occurs when a block of memory released using the `delete` operator was not previously allocated with the `new` operator.

This defect applies only if the code language for the project is C++.

Examples

Bad Deletion Error

```
void Assign_Ones(void)
{
  int p[10];

  for(int i=0;i<10;i++)
     *(p+i)=1;

  delete[] p;
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is released using the `delete` operator. However, `p` points to a memory location that was not dynamically allocated.

Correction: Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
void Assign_Ones(void)
{
  int p[10];

  for(int i=0;i<10;i++)
```

```
      *(p+i)=1;

  /* Fix: Remove deallocation of p */
}
```

Correction — Introduce Pointer Allocation

If the number of array elements is not known at compile time, one possible correction is to dynamically allocate memory to the array `p` using the `new` operator.

```
void Assign_Ones(int num)
   {
     /* Fix: Allocate memory dynamically to p */
     int *p = new int[10];

     for(int i=0;i<10;i++)
        *(p+i)=1;

     delete[] p;
   }
```

Check Information

Category: Dynamic memory
Language: C++
Default: off
Command-Line Syntax: `bad_delete`

See Also

Invalid free of pointer | "Find defects (C/C++)"

More About

- "Dynamic Memory Defects"
- "Review and Comment Results"

Invalid use of == operator

Equality operation in assignment statement

Description

Invalid use of == operator occurs when an equality operator instead of an assignment operator is used in a simple statement. A common correction is removing one of the equal signs (=).

Examples

Equality Evaluation in `for`-Loop

```
void populate_array(void)
{
    int i = 0;
    int j = 0;
    int array[4];

    for (j == 5; j < 9; j++) {
        array[i] = j;
        i++;
    }
}
```

Inside the `for-loop`, the statement `j == 5` tests whether `j` is equal to 5 instead of setting `j` to 5. The `for-loop` iterates from 0 to 8 because `j` starts with a value of 0, not 5. A by-product of the invalid equality operator is an out-of-bounds array access in the next line.

Correction — Change to Assignment Operator

One possible correction is to change the == operator to a single equal sign (=). Changing the == sign resolves both defects because the `for-loop` iterates the intended number of times.

```
void populate_array(void)
```

```
{
    int i = 0;
    int j = 0;
    int array[4];

    for (j = 5; j < 9; j++) {
        array[i] = j;
        i++;
    }
}
```

Check Information

Category: Programming
Language: C | C++
Default: on for handwritten code, off for generated code
Command-Line Syntax: `bad_equal_equal_use`

See Also

Invalid use of = operator | “Find defects (C/C++)”

More About

- “Programming Defects”
- “Review and Comment Results”

Invalid use of = operator

Assignment in control statement

Description

Invalid use of = operator occurs when an assignment is made inside a logical statement, such as `if` or `while`. Use the equals operator as an assignment operator, not to determine equality. A common correction for this defect is adding a second equal sign (==).

Examples

Assignment in an `if`-Statement

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha = beta){
        printf("Equal\n");
    }
}
```

The equal sign is flagged as a defect because the assignment operator is used within the if-statement. Due to the single equal sign, the statement assigns the value `beta` to `alpha`, then determines the logical value of `alpha`.

Correction — Equality Operator in `if`-Statement

One possible correction is adding an additional equal sign. This correction changes the assignment operator to an equality operator. The if-statement evaluates the equality between `alpha` and `beta`.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if(alpha == beta){
        printf("Equal\n");
    }
}
```

Correction — Assignment Inside an `if`-Statement

If an assignment must be made inside a control statement, one possible correction is clarifying the control statement. This correction assigns the value of `beta` to `alpha`, and determines if `alpha` is nonzero.

```
#include <stdio.h>

void equality_test(int alpha, int beta)
{
    if((alpha = beta) != 0){
        printf("Equal\n");
    }
}
```

Check Information

Category: Programming
Language: C | C++
Default: on for handwritten code, off for generated code
Command-Line Syntax: `bad_equal_use`

See Also

Invalid use of == operator | "Find defects (C/C++)"

More About

- "Programming Defects"
- "Review and Comment Results"

Invalid use of floating point operation

Imprecise comparison of floating point variables

Description

Invalid use of floating point operation occurs when you use an equality (==) or inequality (!=) operation with floating point numbers. It is possible that the equality or inequality of two floating point values is not exact because floating point representation can be imprecise.

There are two situations when Polyspace does not flag floating point comparison: when one of the operands is `0.0` because zero can be represented exactly, and when comparing a variable against itself such as `foo == foo` or `foo != foo`.

Examples

Two Equal Floats

```
float onePointOne(void) {
    float flt = 1.0;
    if (flt == 1.1)
        return flt;
    return 0;
}
```

In this function, the if-statement tests the equality of `flt` and the number 1.1. Even though the equality in this function is obvious (1.0 is not equal to 1.1), longer floating point values are not quite so simple. Do not use equality with floating points because it can produce unexpected behavior.

Correction — Change the Operator

One possible correction is to use a different operator that is not as strict. For example, an inequality like > or <.

```
float onePointOne(void) {
    float flt = 1.0;
    if (fabs(flt-1.1) < Epilson)
        return flt;
    return 0;
}
```

Correction — Change the Operands

Another possible correction is to change the operands to more precise data types. In this example, using integers instead of floats corrects the error.

```
int onePointOne(void) {
    int flt = 1;
    if (flt == 1)
        return flt;
    return 0;
}
```

Check Information

Category: Programming
Language: C | C++
Default: off
Command-Line Syntax: `bad_float_op`

See Also

"Find defects (C/C++)"

More About

- "Programming Defects"
- "Review and Comment Results"

Invalid free of pointer

Pointer deallocation without a corresponding dynamic allocation

Description

Invalid free of pointer occurs when a block of memory released using the `free` function was not previously allocated using `malloc`, `calloc`, or `realloc`.

Examples

Invalid Free of Pointer Error

```
#include <stdlib.h>

void Assign_Ones(void)
{
  int p[10];
  for(int i=0;i<10;i++)
     *(p+i)=1;

  free(p);
  /* Defect: p does not point to dynamically allocated memory */
}
```

The pointer `p` is deallocated using the `free` function. However, `p` points to a memory location that was not dynamically allocated.

Correction — Remove Pointer Deallocation

If the number of elements of the array `p` is known at compile time, one possible correction is to remove the deallocation of the pointer `p`.

```
#include <stdlib.h>

void Assign_Ones(void)
 {
  int p[10];
  for(int i=0;i<10;i++)
```

```
     *(p+i)=1;
  /* Fix: Remove deallocation of p */
 }
```

Correction — Introduce Pointer Allocation

If the number of elements of the array p is not known at compile time, one possible correction is to dynamically allocate memory to the array p.

```
#include <stdlib.h>

void Assign_Ones(int num)
{
  int *p;
  /* Fix: Allocate memory dynamically to p */
  p=(int*) calloc(10,sizeof(int));
  for(int i=0;i<10;i++)
     *(p+i)=1;
  free(p);
}
```

Check Information

Category: Dynamic Memory
Language: C | C++
Default: on
Command-Line Syntax: bad_free

See Also

Invalid deletion of pointer | “Find defects (C/C++)”

More About

- “Dynamic Memory Defects”
- “Review and Comment Results”

Invalid use of standard library floating point routine

Wrong arguments to standard library function

Description

Invalid use of standard library floating point routine occurs when you use invalid arguments with a floating point function from the standard library. This defect picks up:

- Rounding and absolute value routines

  `ceil, fabs, floor, fmod`
- Fractions and division routines

  `fmod, modf`
- Exponents and log routines

  `frexp, ldexp, sqrt, pow, exp, log, log10`
- Trigonometry function routines

  `cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh, atanh`

Examples

Arc Cosine Operation

```
double arccosine(void) {

    double degree = 5.0;
    return acos(degree);
}
```

The input value to `acos` must be in the interval `[-1,1]`. This input argument, `degree`, is outside this range.

Correction — Change Input Argument

One possible correction is to change the input value to fit the specified range. In this example, change the input value from degrees to radians to fix this defect.

```
double arccosine(void) {

    double degree = 5.0;
    double radian = degree*180/(3.14159);
    return acos(radian);
}
```

Check Information

Category: Numerical
Language: C | C++
Default: on
Command-Line Syntax: `float_std_lib`

See Also

Invalid use of standard library integer routine | Invalid use of standard library memory routine | Invalid use of standard library string routine | Invalid use of standard library routine | "Find defects (C/C++)"

More About

- "Numerical Defects"
- "Review and Comment Results"

Invalid use of standard library integer routine

Wrong arguments to standard library function

Description

Invalid use of standard library integer routine occurs when you use invalid arguments with an integer function from the standard library. This defect picks up:

- Character Conversion

  `toupper, tolower`
- Character Checks

  `isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit`
- Integer Division

  `div, ldiv`
- Absolute Values

  `abs, labs`

Examples

Absolute Value of Large Negative

```
int absoluteValue(void) {

    int neg = INT_MIN;
    return abs(neg);
}
```

The input value to `abs` is `INT_MIN`. The absolute value of `INT_MIN` is `INT_MAX+1`. This number cannot be represented by the type `int`.

Correction — Change Input Argument

One possible correction is to change the input value to fit returned data type. In this example, change the input value to INT_MIN+1.

```
int absoluteValue(void) {

    int neg = INT_MIN+1;
    return abs(neg);
}
```

Check Information

Category: Numerical
Language: C | C++
Default: on
Command-Line Syntax: int_std_lib

See Also

Invalid use of standard library floating point routine | Invalid use of standard library memory routine | Invalid use of standard library string routine | Invalid use of standard library routine | "Find defects (C/C++)"

More About

- "Numerical Defects"
- "Review and Comment Results"

Invalid use of standard library memory routine

Standard library memory function called with invalid arguments

Description

Invalid use of standard library memory routine occurs when a memory library function is called with invalid arguments.

Examples

Invalid Use of Standard Library Memory Routine Error

```
#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);
  /* Defect: Arguments of memcpy invalid: str2 has size < 6 */

  return str2;
 }
```

The size of string `str2` is 5, but six characters of string `str1` are copied into `str2` using the `memcpy` function.

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)
 {
  /* Fix: Declare str2 with size 6 */
  char str1[10],str2[6];

  printf("Enter string:\n");
  scanf("%s",str1);

  memcpy(str2,str1,6);
  return str2;
 }
```

Check Information

Category: Static memory
Language: C | C++
Default: on
Command-Line Syntax: `mem_std_lib`

See Also

Invalid use of standard library string routine | “Find defects (C/C++)”

More About

- “Static Memory Defects”
- “Review and Comment Results”

Invalid use of standard library string routine

Standard library string function called with invalid arguments

Description

Invalid use of standard library string routine occurs when a string library function is called with invalid arguments.

Examples

Invalid Use of Standard Library String Routine Error

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

 char* Copy_String(void)
 {
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";

  res=strcpy(gbuffer,text);
  /* Error: Size of text is less than gbuffer */

  return(res);
 }
```

The string `text` is larger in size than `gbuffer`. Therefore, the function `strcpy` cannot copy `text` into `gbuffer`.

Correction — Use Valid Arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

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

 char* Copy_String(void)
```

```
{
 char *res;
 /*Fix: gbuffer has equal or larger size than text */
 char gbuffer[20],text[20]="ABCDEFGHIJKL";

 res=strcpy(gbuffer,text);

 return(res);
}
```

Check Information

Category: Static memory
Language: C | C++
Default: on
Command-Line Syntax: `str_std_lib`

See Also

Invalid use of standard library memory routine | “Find defects (C/C++)”

More About

- “Static Memory Defects”
- “Review and Comment Results”

Integer division by zero

Dividing integer number by zero

Description

Integer division by zero occurs when the denominator of a division operation is a zero.

Examples

Dividing an Integer by Zero

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    result = num/denom;

    return result;
}
```

A division by zero error occurs at `num/denom` because `denom` is zero.

Correction — Check Before Division

```
int fraction(int num)
{
    int denom = 0;
    int result = 0;

    if (denom != 0)
        result = num/denom;

    return result;
}
```

Before dividing, add a test to see if the denominator is zero, checking before division occurs. If `denom` is always zero, this correction can produce a dead code defect in your Polyspace results.

Correction — Change Denominator

One possible correction is to change the denominator value so that `denom` is not zero.

```
int fraction(int num)
{
    int denom = 2
    int result = 0;

    result = num/denom;

    return result;
}
```

Check Information

Category: Numerical
Language: C | C++
Default: on
Command-Line Syntax: `int_zero_div`

See Also

Integer division by zero | Float division by zero | “Find defects (C/C++)”

More About

- “Numerical Defects”
- “Review and Comment Results”

Large pass-by-value argument

Large argument passed between functions by value

Description

Large pass-by-value argument occurs when a large input argument or return value is passed between functions by its value. For variables larger than 64 bytes, pass the value by pointer or by reference to save stack space and copy time.

Examples

Passing a Large `struct` Between Functions

```
typedef struct s_userid {
    char name[2];
    int idnumber[100];
} userid;

char username(userid first) {
    return first.name[0];
}
```

The large structure, `userid`, is passed to the function `username`. Because `userid` is larger than 64 bytes, this function produces a large pass-by-value defect.

Correction — Pass-By-Reference

One possible correction is to pass the argument by reference instead of by value. In this example, the pointer to a `userid` structure is passed instead of the actual structure.

```
typedef struct s_userid {
    char name[2];
    int idnumber[100];
} userid;

char username(userid *first) {
    return (*first).name[0];
}
```

Check Information

Category: Other
Language: C | C++
Default: off
Command-Line Syntax: `pass_by_value`

See Also

"Find defects (C/C++)"

More About

- "Other Defects"
- "Review and Comment Results"

Line with more than one statement

Multiple statements on a line

Description

Before preprocessing starts, **Line with more than one statement** checks for additional text after the semicolon (`;`) on a line. A defect is not raised for comments, `for`-loop definitions, braces, or backslashes.

Examples

Single-Line Initialization

```
int multi_init(void){
_   int abc = 4; int efg = 0; //defect

    return abc*efg;
}
```

In this example, `abc` and `efg` are initialized on the second line of the function as separate statements.

Correction — Comma-Separated Initialization

One possible correction is to use a comma instead of a semicolon to declare multiple variables on the same line.

```
int multi_init(void){
    int a = 4, b = 0;

    return a*b;
}
```

Correction — New Line for Each Initialization

One possible correction is to separate each initialization. By putting the initialization of `b` on the next line, the code longer raises a defect.

```
int multi_init(void){
    int a = 4;
    int b = 0;

    return a*b;
}
```

Single-Line Loops

```
int multi_loop(void){
    int a, b = 0;
    int index = 1;
    int tab[9] = {1,1,2,3,5,8,13,21};

    for(a=0; a < 3; a++) {b+=a;} // no defect

_   for(b=0; b < 3; b++) {a+=b; index=b;} //defect

_   while (index < 7) {index++; tab[index] = index * index;} //defect
    return a*b;
}
```

In this example, there are three loops coded on single lines, each with multiple semicolons.

- The first `for` loop has multiple semicolons. Polyspace does not raise a defect for multiple statements within a `for` loop declaration.
- Polyspace does raise a defect on the second `for` loop because there are multiple statements after the `for` loop declaration.
- The `while` loop also has multiple statements after the loop declaration. Polyspace raises a defect on this line.

Correction — New Line for Each Loop Statement

One possible correction is to use a new line for each statement after the loop declaration.

```
int multi_loop(void){
    int a, b = 0;
    int index = 1;
    int tab[9] = {1,1,2,3,5,8,13,21};

    for(a=0; a < 3; a++) {b+=a;}
    for(b=0; b < 3; b++){
      a+=b;
      index=b;
    }

    while (index < 7){
      index++;
      tab[index] = index * index;
    }
    return a*b;
}
```

Single-line Conditionals

```
int multi_if(void){

    int a, b = 1;
    if(a == 0) { a++;} // no defect
_    else if(b == 1) {b++; a *= b;} //defect
}
```

In this example, there are two conditional statements an: `if` and an `else if`. The `if` line does not raise a defect because only one statement follows the condition. The `else if` statement does raise a defect because two statements follow the condition.

Correction — New Lines for Multi-Statement Conditionals

One possible correction is to use a new line for conditions with multiple statements.

```
int multi_if(void){
    int a, b = 1;

    if(a == 0) a++;
    else if(b == 1){
      b++;
      a *= b;
    }
}
```

Check Information

Category: Other
Language: C | C++

Default: off
Command-Line Syntax: `more_than_one_statement`

See Also

"Find defects (C/C++)"

More About

- "Other Defects"
- "Review and Comment Results"

Memory leak

Memory allocated dynamically not freed

Description

Memory leak occurs when you do not free a block of memory allocated through `malloc`, `calloc`, or `realloc`. If the memory is allocated in a function `func`, the defect does not occur if:

- Within `func`, you free the memory using the `free` function.
- `func` returns the pointer assigned by `malloc`, `calloc`, or `realloc`.

Examples

Memory Leak Error

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

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }


    *pi = 42;
    /* Defect: pi is not freed */
  }
```

In this example, `pi` is dynamically allocated by `malloc`. The function `assign_memory` does not free the memory, nor does it return `pi`.

Correction — Free Memory

One possible correction is to free the memory referenced by `pi` using the `free` function. The `free` function must be called before the function `assign_memory` terminates

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

void assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return;
        }
    *pi = 42;

    /* Fix: Free the pointer pi*/
    free(pi);
}
```

Correction — Return Pointer from Dynamic Allocation

Another possible correction is to return the pointer `pi`. Returning `pi` allows the function calling `assign_memory` to free the memory block using `pi`.

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

int* assign_memory(void)
{
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL)
        {
         printf("Memory allocation failed");
         return(pi);
        }
    *pi = 42;

    /* Fix: Return the pointer pi*/
    return(pi);
}
```

Check Information

Category: Dynamic memory
Language: C | C++
Default: off
Command-Line Syntax: `mem_leak`

See Also

"Find defects (C/C++)"

More About

- "Dynamic Memory Defects"
- "Review and Comment Results"

Missing lock

Unlock function without lock function

Description

Missing lock occurs when a task calls an unlock function before calling the corresponding lock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. When a task `my_task` calls a lock function `my_lock`, other tasks calling `my_lock` must wait till `my_task` calls the corresponding unlock function. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Examples

Missing lock

```
void begin_critical_section(void);
void end_critical_section(void);

void my_task(void)
{
    global_var += 1;
    end_critical_section();
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Value	
Entry points	`my_task`	
Critical section details	**Starting procedure**	**Ending procedure**
	`begin_critical_section`	`end_critical_section`

my_task calls end_critical_section before calling begin_critical_section.

Correction — Provide Lock

One possible correction is to call the lock function begin_critical_section before the instructions in the critical section.

```
void begin_critical_section(void);
void end_critical_section(void);

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
    end_critical_section();
}
```

Lock in Condition

```
int global_var;
void reset() {
    global_var=0;
}

void begin_critical_section(void);
void end_critical_section(void);

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      if(index%10==0) {
        begin_critical_section();
        global_var ++;
        reset();
      }
      end_critical_section();
      index++;
    }
}
```

In this example, to emulate multitasking behavior, you must specify the following options:

Option	Value	
Entry points	`my_task`	
Critical section details	**Starting procedure**	**Ending procedure**
	`begin_critical_section`	`end_critical_section`

In the `while` loop, `my_task` leaves a critical section through the call `end_critical_section();`. In an iteration of the `while` loop:

- If `my_task` enters the `if` condition branch, the critical section begins through a call to `begin_critical_section`.
- If `my_task` does not enter the `if` condition branch and leaves the `while` loop, the critical section does not begin. Therefore, a **Missing lock** defect occurs.
- If `my_task` does not enter the `if` condition branch and continues to the next iteration of the `while` loop, the unlock function `end_critical_section` is called again. A **Double unlock** defect occurs.

Because `numCycles` is a `volatile` variable, it can take any value. Any of the cases above are possible. Therefore, a **Missing lock** defect and a **Double unlock** defect appear on the call `end_critical_section`.

Check Information

Category: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: `bad_unlock`

See Also

"Find defects (C/C++)" | Data race including atomic operations | Data race | Deadlock | Double lock | Double unlock | Missing unlock

More About

- "Set Up Multitasking Analysis"
- "Review Concurrency Defects"

Missing unlock

Lock function without unlock function

Description

Missing unlock occurs when:

- A task calls a lock function.
- The task ends without a call to an unlock function.

In multitasking code, a lock function begins a critical section of code and an unlock function ends it. 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. Polyspace requires that both lock and unlock functions must have the form `void func(void)`.

To find this defect, before analysis, you must specify the multitasking options. On the **Configuration** pane, select **Multitasking**.

Examples

Missing Unlock

```
void begin_critical_section(void);
void end_critical_section(void);

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
}
```

In this example, to emulate multitasking behavior, specify the following options:

Option	Value
Entry points	`my_task`

Option	Value	
Critical section details	**Starting procedure**	**Ending procedure**
	`begin_ critical_section`	`end_ critical_section`

`my_task` enters a critical section through the call `begin_critical_section();`. `my_task` ends without calling `end_critical_section`.

Correction — Provide Unlock

One possible correction is to call the unlock function `end_critical_section` after the instructions in the critical section.

```
void begin_critical_section(void);
void end_critical_section(void);

void my_task(void)
{
    begin_critical_section();
    global_var += 1;
    end_critical_section();
}
```

Unlock in Condition

```
int global_var;
void reset() {
    global_var=0;
}

void begin_critical_section(void);
void end_critical_section(void);

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        reset();
        end_critical_section();
      }
      index++;
    }
}
```

In this example, to emulate multitasking behavior, specify the following options.

Option	Value	
Entry points	`my_task`	
Critical section details	**Starting procedure**	**Ending procedure**
	`begin_ critical_section`	`end_ critical_section`

In the `while` loop, `my_task` enters a critical section through the call `begin_critical_section();`. In an iteration of the `while` loop:

- If `my_task` enters the `if` condition branch, the critical section ends through a call to `end_critical_section`.
- If `my_task` does not enter the `if` condition branch and leaves the `while` loop, the critical section does not end. Therefore, a **Missing unlock** defect occurs.
- If `my_task` does not enter the `if` condition branch and continues to the next iteration of the `while` loop, the lock function `begin_critical_section` is called again. A **Double lock** defect occurs.

Because `numCycles` is a `volatile` variable, it can take any value. Any of the cases above is possible. Therefore, a **Missing unlock** defect and a **Double lock** defect appear on the call `begin_critical_section`.

Correction — Place Unlock Outside Condition

One possible correction is to call the unlock function `end_critical_section` outside the `if` condition.

```
int global_var;
void reset() {
    global_var=0;
}

void begin_critical_section(void);
void end_critical_section(void);
void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        reset();
      }
      end_critical_section();
      index++;
    }
}
```

Correction — Place Unlock in Every Conditional Branch

Another possible correction is to call the unlock function `end_critical_section` in every branches of the `if` condition.

```
int global_var;
void reset() {
    global_var=0;
}

void begin_critical_section(void);
void end_critical_section(void);

void my_task(void) {
    int index=0;
    volatile int numCycles;

    while(numCycles) {
      begin_critical_section();
      global_var ++;
      if(index%10==0) {
        reset();
        end_critical_section();
      }
      else
        end_critical_section();
      index++;
    }
}
```

Check Information

Category: Concurrency
Language: C | C++
Default: On
Command-Line Syntax: `bad_lock`

See Also

“Find defects (C/C++)” | Data race including atomic operations | Data race | Deadlock | Double lock | Double unlock | Missing lock

More About

- “Set Up Multitasking Analysis”
- “Review Concurrency Defects”

Missing null in string array

String does not terminate with null character

Description

Missing null in string array occurs when a string does not have enough space to terminate with a null character '\0'. This defect can cause various memory errors in your code, so is important to fix it.

This defect applies only for projects in C.

Examples

Array size is too small

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[5] = "THREE";
}
```

The character array three has a size of 5 and 5 characters 'T', 'H', 'R', 'E', and 'E'. There is no room for the null character at the end because three is only five bytes large.

Correction — Increase Array Size

One possible correction is to change the array size to allow for the five characters plus a null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[6] = "THREE";
}
```

Correction — Change Initialization Method

One possible correction is to initialize the string by leaving the array size blank. This initialization method allocates enough memory for the five characters and a terminating-null character.

```
void countdown(int i)
{
    static char one[5]   = "ONE";
    static char two[5]   = "TWO";
    static char three[]  = "THREE";
}
```

Check Information

Category: Programming
Language: C | C++
Default: on for handwritten code, off for generated code
Command-Line Syntax: `missing_null_char`

See Also

"Find defects (C/C++)"

More About

- "Programming Defects"
- "Review and Comment Results"

Missing or invalid return statement

Function does not return value though return type is not `void`

Description

Missing or invalid return statement occurs when a function does not return a value along at least one execution path. If the return type of the function is `void`, this error does not occur.

Examples

Missing or invalid return statement error

```
int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }
 }
/* Defect: No return value if n is not 0*/
```

If `n` is equal to 0, the code does not enter the `if` statement. Therefore, the function `AddSquares` does not return a value if `n` is 0.

Correction — Place Return Statement on Every Execution Paths

One possible correction is to return a value in every branch of the `if...else` statement.

```
 int AddSquares(int n)
 {
   int i=0;
   int sum=0;

   if(n!=0)
    {
     for(i=1;i<=n;i++)
        {
         sum+=i^2;
        }
     return(sum);
    }

   /*Fix: Place a return statement on branches of if-else */
   else
     return 0;
  }
```

Check Information

Category: Data-flow
Language: C | C++
Default: on
Command-Line Syntax: `missing_return`

See Also

"Find defects (C/C++)"

More About

- "Data-flow Defects"
- "Review and Comment Results"

Non-initialized pointer

Pointer not initialized before dereference

Description

Non-initialized pointer occurs when a pointer is not assigned an address before dereference.

Examples

Non-initialized pointer error

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
      {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
      }

    *pi = j;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```

If `prev` is not `NULL`, the pointer `pi` is not assigned an address. However, `pi` is dereferenced on every execution paths, irrespective of whether `prev` is `NULL` or not.

Correction — Initialize Pointer on Every Execution Path

One possible correction is to assign an address to `pi` when `prev` is not `NULL`.

```
#include <stdlib.h>

int* assign_pointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL)
       {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
       }
    /* Fix: Initialize pi in branches of if statement  */
    else
        pi = prev;


    *pi = j;

    return pi;
}
```

Check Information

Category: Data-flow
Language: C | C++
Default: on
Command-Line Syntax: `non_init_ptr`

See Also

Non-initialized variable | "Find defects (C/C++)"

More About

- "Data-flow Defects"
- "Review and Comment Results"

Non-initialized variable

Variable not initialized before use

Description

Non-initialized variable occurs when a variable is not initialized before its value is read.

Examples

Non-initialized variable error

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    int val;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
    /* Defect: val does not have a value if command is not 2 */
}
```

If `command` is not 2, the variable `val` is unassigned. In this case, the return value of function `get_sensor_value` is undetermined.

Correction — Initialize During Declaration

One possible correction is to initialize `val` during declaration so that only its value is dependant on different execution paths.

```
int get_sensor_value(void)
{
    extern int getsensor(void);
    int command;
    /* Fix: Initialize val */
    int val=0;

    command = getsensor();
    if (command == 2)
      {
        val = getsensor();
      }

    return val;
 }
```

`val` is assigned an initial value of 0. When `command` is not equal to 2, the function `get_sensor_value` returns this value.

Check Information

Category: Data-flow
Language: C | C++
Default: on
Command-Line Syntax: `non_init_var`

See Also

Non-initialized pointer | “Find defects (C/C++)”

More About

- “Data-flow Defects”
- “Review and Comment Results”

Null pointer

NULL pointer dereferenced

Description

Null pointer occurs when you use a pointer with a value of NULL as if it points to a valid memory location.

Examples

Null pointer error

```
#include <stdlib.h>

int FindMax(int *arr, int Size)
{
 int* p=NULL;

 *p=arr[0];
 /* Defect: Null pointer dereference */

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

The pointer p is initialized with value of NULL. However, when the value arr[0] is written to *p, p is assumed to point to a valid memory location.

Correction — Assign Address to Null Pointer Before Dereference

One possible correction is to initialize p with a valid memory address before dereference.

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

```
int FindMax(int *arr, int Size)
{
 /* Fix: Assign address to null pointer */
 int* p=&arr[0];

 for(int i=0;i<Size;i++)
  {
   if(arr[i] > (*p))
     *p=arr[i];
  }

 return *p;
}
```

Check Information

Category: Static memory
Language: C | C++
Default: on
Command-Line Syntax: `null_ptr`

See Also

Arithmetic operation with NULL pointer | Non-initialized pointer | "Find defects (C/C++)"

More About

- "Static Memory Defects"
- "Review and Comment Results"

Invalid use of standard library routine

Wrong arguments to standard library function

Description

Invalid use of standard library routine occurs when you use invalid arguments with a function from the standard library. This defect picks up errors related to other functions not covered by float, integer, memory, or string standard library routines.

Examples

Calling `printf` Without a String

```
void print_null(void) {

  printf(NULL);
}
```

The function `printf` takes only string input arguments or format specifiers. In this function, the input value is NULL, which is not a valid string.

Correction — Use Compatible Input Arguments

One possible correction is to change the input arguments to fit the requirements of the standard library routine. In this example, the input argument was changed to a character.

```
void print_null(void) {
    char zero_val = '0';
    printf(zero_val);
}
```

Check Information

Category: Other
Language: C | C++
Default: on

Command-Line Syntax: `other_std_lib`

See Also

Invalid use of standard library integer routine | Invalid use of standard library floating point routine | Invalid use of standard library memory routine | Invalid use of standard library string routine | “Find defects (C/C++)”

More About

- “Other Defects”
- “Review and Comment Results”

Pointer access out of bounds

Pointer dereferenced outside its bounds

Description

Pointer access out of bounds occurs when a pointer is dereferenced outside its bounds.

When a pointer is assigned an address, a block of memory is associated with the pointer. You cannot access memory beyond that block using the pointer.

Examples

Pointer access out of bounds error

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
   {
    ptr++;
    *ptr=i;
    /* Defect: ptr out of bounds for i=9 */
   }

 return(arr);
}
```

`ptr` is assigned the address `arr` that points to a memory block of size `10*sizeof(int)`. In the `for`-loop, `ptr` is incremented 10 times. In the last iteration of the loop, `ptr` points outside the memory block assigned to it. Therefore, it cannot be dereferenced.

Correction — Check Pointer Stays Within Bounds

One possible correction is to reverse the order of increment and dereference of `ptr`.

```
int* Initialize(void)
{
 int arr[10];
 int *ptr=arr;

 for (int i=0; i<=9;i++)
     {
      /* Fix: Dereference pointer before increment */
      *ptr=i;
      ptr++;
     }

 return(arr);
}
```

After the last increment, even though `ptr` points outside the memory block assigned to it, it is not dereferenced more.

Check Information

Category: Static memory
Language: C | C++
Default: on
Command-Line Syntax: `out_bound_ptr`

See Also

Array access out of bounds | “Find defects (C/C++)”

More About

- “Static Memory Defects”
- “Review and Comment Results”

Pointer to non-initialized value converted to const pointer

Pointer to constant assigned address that does not contain a value

Description

Pointer to non initialized value converted to const pointer occurs when a pointer to a constant is assigned an address that does not yet contain a value.

Examples

Pointer to non initialized value converted to const pointer error

```
#include<stdio.h>

void Display_Parity()
 {
  int num,parity;
  const int* num_ptr = &num;
  /* Defect: Address &num does not store a value */

  printf("Enter a number\n:");
  scanf("%d",&num);

  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");

 }
```

`num_ptr` is declared as a pointer to a constant. However the variable `num` does not contain a value when `num_ptr` is assigned the address `&num`.

Correction — Store Value in Address Before Assignment to Pointer

One possible correction is to obtain the value of `num` from the user before `&num` is assigned to `num_ptr`.

```
#include<stdio.h>

void Display_Parity()
 {
  int num,parity;
  const int* num_ptr;

  printf("Enter a number\n:");
  scanf("%d",&num);

 /* Fix: Assign &num to pointer after it receives a value */
  num_ptr=&num;
  parity=((*num_ptr)%2);
  if(parity==0)
    printf("The number is even.");
  else
    printf("The number is odd.");
 }
```

The `scanf` statement stores a value in `&num`. Once the value is stored, it is legitimate to assign `&num` to `num_ptr`.

Check Information

Category: Data-flow
Language: C | C++
Default: off
Command-Line Syntax: `non_init_ptr_conv`

See Also

"Find defects (C/C++)"

More About

- "Data-flow Defects"

- "Review and Comment Results"

Partially accessed array

Array partly read or written before end of scope

Description

Partially accessed array occurs when an array is partially read or written before the end of array scope. For arrays local to a function, the end of scope occurs when the function ends.

Examples

Partially accessed array error

```
int Calc_Sum(void)
{
  int tab[5]={0,1,2,3,4},sum=0;
  /* Defect: tab[4] is not read */

  for (int i=0; i<4;i++) sum+=tab[i];

  return(sum);

 }
```

The array `tab` is only partially read before end of function `Calc_Sum`. While calculating `sum`, `tab[4]` is not included.

Correction — Access Every Array Element

One possible correction is to read every element in the array `tab`.

```
int Calc_Sum(void)
{
  int tab[5]={0,1,2,3,4},sum=0;

  /* Fix: Include tab[4] in calculating sum */
  for (int i=0; i<5;i++) sum+=tab[i];

  return(sum);

 }
```

Check Information

Category: Data-flow
Language: C | C++
Default: on for handwritten code, off for generated code
Command-Line Syntax: `partially_accessed_array`

See Also

"Find defects (C/C++)"

More About

- "Data-flow Defects"
- "Review and Comment Results"

Qualifier removed in conversion

Variable qualifier is lost during conversion

Description

Qualifier removed in conversion occurs during a conversion when one variable has a qualifier and the other does not. For example, when converting from a `const int` to an `int`, the conversion removes the `const` qualifier.

This defect applies only for projects in C.

Examples

Cast of Character Pointers

```
void implicit_cast(void) {
    const char  cc, *pcc = &cc;
    char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

During the assignment to the character `q`, the variables, `cc` and `pcc`, are converted from `const char` to `char`. The `const` qualifier is removed during the conversion causing a defect.

Correction — Add Qualifiers

One possible correction is to add the same qualifiers to the new variables. In this example, changing `q` to a `const char` fixes the defect.

```
void implicit_cast(void) {
    const char  cc, *pcc = &cc;
    const char * quo;
    quo = &cc;
    quo = pcc;

    read(quo);
}
```

Correction — Remove Qualifiers

One possible correction is to remove the qualifiers in the converted variable. In this example, removing the `const` qualifier from the `cc` and `pcc` initialization fixes the defect.

```
void implicit_basic_cast(void) {
    char  cc, *pcc = &cc;
    char * quo;

    quo = &cc;
    quo = pcc;

    read(quo);
}
```

Check Information

Category: Programming
Language: C | C++
Default: off
Command-Line Syntax: `qualifier_mismatch`

See Also

"Find defects (C/C++)"

More About

- "Programming Defects"
- "Review and Comment Results"

Shift of a negative value

Shift operator on negative value

Description

Shift of a negative value occurs when a bit-wise shift is used on a negative number. Shifts can overwrite the sign bit that identifies a number as negative.

Examples

Shifting a negative variable

```
int shifting(int val)
{
    int res = -1;
    return res << val;
}
```

In the return statement, the variable `res` is shifted a certain number of bits to the left. However, because `res` is negative, the shift might overwrite the sign bit.

Correction — Change the Data Type

One possible correction is to change the data type of the shifted variable to unsigned. This correction eliminates the sign bit, so left shifting does not change the sign of the variable.

```
int shifting(int val)
{
    unsigned int res = -1;
    return res << val
}
```

Check Information

Category: Numerical
Language: C | C++

Default: off
Command-Line Syntax: `shift_neg`

See Also

Shift operation overflow | “Find defects (C/C++)”

More About

- “Numerical Defects”
- “Review and Comment Results”

Shift operation overflow

Overflow from shifting operation

Description

Shift operation overflow occurs when a shift operation exceeds the space available to represent the resulting value.

The exact storage allocation for different data types depends on your processor. See “Target processor type (C)” on page 1-5 or “Target processor type (C++)” on page 2-2.

Examples

Left Shift of Integer

```
int left_shift(void) {

    int foo = 33;
    return 1 << foo;
}
```

In the return statement of this function, bit-wise shift operation is performed shifting 1 `foo` bits to the left. However, an `int` has only 32 bits, so the range of the shift must be between 0 and 31. Therefore, this shift operation causes an overflow.

Correction — Different storage type

One possible correction is to store the shift operation result in a larger data type. In this example, by returning a `long` instead of an `int`, the overflow defect is fixed.

```
long left_shift(void) {

    int foo = 33;
    return 1 << foo;
}
```

Check Information

Category: Numerical

Language: C | C++
Default: off
Command-Line Syntax: `shift_ovfl`

See Also

"Find defects (C/C++)"

More About

- "Numerical Defects"
- "Review and Comment Results"

Sign change integer conversion overflow

Overflow when converting between signed and unsigned integers

Description

Sign change integer conversion overflow occurs when converting an unsigned integer to a signed integer. If the variable does not have enough bytes to represent both the original constant and the sign bit, the conversion overflows.

The exact storage allocation for different integer types depends on your processor. See “Target processor type (C)” on page 1-5 or “Target processor type (C++)” on page 2-2.

Examples

Convert from `unsigned char` to `char`

```
char sign_change(void) {
    unsigned char count = 255;

    return (char)count;
}
```

In the return statement, the unsigned character variable `count` is converted to a signed character. However, `char` has 8 bits, 1 for the sign of the constant and 7 to represent the number. The conversion operation overflows because 255 uses 8 bits.

Correction — Change conversion types

One possible correction is using a larger integer type. By using an `int`, there are enough bits to represent the sign and the number value.

```
int sign_change(void) {
    unsigned char count = 255;

    return (int)count;
}
```

Check Information

Category: Numerical
Language: C | C++
Default: on
Command-Line Syntax: `sign_change`

See Also

Float conversion overflow | Unsigned integer conversion overflow | Integer conversion overflow | “Find defects (C/C++)”

More About

- “Numerical Defects”
- “Review and Comment Results”

Uncalled function

Function with static scope not called in file

Description

Uncalled function occurs when a `static` function is not called in the same file where it is defined.

Examples

Uncalled function error

Save the following code in the file `Initialize_Value.c`

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

static int Initialize(void)
/* Defect: Function not called */
  {
   int input;
   printf("Enter an integer:");
   scanf("%d",&input);
   return(input);
  }

 void main()
  {
   int num;

   num=0;

   printf("The value of num is %d",num);
  }
```

The `static` function `Initialize` is not called in the file `Initialize_Value.c`.

Correction — Call Function at Least Once

One possible correction is to call `Initialize` at least once in the file `Initialize_Value.c`.

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

static int Initialize(void)
  {
   int input;
   printf("Enter an integer:");
   scanf("%d",&input);
   return(input);
  }

 void main()
  {
   int num;

   /* Fix: Call static function Initialize */
   num=Initialize();

   printf("The value of num is %d",num);
  }
```

Check Information

Category: Data-flow
Language: C | C++
Default: off
Command-Line Syntax: `uncalled_func`

See Also

"Find defects (C/C++)"

More About

- "Data-flow Defects"
- "Review and Comment Results"

Unprotected dynamic memory allocation

Pointer returned from dynamic allocation not checked for NULL value

Description

Unprotected dynamic memory allocation occurs when the code does not check whether or not the dynamic memory allocation succeeded.

When memory is dynamically allocated using malloc, calloc, or realloc, it returns a value NULL if the requested memory is not available. If the code following the allocation accesses the memory block without checking for the NULL value, this access is not protected from failures.

Examples

Unprotected dynamic memory allocation error

```
#include <stdlib.h>

void Assign_Value(void)
{
  int* p = (int*)calloc(5, sizeof(int));

  *p = 2;
  /* Defect: p is not checked for NULL value */

  free(p);
}
```

If the memory allocation fails, the function calloc returns NULL to p. Before accessing the memory through p, the code does not check whether p is NULL

Correction — Check for NULL Value

One possible correction is to check whether p has value NULL before dereference.

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

```
void Assign_Value(void)
 {
   int* p = (int*)calloc(5, sizeof(int));

   /* Fix: Check if p is NULL */
   if(p!=NULL) *p = 2;

   free(p);
 }
```

Check Information

Category: Dynamic memory
Language: C | C++
Default: off
Command-Line Syntax: `unprotected_memory_allocation`

See Also

"Find defects (C/C++)"

More About

- "Dynamic Memory Defects"
- "Review and Comment Results"

Unreachable code

Code following control-flow statements

Description

Unreachable code defects occur on code which follow statements that change the flow of the code, for example `break`, `goto`, or `return`. These statements move the flow of the program to another section or function. Because of this flow escape, the statements following the control-flow code, statistically, do not execute, and therefore the statements are not reachable.

Examples

Unreachable code after return

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }
    return card;

    if (card < HEARTS) {
        guess(card);
    }
    return card;
}
```

In this example, the first return statement changes the flow of code back to where the function was called. Because of this return statement, the `if`-block and `return` statement do not execute.

Correction — Remove return

One possible correction is to remove the escape statement. In this example, remove the first return statement to reach the final if statement.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card < HEARTS) {
        guess(card);
    }
    return card;
}
```

Correction — Remove unreachable code

Another possible correction is to remove the unnecessary code. Because the function does not reach the second `if`-statement, removing it simplifies the code.

```
typedef enum _suit {UNKNOWN_SUIT, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void guess(suit s);

suit deal(void){
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }
    return card;
}
```

Check Information

Category: Data-flow
Language: C | C++
Default: on

Command-Line Syntax: `unreachable`

See Also

"Find defects (C/C++)" | `Code deactivated by constant false condition` | `Dead code` | `Useless if`

More About

- "Data-flow Defects"
- "Review and Comment Results"

Unreliable cast of function pointer

Function pointer cast to another function pointer with different argument or return type

Description

Unreliable cast of function pointer occurs when a function pointer is cast to another function pointer that has different argument or return type.

This defect applies only if the code language for the project is C.

Examples

Unreliable cast of function pointer error

```
#include <math.h>
#include <stdio.h>
#define PI 3.142

double Calculate_Sum(int (*fptr)(double))
{
    double  sum = 0.0;
    double  y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;

    fp = sin;
    sum = Calculate_Sum(fp);
    /* Defect: fp implicitly cast to int(*) (double) */

    printf("sum(sin): %f\n", sum);
    return 0;
}
```

The function pointer `fp` is declared as `double (*)(double)`. However in passing it to function `Calculate_Sum`, `fp` is implicitly cast to `int (*)(double)`.

Correction — Avoid Function Pointer Cast

One possible correction is to check that the function pointer in the definition of `Calculate_Sum` has the same argument and return type as `fp`. This step makes sure that `fp` is not implicitly cast to a different argument or return type.

```
#include <math.h>
#include <stdio.h>
# define PI 3.142

/*Fix: fptr has same argument and return type everywhere*/
double Calculate_Sum(double (*fptr)(double))
{
    double  sum = 0.0;
    double y;

    for (int i = 0;  i <= 100;  i++)
    {
        y = (*fptr)(i*PI/100);
        sum += y;
    }
    return sum / 100;
}

int main(void)
{
    double  (*fp)(double);
    double  sum;


    fp = sin;
    sum = Calculate_Sum(fp);
    printf("sum(sin): %f\n", sum);

    return 0;
}
```

Check Information

Category: Static memory
Language: C | C++
Default: on
Command-Line Syntax: `func_cast`

See Also

"Find defects (C/C++)"

More About

- "Static Memory Defects"
- "Review and Comment Results"

Unreliable cast of pointer

Pointer implicitly cast to different data type

Description

Unreliable cast of pointer occurs when a pointer is implicitly cast to a data type different from its declaration type. Such an implicit casting can take place, for instance, when a pointer to data type `char` is assigned the address of an integer.

This defect applies only if the code language for the project is C.

Examples

Unreliable cast of pointer error

```
#include <string.h>

void Copy_Integer_To_String()
{
 int src[]={1,2,3,4,5,6,7,8,9,10};
 char buffer[]="Buffer_Text";
 strcpy(buffer,src);
 /* Defect: Implicit cast of (int*) to (char*) */
}
```

`src` is declared as an `int*` pointer. The `strcpy` statement, while copying to `buffer`, implicitly casts `src` to `char*`.

Correction — Avoid Pointer Cast

One possible correction is to declare the pointer `src` with the same data type as `buffer`.

```
#include <string.h>
 void Copy_Integer_To_String()
{
 /* Fix: Declare src with same type as buffer */
 char *src[10]={"1","2","3","4","5","6","7","8","9","10"};
 char *buffer[10];
for(int i=0;i<10;i++)
  buffer[i]="Buffer_Text";

for(int i=0;i<10;i++)
  buffer[i]= src[i];
}
```

Check Information

Category: Static memory
Language: C | C++
Default: on
Command-Line Syntax: `ptr_cast`

See Also

Unreliable cast of function pointer | “Find defects (C/C++)”

More About

- “Static Memory Defects”
- “Review and Comment Results”

Unsigned integer conversion overflow

Overflow when converting between unsigned integer types

Description

Unsigned integer conversion overflow occurs when converting an unsigned integer to a smaller unsigned integer type. If the variable does not have enough bytes to represent the original constant, the conversion overflows.

The exact storage allocation for different integer types depends on your processor. See “Target processor type (C)” on page 1-5 or “Target processor type (C++)” on page 2-2.

Examples

Converting from `int` to `char`

```
unsigned char convert(void) {
    unsigned int unum = 1000000U;

    return (unsigned char)unum;
}
```

In the return statement, the unsigned integer variable `unum` is converted to an unsigned character type. However, the conversion overflows because 1000000 requires at least 20 bits. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `unum` is reduced by modulo `2^8` because a character data type can only represent `2^8-1`.

Correction — Change Conversion Type

One possible correction is to convert to a different integer type that can represent the entire number. For example, `long`.

```
unsigned long convert(void) {
    unsigned int unum = 1000000U;
    return (unsigned long)unum;
}
```

Check Information

Category: Numerical
Language: C | C++
Default: on
Command-Line Syntax: `uint_conv_ovfl`

See Also

Float conversion overflow | Integer conversion overflow | Sign change integer conversion overflow | “Find defects (C/C++)”

More About

- “Numerical Defects”
- “Review and Comment Results”

Unsigned integer overflow

Overflow from operation between unsigned integers

Description

Unsigned integer overflow occurs when an operation on unsigned integer variables exceeds the space available to represent the resulting value. The exact storage allocation for different integer types depends on your processor. See “Target processor type (C)” on page 1-5 or “Target processor type (C++)” on page 2-2.

Examples

Add One to Maximum Unsigned Integer

```
unsigned int plusplus(void) {

    unsigned uvar = UINT_MAX;
    uvar++;
    return uvar;
}
```

In the third statement of this function, the variable `uvar` is increased by 1. However, the value of `uvar` is the maximum unsigned integer value, so 1 plus the maximum integer value cannot be represented by an `unsigned int`. The C programming language standard does not view unsigned overflow as an error because the program automatically reduces the result by modulo the maximum value plus 1. In this example, `uvar` is reduced by modulo `UINT_MAX`. The result is `uvar = 1`.

Correction — Different Storage Type

One possible correction is to store the operation result in a larger data type. In this example, by returning an `unsigned long` instead of an `unsigned int`, the overflow error is fixed.

```
unsigned long plusplus(void) {

    unsigned uvar = UINT_MAX;
    unsigned long ulvar = uvar++;
    return ulvar;
}
```

Check Information

Category: Numerical
Language: C | C++
Default: off
Command-Line Syntax: `uint_ovfl`

See Also

Integer overflow | Float overflow | “Find defects (C/C++)”

More About

- “Numerical Defects”
- “Review and Comment Results”

Useless if

Unnecessary if conditional

Description

Useless if occurs on if-statements where the condition is always true. This defect occurs only on if-statements that do not have an else-statement.

This defect shows unnecessary if-statements when there is no difference in code execution if the if-statement is removed.

Examples

if with Enumerated Type

```
typedef enum _suit {UNKNOWN, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card < 7) {
        do_something(card);
    }
}
```

The type `suit` is enumerated with five options. However, the conditional expression `card < 7` always evaluates to true because `card` can be at most 5. The `if` statement is unnecessary.

Correction — Change Condition

One possible correction is to change the if-condition in the code. In this correction, the 7 is changed to `UNKNOWN_SUIT` to relate directly to the type of `card`.

```
typedef enum _suit {UNKNOWN, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    if (card > UNKNOWN_SUIT) {
        do_something(card);
    }
}
```

Correction — Remove If

Another possible correction is to remove the if-condition in the code. Because the condition is always true, you can remove the condition to simplify your code.

```
typedef enum _suit {UNKNOWN, SPADES, HEARTS, DIAMONDS, CLUBS} suit;
suit nextcard(void);
void do_something(suit s);

void bridge(void)
{
    suit card = nextcard();
    if ((card < SPADES) || (card > CLUBS)){
        card = UNKNOWN_SUIT;
    }

    do_something(card);
}
```

Check Information

Category: Data-flow
Language: C | C++
Default: on
Command-Line Syntax: `useless_if`

See Also

“Find defects (C/C++)” | `Code deactivated by constant false condition` | `Dead code` | `Unreachable code`

More About

- “Data-flow Defects”
- “Review and Comment Results”

Use of previously freed pointer

Memory accessed after deallocation

Description

Use of previously freed pointer occurs when a block of memory is accessed after it is freed using the `free` function.

Examples

Use of Previously Freed Pointer Error

```
#include <stdlib.h>
#include <stdio.h>
 int increment_content_of_address(int base_val, int shift)
   {
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;
    free(pi);

    j = *pi + shift;
    /* Defect: Reading a freed pointer */

    return j;
   }
```

The `free` statement releases the block of memory that `pi` refers to. Therefore, dereferencing`pi` after the `free` statement is not valid.

Correction — Free Pointer After Use

One possible correction is to free the pointer `pi` only after the last instance where it is accessed.

```
int increment_content_of_address(int base_val, int shift)
```

```
{
    int j;
    int* pi = (int*)malloc(sizeof(int));
    if (pi == NULL) return 0;

    *pi = base_val;

    j = *pi + shift;
    *pi = 0;

    /* Fix: The pointer is freed after its last use */
    free(pi);
    return j;
}
```

Check Information

Category: Dynamic memory
Language: C | C++
Default: on
Command-Line Syntax: `freed_ptr`

See Also

Deallocation of previously deallocated pointer | “Find defects (C/C++)”

More About

- “Dynamic Memory Defects”
- “Review and Comment Results”

Variable shadowing

Variable hides another variable of same name with nested scope

Description

Variable shadowing occurs when a variable hides another variable of the same name with nested scope.

Examples

Variable Shadowing Error

```
#include <stdio.h>

int fact[5]={1,2,6,24,120};

int factorial(int n)
 {
  int fact=1;
  /*Defect: Local variable hides global array with same name */

  for(int i=1;i<=n;i++)
    fact*=i;

  return(fact);
 }
```

Inside the `factorial` function, the integer variable `fact` hides the global integer array `fact`.

Correction — Change Variable Name

One possible correction is to change the name of one of the variables, preferably the one with more local scope.

```
#include <stdio.h>

int fact[5]={1,2,6,24,120};
int factorial(int n)
 {
  /* Fix: Change name of local variable */
  int f=1;

  for(int i=1;i<=n;i++)
    f*=i;

  return(f);
 }
```

Check Information

Category: Data-flow
Language: C | C++
Default: on for handwritten code, off for generated code
Command-Line Syntax: `var_shadowing`

See Also

"Find defects (C/C++)"

More About

- "Data-flow Defects"
- "Review and Comment Results"

Write without further read

Variable never read after assignment

Description

Write without further read occurs when a value assigned to a variable is never read.

Examples

Write Without Further Read Error

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();
    /* Defect: Useless write */
}
```

After the variable `level` gets assigned the value `4 * getsensor()`, it is not read.

Correction — Use Value After Assignment

One possible correction is to use the variable `level` after the assignment.

```
void sensor_amplification(void)
{
    extern int getsensor(void);
    int level;

    level = 4 * getsensor();

    /* Fix: Use level after assignment */
    printf('The value is %d', level)

}
```

The variable `level` is printed, reading the new value.

Check Information

Category: Data-flow
Language: C | C++
Default: on for handwritten code, off for generated code
Command-Line Syntax: `useless_write`

See Also

“Find defects (C/C++)”

More About

- “Data-flow Defects”
- “Review and Comment Results”

Wrong allocated object size for cast

Allocated memory does not match destination pointer

Description

Wrong allocated object size for cast occurs during pointer conversion when the pointer's address is unaligned. If a pointer is converted to a different pointer type, the size of the allocated memory must be a multiple of the size of the destination pointer.

Examples

Dynamic Allocation of Pointers

```
void dyn_non_align(void){
    void *ptr = malloc(13);
    long *dest;

    dest = (long*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to a `long*` in line 5. The dynamically allocated memory of `ptr`, 13 bytes, is not a multiple of the size of `dest`, 4 bytes. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the allocated memory to 12 instead of 13.

```
void dyn_non_align(void){
    void *ptr = malloc(12);
    long *dest;

    dest = (long*)ptr;
}
```

Static Allocation of Pointers

```
void static_non_align(void){
    char arr[13], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr; //defect
}
```

In this example, the software raises a defect on the conversion of `ptr` to an `int*` in line 6. `ptr` has a memory size of 13 bytes because the array `arr` has a size of 13 bytes. The size of `dest` is 4 bytes, which is not a multiplie of 13. This misalignment causes the **Wrong allocated object size for cast** defect.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the size of the array `arr` to a multiple of 4.

```
void static_non_align(void){
    char arr[12], *ptr;
    int *dest;

    ptr = &arr[0];
    dest = (int*)ptr;
}
```

Allocation with a Function

```
void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(13);  //defect
    dest2 = (char*)my_alloc(13); //not a defect
}
```

In this example, the software raises a defect on the conversion of the pointer returned by `my_alloc(13)` to an `int*` in line 11. `my_alloc(13)` returns a pointer with a dynamically allocated size of 13 bytes. The size of `dest1` is 4 bytes, which is not a divisor of 13. This misalignment causes the **Wrong allocated object size for cast** defect. In line 12, the same function call, `my_alloc(13)`, does not call a defect for the conversion to `dest2` because the size of `char*`, 1 byte, a divisor of 13.

Correction — Change the Size of the Pointer

One possible correction is to use a pointer size that is a multiple of the destination size. In this example, resolve the defect by changing the argument for `my_alloc` to a multiple of 4.

```
void *my_alloc(int size) {
    void *ptr_func = malloc(size);
    if(ptr_func == NULL) exit(-1);
    return ptr_func;
}

void fun_non_align(void){
    int *dest1;
    char *dest2;

    dest1 = (int*)my_alloc(12);
    dest2 = (char*)my_alloc(13);
}
```

Check Information

Category: Static Memory
Language: C | C++
Default: off
Command-Line Syntax: `object_size_mismatch`

See Also

Unreliable cast of pointer | “Find defects (C/C++)”

More About

- “Static Memory Defects”

- “Review and Comment Results”

Wrong type used in sizeof

sizeof argument does not match pointer type

Description

Wrong type used in sizeof occurs when the size specified for the block of memory does not match the pointer type being initialized.

Examples

Allocate a Char Array With sizeof

```
void test_case_1(void) {
    char* str;

    str = malloc(sizeof(char*) * 5);
    free(str);

}
```

In this example, memory is allocated for the character pointer str using a malloc of five char pointers. However, str is a pointer to a character, not a pointer to a character pointer. Therefore the sizeof argument, char*, is incorrect.

Correction — Match Pointer Type to sizeof Argument

One possible correction is to match the argument to the pointer type. In this example, str is a character pointer, therefore the argument must also be a character.

```
void test_case_1(void) {
    char* str;

    str = malloc(sizeof(char) * 5);
    free(str);

}
```

Check Information

Category: Programming
Language: C | C++
Default: on for handwritten code, off for generated code
Command-Line Syntax: `ptr_sizeof_mismatch`

See Also

"Find defects (C/C++)"

More About

- "Programming Defects"
- "Review and Comment Results"

5

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.

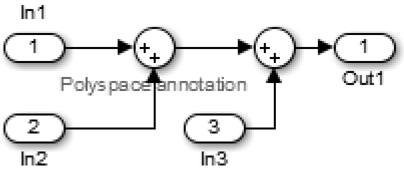
In the generated code associated with the annotated block, Polyspace adds code comments before and after the lines of code. Polyspace reads these comments and marks Polyspace results of the specified `kind` with the annotated information.

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 Bug Finder Analysis

Use the Polyspace annotation function to annotate a block and see the annotation in the analysis results.

In the example model WhereAreTheErrors_v2, add an annotation to the switch block for MISRA C rule 13.7 violations with a comment, a classification, and a status.

```
model = 'WhereAreTheErrors_v2';
open(model)
pslinkfun('annotations','type','Misra-C', 'kind', '13.7','block',...
 'WhereAreTheErrors_v2/Switch1','status','fix','comment','must fix')
```

In the open model, you can see a Polyspace annotation added to the Switch block.

Generate code for the model and run an analysis. After the analysis is finished, open the results in the Polyspace environment:

```
slbuild(model)
opts = pslinkoptions(model);
opts.VerificationMode = 'BugFinder';
opts.VerificationSettings = 'PrjConfigAndMisra';
pslinkrun(model,opts)
pslinkfun('openresults',model)
```

The five MISRA C 13.7 rule violations are annotated with the information you added to the switch block. The annotations appear in the **Status** and **Comment** columns.

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 = 'BugFinder';
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

'DEFECT' | 'MISRA-C' | 'MISRA-AC-AGC' | 'MISRA-CPP' | 'JSF'

The type of result with which to annotate the block, specified as:

- ‘DEFECT’ for defects.
- ‘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
‘DEFECT’	Use the abbreviation associated with the type of defect that you want to annotate. For example, 'int_ovfl' – Integer overflow. For the list of possible checks, see: “Polyspace Bug Finder Results”.
‘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 “MISRA C:2004 Coding Rules”.
‘MISRA-AC-AGC’	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 “MISRA C:2004 Coding 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 “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 “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.

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 Bug Finder analysis.

```
opts = pslinkoptions(model);
opts.VerificationMode = 'BugFinder';
```

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 Bug Finder analysis.

```
opts = pslinkoptions(model);
opts.VerificationMode = 'BugFinder';
```

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`

polyspaceBugFinder

Run Polyspace Bug Finder analysis from MATLAB

Syntax

```
polyspaceBugFinder
```

```
polyspaceBugFinder(projectFile)
polyspaceBugFinder(resultsFile)
polyspaceBugFinder('-results-dir',resultsFolder)
```

```
polyspaceBugFinder('-help')
```

```
polyspaceBugFinder('-sources',sourceFiles)
polyspaceBugFinder('-sources',sourceFiles,Name,Value)
```

Description

`polyspaceBugFinder` opens Polyspace Bug Finder.

`polyspaceBugFinder(projectFile)` opens a Polyspace project file in Polyspace Bug Finder.

`polyspaceBugFinder(resultsFile)` opens a Polyspace results file in Polyspace Bug Finder.

`polyspaceBugFinder('-results-dir',resultsFolder)` opens a Polyspace results file from resultsFolder in Polyspace Bug Finder.

`polyspaceBugFinder('-help')` displays options that can be supplied to the `polyspaceBugFinder` command to run a Polyspace Bug Finder analysis.

`polyspaceBugFinder('-sources',sourceFiles)` runs a Polyspace Bug Finder analysis on the source files specified in sourceFiles.

`polyspaceBugFinder('-sources',sourceFiles,Name,Value)` runs a Polyspace Bug Finder analysis 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 `Bug_Finder_Example.psprj` from the folder *`Matlab_Install`*`\polyspace\examples\cxx\Bug_Finder_Example`.

Assign the full path to the project file to a MATLAB variable `prjFile`.

```
prjFile = fullfile(matlabroot, 'polyspace', 'examples', 'cxx', ...
         'Bug_Finder_Example', 'Bug_Finder_Example.psprj');
```

Use `prjFile` to open the project.

```
polyspaceBugFinder(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\Bug_Finder_Example\Results`.

Assign the full path to the folder to a MATLAB variable `resFolder`.

```
resFolder = fullfile(matlabroot, 'polyspace', 'examples',  ...
         'cxx', 'Bug_Finder_Example', 'Results');
```

Use `resFolder` to open the results.

```
polyspaceBugFinder('-results-dir',resFolder)
```

Run Polyspace Analysis from MATLAB

This example shows how to run a Polyspace analysis 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.

```
polyspaceBugFinder('-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:

```
polyspaceBugFinder('-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 5-22. Here you specify checking of MISRA C coding rules using the option `-misra2`. For more information on this option, see "Check MISRA C:2004".

Assign the source file path to a MATLAB variable `sourceFileName`.

```
sourceFileName = fullfile(matlabroot, 'polyspace',...
'examples', 'cxx', 'Bug_Finder_Example','sources','dataflow.c')
```

Assign the results folder path to a MATLAB variable `resFolder`.

```
resFolder = fullfile('C:\','Polyspace_Results')
```

Run Polyspace Bug Finder analysis with additional option `-misra2`.

```
polyspaceBugFinder('-sources',sourceFileName,...
     '-results-dir',resFolder,'-misra2','required-rules')
```

Open the results file.

```
polyspaceBugFinder('-results-dir',resFolder)
```

- "Specify Options from MATLAB Command Line"

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 `.psbf` file

string

Name of results file with extension `.psbf`, 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 options that can also be set from the user interface, see the **Command-Line Information** section in:
  - “Analysis Options for C”
  - “Analysis Options for C++”
- For options that cannot be set from the user interface, see the **Polyspace Analysis Options** section in “Command-Line Invocation”.

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**.

```
polyspaceBugFinder('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:

```
polyspaceBugFinder -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 `polyspaceBugFinder`.
`-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',...
'Bug_Finder_Example','sources');
copyfile(demo,'C:\psdemo\src\')
```

Submit two jobs to your scheduler.

```
polyspaceBugFinder -batch -scheduler myMJS@myCompany.com
    -sources C:\psdemo\src\*.c'
    -results-dir 'C:\psdemo\res1'
polyspaceBugFinder -batch -scheduler myMJS@myCompany.com
    -sources 'C:\psdemo\src\numeric.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

`polyspaceBugFinder`

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 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

`'MISRA-C' | 'MISRA-CPP' | 'JSF'`

The type of result with which to annotate the block, specified as:

- `‘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
`‘MISRA-C’`	Use the rule number you want to annotate. For example, `'2.2'`. For the list of supported MISRA C rules and their numbers, see “MISRA C:2004 Coding Rules”.
`‘MISRA-CPP’`	Use the rule number you want to annotate. For example, `'0-1-1'`. For the list of supported MISRA C++ rules and their numbers, see “MISRA C++ Coding Rules”.
`‘JSF’`	Use the rule number you want to annotate. For example, `'3'`. For the list of supported JSF C++ rules and their numbers, see “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`.

Example: `'status','no action planned'`

`'comment'` — additional comments

string

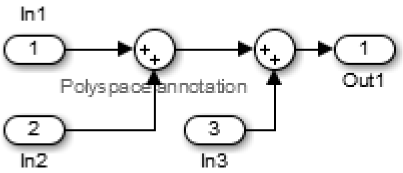
Additional comments specified as a string. The comments provide more information about why the results are justified.

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';
```

Change the analysis options to run a Bug Finder analysis:

```
config.VerificationMode = 'BugFinder';
```

```
config = 

                   ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfigAndMisra'
          OpenProjectManager: 0
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'BugFinder'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
     CxxVerificationSettings: 'PrjConfig'
```

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

`'BugFinder'` (default) | `'CodeProver'`

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`

6

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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”

- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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 Bug Finder analysis.

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

- “Activate Coding Rules Checker”

- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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 “Annotate Code for Rule Violations”.

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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “Review Coding Rule Violations”
- “Polyspace MISRA C:2012 Checker”

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

- "Activate Coding Rules Checker"
- "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 defect analysis. 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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”

- “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

- "Activate Coding Rules Checker"

- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”

- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"

- “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

- "Activate Coding Rules Checker"

- “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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”

- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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 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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”

- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"

- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"

- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"

- “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

- "Activate Coding Rules Checker"

- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”

- “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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”

- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"

- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"

- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"

- “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

- “Activate Coding Rules Checker”
- “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

- "Activate Coding Rules Checker"
- "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

- “Activate Coding Rules Checker”

- “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

- "Activate Coding Rules Checker"
- "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

- "Activate Coding Rules Checker"
- "Review Coding Rule Violations"
- "Polyspace MISRA C:2012 Checker"
- "Software Quality Objective Subsets (C:2012)"

Introduced in R2014b

7

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.	

8

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, see “Review Code Metrics”.

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, see “Review Code Metrics”.

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, see “Review Code Metrics”.

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, see “Review Code Metrics”.

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, see "Review Code Metrics".

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, see “Review Code Metrics”.

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.

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, see “Review Code Metrics”.

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, see “Review Code Metrics”.

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, see “Review Code Metrics”.

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, see “Review Code Metrics”.

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 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, see “Review Code Metrics”.

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 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, see “Review Code Metrics”.

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`