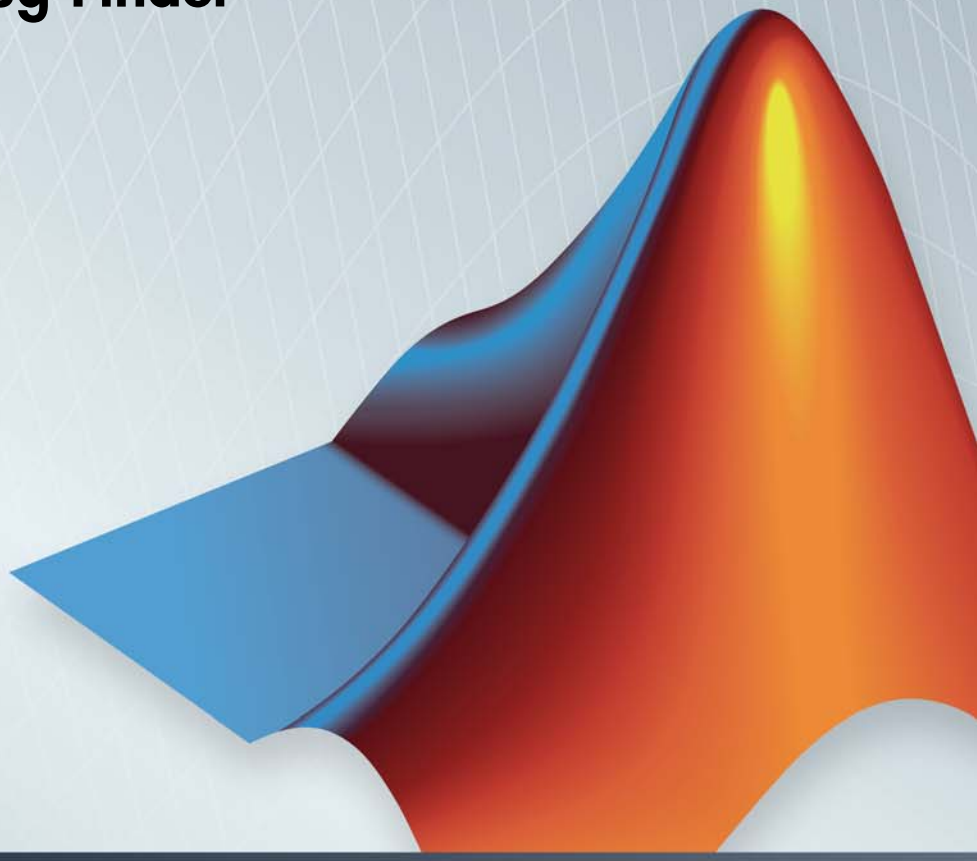
# Polyspace® Bug Finder™

Reference

**R**2014**a**



MATLAB® & SIMULINK®

MathWorks®

### How to Contact MathWorks

| | |
|---|---|
| `www.mathworks.com` | Web |
| `comp.soft-sys.matlab` | Newsgroup |
| `www.mathworks.com/contact_TS.html` | Technical Support |

| | |
|---|---|
| `suggest@mathworks.com` | Product enhancement suggestions |
| `bugs@mathworks.com` | Bug reports |
| `doc@mathworks.com` | Documentation error reports |
| `service@mathworks.com` | Order status, license renewals, passcodes |
| `info@mathworks.com` | Sales, pricing, and general information |

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

*Polyspace® Bug Finder™ Reference*



#### Trademarks

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

#### Patents

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

#### Revision History

| | | |
|---|---|---|
| September 2013 | Online only | New for Version 1.0 (Release 2013b) |
| March 2014 | Online only | Revised for Version 1.1 (Release 2014a) |

# Contents

## 1 Option Descriptions

## 2 Option Descriptions for C++ Code

1

# Option Descriptions

# Target operating system

Specify the operating system of your target application.

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" on page 1-5 | "Dialect" on page 1-10 | "Dialect" on page 2-4

## Related Examples

- "Specify Analysis Options"

## Concepts

- "Compile Operating System-Dependent Code"

# Target processor type

Specify the target processor type.

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`

- `i386`
- `m68k`
- `powerpc`
- `c-167`
- `x86_64`
- `tms320c3x`
- `sharc21x61`
- `necv850`
- `hc08`
- `hc12`
- `mpc5xx`
- `c18`
- `mcpu...(Advanced)`

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

## 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" on page 1-7

## Related Examples

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

# Generic target options

The **Generic target options** dialog box is only available when you select a `mcpu` target.

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.<br><br>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.<br><br>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`.<br><br>`signed`: Specifies that `char` is signed, overriding target's default.<br><br>`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<br><br>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`, `sharc21x61`, `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.<br>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` |

# Dialect

Allow syntax associated with C language extensions.

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.

`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, 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`
**Type:** string
**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" on page 1-3 | "Target processor type" on page 1-5

## Related Examples

- "Analyze Keil or IAR Dialects"

# Sfr type support

Specifies the `sfr` types.

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

**Dialect** enables this parameter.

## Command-Line Information

**Parameter:** `-sfr-types` *`sfr_name=size_in_bits`*`,...`
**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

Specifies how division and modulus of a negative numbers is interpreted by the analysis

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.

---

**With the *-div-round-down* option:**

**Example Shell Script Entry:**

```
polyspace-bug-finder-nodesktop -div-round-down ...
```

## 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`

: `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

Allows the analysis to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition.

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

## Settings

**Default:** `signed-int`

`signed-int` On
: 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

Choose between arithmetical and logical computation.

## Settings

**Default: `Arithmetic`**

**`Arithmetic`**

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`
**Default:** off
**Example:** `polyspace-bug-finder-nodesktop -logical-signed-right-shift`

# Preprocessor definitions

Define macro compiler flags to be used during compilation phase.

Some defines are applied by default, depending on your **Target operating system**.

## Settings

**No default:**

Enter only one flag per line. The flag must be in the format *`Flag=Value`*. If you want Polyspace to ignore the flag, leave the *`Value`* blank.

For example, `USE_COM1` to replace all instances of `USE_COM1` by 1. Or `USE_COM1=` to ignore `USE_COM1`.

## Command-Line Information

You can specify only one flag with each `-D` option. However, you can specify the option multiple times.

**Parameter:** `-D`
**Value:** macro compiler flags
**Example:** `polyspace-bug-finder-nodesktop -D HAVE_MYLIB -D USE_COM1=`

# Undefined preprocessor definitions

Specify macro compiler flags as undefined

Some defines are applied by default, depending on your **Target operating system**.

## Settings

**No default**

Enter only one flag per line to be undefined during the analysis.

## Command-Line Information

You can specify only one flag with each `-U` option. However, you can specify the option multiple times.

**Parameter:** `-U`
**Value:** macro compiler flags
**Example:** `polyspace-bug-finder-nodesktop -U HAVE_MYLIB -U USE_COM1`

# Code from DOS or Windows file system

Specify that DOS or Windows® files are in analysis

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

Specify a perl script to run on each source file after the preprocessing phase

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 .ci file. Adding a line or removing one could result in some unpredictable behavior on the location of checks and MACROS in the Polyspace viewer.

You can find each preprocessed file in the results directory in the zipped file ci.zip located in*`results`*`/ALL/SRC/MACROS`. The extension of the preprocessed file is .ci.

---

**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 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*`\matlab\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`
**Value:** Path to script (and path to executable on Windows)

# Include

Specify files to be included by each C file involved in the analysis.

## 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>`.

## Command-Line Information

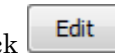
**Parameter:** `-include`
**Value:** file name
**Example:** `polyspace-bug-finder-nodesktop -include` `` `pwd`/sources/a_file.h -include /inc/inc_file.h ``

# Variable/function range setup

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.

## Settings

**No default**

Enter full path to the template file. Alternately, click 

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:** full path to Data Range Specifications template file
**Example:** `polyspace-code-prover-nodesktop -sources` *`file_name`*
`-data-range-specifications "C:\DRS\range.txt"`

## See Also

"Functions to stub" on page 1-27 **|** "Ignore default initialization of global variables" on page 1-26

## Related Examples

- "Specify Analysis Options"
- "Specify Data Ranges for Global Variables"

## Concepts

- "Overview of Data Range Specifications (DRS)"

# Ignore default initialization of global variables

Specify that Polyspace must treat global variables as non-initialized by default.

## Settings

**Default**: Off

☑ On

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

☐ Off

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

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

## Command-Line Information

**Parameter:** `-no-def-init-glob`

## See Also

"Functions to stub" on page 1-27 | "Variable/function range setup" on page 1-25

## Related Examples

- "Specify Analysis Options"

# Functions to stub

Specify functions that you want the software to stub.

## Settings

Click  to add a field. Enter function name.

## Command-Line Information

**Parameter:** `-functions-to-stub`
**Value:** Function name
**Example:** `polyspace-bug-finder-nodesktop -sources` *`file_name`*
`-functions-to-stub function_1,function_2`

## See Also

"Ignore default initialization of global variables" on page 1-26 | "Variable/function range setup" on page 1-25

## Related Examples

- "Specify Analysis Options"

# Entry points

Specify functions that serve as entry points to your code. Use this option when your code is intended for multitasking.

## Settings

**Default:** none

Click  to add a field. Enter function name.

## Dependencies

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

## Tips

- The entry point function must:
  - Have the form `void functionName (void)`.
  - Contain an infinite loop.

- If a function `func` takes arguments, you cannot use it directly as entry point. To use `func` as entry point:
  1. Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
  2. Declare arguments to `func` as `volatile` variables local to `newFunc`. Call `func` inside `newFunc`.
  3. Specify `newFunc` as entry point.
- If a function `func` does not contain an infinite loop, you cannot use it directly as entry point. To use `func` as entry point:
  1. Create a new function `newFunc`. The declaration must be of the form `void newFunc (void)`.
  2. Call `func` inside an infinite loop in `newFunc`. For example:

     ```
     void newFunc(void) {
     ```

```
  while(1) {
     func();
   }
}
```

**3** Specify `newFunc` as entry point.

## Command-Line Information

**Parameter:** `-entry-points`
**Value:** Function name
**Example:** `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-entry_points func_1,func_2`

## See Also

"Critical section details" on page 1-30 | "Temporally exclusive tasks" on page 1-32

## Related Examples

- "Specify Analysis Options"
- "Model Synchronous Tasks"

# Critical section details

Specify procedures that begin and end critical sections. Polyspace considers that variables in critical sections cannot be accessed simultaneously by multiple tasks.

## Settings

**Default:** none

Click  to add a field.

- In **Procedure beginning**, enter name of procedure that begins the critical section.
- In **Procedure ending**, enter name of procedure that ends the critical section.

## Dependencies

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

## Tips

- You cannot use this option if you select **Code Prover Verification > Verify Module** on the **Configuration** pane.

## Command-Line Information

**Parameter:** `-critical-section-begin | -critical-section-end`
**Value:** Entries in the form "*`procedure_1_name`*:*`critical_section_name`*"
**Example:** `polyspace-bug_finder-nodesktop -sources` *`file_name`* `-critical-section-begin "start_my_semaphore:cs" -critical-section-end "end_my_semaphore:cs"`

## See Also

"Entry points" on page 1-28 | "Temporally exclusive tasks" on page 1-32

## Related Examples

- "Specify Analysis Options"

## Concepts

- “Critical Sections”
- “Are Interruptions Maskable or Preemptive?”
- “Semaphores”
- “Mutual Exclusion”

# Temporally exclusive tasks

Specify functions that cannot execute simultaneously. Use this option to implement temporal exclusion for multitasking code.

## Settings

**Default:** none

Click  to add a field. In each field, enter a comma-separated list of functions. Polyspace considers that the functions in the list cannot execute simultaneously.

## Dependencies

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

## Tips

- You cannot use this option if you select **Code Prover Verification > Verify Module** on the **Configuration** pane.

## 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`
**Value:** Name of temporal exclusions file
**Example:** `polyspace-bug-finder-nodesktop -sources` *`file_name`*
`-temporal-exclusions-file "C:\exclusions_file.txt"`

## See Also

"Entry points" on page 1-28 | "Critical section details" on page 1-30

## Related Examples

- "Specify Analysis Options"

## Concepts

- “Mutual Exclusion”
- “Are Interruptions Maskable or Preemptive?”
- “Critical Sections”
- “Semaphores”

# Check MISRA C:2004 rules

Specify whether to check for violation of MISRA C®:2004 rules. Each option corresponds to a subset of rules to check. After verification, the **Results Summary** pane lists the coding rule violations.

## Settings

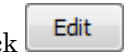
**Default:** `required-rules`

`required-rules`
: Check required coding rules.

`all-rules`
: Check required and advisory coding rules.

`SQO-subset1`
: Check only a subset of MISRA C rules. For more information, see "Software Quality Objective Subsets (C)".

`SQO-subset2`
: Check a subset of rules including `SQO-subset1` and some additional rules. For more information, see "Software Quality Objective Subsets (C)".

`custom`
: Specify coding rules to check. Click 

to create a coding rules file. After creating and saving the file, to reuse it for another project, enter full path to the file in the space provided.

  Format of the custom file:

  *rule number* `off|warning`

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

```
# MISRA configuration file for my_project
10.5 off # disable misra rule number 10.5
17.2 warning # violation misra rule 17.2 is a warning
17.3 warning # violation of misra rule 17.3 is a warning
```

## 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 | -custom` *`file_name`*
**Example:** `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-misra2 all-rules`

**See Also** "Files and folders to ignore" on page 1-40

**Related Examples**

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

**Concepts**

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

# Check MISRA AC AGC rules

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 option corresponds to a subset of rules to check. After verification, the **Results Summary** pane lists the coding rule violations.

## Settings

**Default**: `OBL-rules`

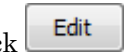
`OBL-rules`
: Check required coding rules.

`OBL-REC-rules`
: Check required and recommended rules.

`all-rules`
: Check required, recommended and readability-related rules.

`SQO-subset1`
: Check a subset of rules. For more information, see "Software Quality Objective Subsets (AC AGC)".

`SQO-subset2`
: Check a subset of rules including `SQO-subset1` and some additional rules. For more information, see "Software Quality Objective Subsets (AC AGC)".

`custom`
: Specify coding rules to check. Click 

to create a coding rules file.

  After creating and saving the file, to reuse it for another project, enter full path to the file in the space provided.

  Format of the custom file:

  ```
  rule number off|warning
  ```

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

  ```
  # MISRA configuration file for my_project
  10.5 off # disable misra rule number 10.5
  ```

```
17.2 warning # violation misra rule 17.2 is a warning
17.3 warning # violation of misra rule 17.3 is a warning
```

## 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` | `-custom` *`file_name`*
**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”

## Concepts

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

# Check custom rules

Define naming conventions for identifiers and check your code against them.

## 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 rules**, 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:

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

   d. Rule number, followed by `warning`. For example:

   ```
   4.3 warning
   ```

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

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

   f. 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
**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"

## Concepts

- "Format of Custom Coding Rules File"
- "Custom Naming Convention Rules"

# Files and folders to ignore

Specify files and folders that the coding rules checker must ignore.

## Settings

**Default**: `all-headers`

`all-headers`
: The coding rules checker ignores include folders that contain `.h` files only.

`all`
: The coding rules checker ignores all include folders.

`custom`
: The coding rules checker ignores include files and folders that you specify.

## Dependencies

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

## Command-Line Information

**Parameter:** `-includes-to-ignore`
**Value:** `all-headers | all | -custom` *`file_name`*
**Default:** `all-headers`
**Example:** `polyspace-bug-finder-nodesktop -sources` *`file_name`* `-misra2 required-rules -includes-to-ignore "C:\usr\include"`

## Related Examples

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

# Effective boolean types

Specify data types that you want Polyspace to treat as Boolean. Use this option to adjust the checking of MISRA C or MISRA AC AGC rules 12.6, 13.2, and 15.4.

## Settings

**Default:** none

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 rules** or **Check MISRA AC AGC rules**.

## Command-Line Information

**Parameter:** `-boolean-types`
**Value:** Name of data type
**Example:** `polyspace-bug-finder-nodesktop -sources` *`filename`*
`-misra2 required-rules -boolean-types boolean1_t,boolean2_t`

## Related Examples

- "Activate Coding Rules Checker"

## Concepts

- "MISRA C:2004 Coding Rules"

# Allowed pragmas

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.

## Settings

**Default:** none

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 rules** or **Check MISRA AC AGC rules**.

## Command-Line Information

**Parameter:** `-allowed-pragmas`
**Value:** Name of pragma
**Example:** `polyspace-bug-finder-nodesktop -sources` *`filename`*
`-misra2 required-rules -allowed-pragmas pragma_01,pragma_02`

## Related Examples

- “Activate Coding Rules Checker”

## Concepts

- “MISRA C:2004 Coding Rules”

# Find defects

Enable or disable defect checking. Activate different defect checkers.

## 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 -disable-checkers FLOAT_ZERO_DIV`
checks for all the numerical defects except "Float division by zero".

## See Also

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

## Related Examples

- "Specify Analysis Options"

**Concepts**

- "Bug Finder Defect Categories"

# Generate report

Specify whether to generate a report during the analysis. 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 **Run > 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" on page 1-46 | "Output format" on page 1-48

## Related Examples

- "Specify Analysis Options"
- "Generate Reports"

# Report template

Specify template for generating analysis report. 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 quantities related to the source code. 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 verification. For more information, see "Analysis Options for C" or "Analysis Options for C++".

`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" on page 1-45 | "Output format" on page 1-48

## Related Examples

- "Generate Reports"

# Output format

Specify output format of generated report.

## 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.

## 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" on page 1-48 | "Report template" on page 1-46

## Related Examples

- “Specify Analysis Options”
- “Generate Reports”

# Interactive

Enable or disable interactive remote analysis. For interactive remote analysis, you need:

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

## Settings

**Default:** Off

☑ On

Run interactive analysis on a remote computer. In this remote analysis mode, the analysis is tethered to your local computer. Therefore, on your local computer:

- If you are running the analysis from the Polyspace user interface, you cannot close the user interface while the analysis is running.
- If you are running the analysis from the command line, you cannot close the command-line window while the analysis is running.

In this mode, the analysis is not queued on the cluster. Therefore, if a worker is not available on the cluster, the analysis aborts.

The software downloads the results to your local computer after the analysis.

☐ Off

Do not run interactive analysis on a remote computer.

## Command-Line Information

If you do not have remote verification setup already, to run an interactive remote verification from the command line, use with the `-scheduler` option.

**Parameter:** `-interactive`
**Value:** `-scheduler` *`host_name`* if you have not set the **Job scheduler host name** in the Polyspace user interface

**Example:** `polyspace-bug-finder-nodesktop -interactive -scheduler NodeHost`
`polyspace-bug-finder-nodesktop -interactive -scheduler MJSName@NodeHost`

## See Also

"Batch" on page 1-52

## Related Examples

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

# Batch

Enable or disable batch remote analysis. 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 Queue Manager. To use the Polyspace Queue Manager:

- In the Polyspace user interface, click  .
- On the DOS or UNIX® command line, use the `polyspace-jobs-manager` command. For more information, see "Manage Remote Analyses at the Command Line".
- On the MATLAB command line, use the `polyspaceJobsManager` function. For more information, see `polyspaceJobsManager`.

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
**Example:** `polyspace-bug-finder-nodesktop -batch -scheduler NodeHost`
`polyspace-bug-finder-nodesktop -batch -scheduler MJSName@NodeHost`

## See Also

"Interactive" on page 1-50 | "Add to results repository" on page 1-54

## Related Examples

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

# Add to results repository

Specify upload of analysis results to the Polyspace Metrics results repository, allowing Web-based reporting of results and code metrics.

## 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 available only for remote verifications.

## 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 Remote Verification and Analysis" | "Set Up Polyspace Metrics" | "Batch" on page 1-52

# Command/script to apply after the end of the code verification

Specify a command or script to be executed after the verification.

## Settings

**Default:** none

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`
**Value:** Full path to script
**Example:** `polyspace-bug-finder-nodesktop -sources` *`file_name`*
`-post-analysis-command` `` `pwd`/send_email ``

## Related Examples

- "Specify Analysis Options"

# Other

| In this section... |
| --- |
|  |
|  |
|  |
|  |

## -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
```

2

# Option Descriptions for C++ Code

# Target processor type

Specifies the target processor type

Doing so 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`

- `i386`
- `sparc`
- `m68k`
- `powerpc`
- `c-167`
- `x86_64`
- `mcpu...(Advanced)`

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

## 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" on page 1-7

**Related Examples**

- "Specify Analysis Options"
- "Predefined Target Processor Specifications"
- "Modify Predefined Target Processor Attributes"

- “Specify Generic Target Processors”

# Dialect

Allow syntax associated with C++ language extensions.

## Settings

**Default:** `none`

`none`
: Analysis allows only ANSI C standard syntax.

`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.

`iso`
: Analysis allows for gnu dialect of ISO C99.

`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 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`.

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

## Limitations

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

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

  **Example**

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

  struct B{
      int b;
  ```

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

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

The expected output from the above code is:

```
B constructor
A constructor
```

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

```
A constructor
B constructor
```

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

- **Vector types and attributes** — Not supported, 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, 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`
**Type:** string
**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" on page 1-3 | "Target processor type" on page 2-2

## Related Examples

- “Analyze Keil or IAR Dialects”

# Pack alignment value

Specifies the default packing alignment for an analysis.

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

Specifies a single directory to be included by *`#import`* directive.

## 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

Specifies C++ `#pragma` packing alignment for structure, union, and class members.

## 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`
**Example:** `polyspace-bug-finder-nodesktop -lang cpp -ignore-pragma-pack`

# Support managed extensions

Visual C++ /FX option allows the partial translation of sources making use of managed extensions to Visual C++ sources without managed extensions.

## 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

Allows the analysis to use different base types to represent an enumerated type, depending on the enumerator values and the selected definition.

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

Specify the scope of the index variable declared within a `for` loop.

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 w_char_t

Specify how to treat `wchar_t`

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

Specifies the underlying type of `wchar_t` to be unsigned long

## 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

Specifies the underlying type of `size_t` to be unsigned long.

## 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

Ignore linkage errors.

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`

# Check MISRA C++ rules

Specify whether to check for violation of MISRA C++ rules. Each option corresponds to a subset of rules to check. In the Results Manager, the **Results Summary** pane lists the coding rule violations.

## Settings

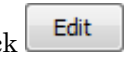
**Default:** `required-rules`

`required-rules`
: Check required coding rules.

`all-rules`
: Check required and advisory coding rules.

`SQO-subset1`
: Check only a subset of MISRA C++ rules. For more information, see "Software Quality Objective Subsets (C++)".

`SQO-subset2`
: Check a subset of rules including `SQO-subset1` and some additional rules. For more information, see "Software Quality Objective Subsets (C++)"

`custom`
: Specify coding rules to check. Click 

to create a coding rules file.

  After creating and saving the file, to reuse it for another project, enter full path to the file in the space provided.

  Format of the custom file:

  ```
  <rule number> off|warning
  ```

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

  ```
  # MISRA configuration file for my_project
  0-1-8 off # disable misra rule number 0-1-8
  1-0-1 warning # violation misra rule 1-0-1 is a warning
  0-1-1 warning # violation of misra rule 0-1-1 is a warning
  ```

## Command-Line Information

**Parameter:** `-misra-cpp`
**Value:** `required-rules | all-rules | SQO-subset1 | SQO-subset2 | -custom` *`file_name`*
**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"

## Concepts

- "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 option corresponds to a subset of rules to check. In the Results Manager, the **Results Summary** pane lists the coding rule violations.

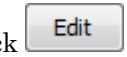
## Settings

**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 

to create a coding rules file.

  After creating and saving the file, to reuse it for another project, enter full path to the file in the space provided.

  Format of the custom file:

  ```
  <rule number> off|warning
  ```

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

  ```
  # JSF configuration file for my_project
  1 off # disable AV rule number 1
  8 warning # violation AV Rule 8 is a warning
  9 warning # violation AV Rule 9 is a warning
  ```

## Tips

- If you check for JSF C++ rules, the `-Wall` option is disabled.

- 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 | -custom` *`file_name`*
**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”

## Concepts

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

# Files and folders to ignore

Specify files and folders that the coding rules checker must ignore.

## Settings

**Default**: `all-headers`

`all-headers`
: The coding rules checker ignores include folders that contain `.h` files only.

`all`
: The coding rules checker ignores all include folders.

`custom`
: The coding rules checker ignores include files and folders that you specify.

## Dependencies

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

## Command-Line Information

**Parameter:** `-includes-to-ignore`
**Value:** `all-headers | all | -custom` *`file_name`*
**Default:** `all-headers`
**Example:** `polyspace-bug-finder-nodesktop -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

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
```

## -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
```

3

# Polyspace Command-Line Options

# -asm-begin -asm-end

## Purpose

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_b
   -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`

## Related Examples

- "Run Analysis from the Command Line"

# -author

## Purpose

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

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

## Related Examples

- "Run Analysis from the Command Line"

## Purpose

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

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

## Related Examples

- "Run Analysis from the Command Line"

# -h[elp]

## Purpose

Display help in the shell window

## Syntax

```
-h
-help
```

## Description

`-h` and `-help` display a simple list of analysis option information in the shell window.

## Examples

Display the command-line help.

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

## See Also

`polyspaceBugFinder`

## Related Examples

- “Run Analysis from the Command Line”

## Purpose

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`

## Related Examples

- "Run Analysis from the Command Line"

# -import-comments

## Purpose

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 and import comments from the previous results.

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

## See Also

`polyspaceBugFinder`

## Related Examples

- "Run Analysis from the Command Line"

## Purpose

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 environment, 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`

## Related Examples

- "Run Analysis from the Command Line"

# -less-range-information

## Purpose

Limit range information displayed in results

## Syntax

```
-less-range-information
```

## Description

`-less-range-information` limits the amount of range information displayed in the results.

By enabling this option, range information is available only for assignments, not read operations.

Because computing range information for read operations can take a long time, selecting this option can reduce verification time significantly.

## Examples

Consider the following code:

```
x = y + z
```

By enabling this option:

```
polyspace-bug-finder-nodesktop -less-range-information
```

range information is available only when you place your cursor over `x`. Without this option enabled, range information is available for `x`, `y`, and `z`.

## Purpose

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 8.

## Examples

Disable parallel processing during the analysis.

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

## See Also

`polyspaceBugFinder`

## Related Examples

- "Run Analysis from the Command Line"

# -no-pointer-information

## Purpose

Turn off pointer information in your results

## Syntax

```
-no-pointer-information
```

## Description

`-no-pointer-information` turns off the pointer information in your analysis results. When you select this option, the software does not provide pointer information through tooltips. As computing pointer information can take a long time, selecting this option can significantly reduce analysis time.

## Examples

Consider the following example:

```
x = *p;
```

If you do not select this option (the default), the software displays pointer information when you place the cursor on `p` or `*`. If you select this option, the software does not display information about the pointer.

## Purpose

Specify permissive verification mode

## Syntax

```
-permissive
```

## Description

`-permissive` selects a more lenient analysis for numerical overflows. This mode is equivalent to using `-ignore-constant-overflows` and `-allow-negative-operand-in-shift`.

## Examples

Analyze in permissive mode:

# -prog

## Purpose

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

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

## Related Examples

- "Run Analysis from the Command Line"

## Purpose

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

`polyspaceBugFinder` | "Output format" on page 1-48 | "Report template" on page 1-46

## Related Examples

- "Run Analysis from the Command Line"
- "Generate Reports"

# -results-dir

## Purpose

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`

## Related Examples

- "Run Analysis from the Command Line"

## Purpose

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`

## Related Examples

- “Run Analysis from the Command Line”
- “Manage Remote Verifications”

# -sources

## Purpose

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`

## Related Examples

- "Run Analysis from the Command Line"

## Purpose

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.

The file must list only one source file per line, and each file name must be given with its absolute path.

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`

## Related Examples

- "Run Analysis from the Command Line"

# -tmp-dir-in-results-dir

## Purpose

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`

## Related Examples

- "Run Analysis from the Command Line"

## Purpose

Display Polyspace version number

## Syntax

```
-v
-version
```

## Description

`-v` or `-version` displays the version number of your Polyspace product.

## Examples

Display the version number of your Polyspace product.

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

produces output such as:

```
Polyspace Bug Finder 1.0 (R2013b)
Copyright 1990-2013 The Mathworks, Inc.
```

## See Also

`polyspaceBugFinder`

## Related Examples

- "Run Analysis from the Command Line"

# -Wall

## Purpose

Print compilation warnings

## Syntax

```
-Wall
```

## Description

By default, the compilation phase prints only warnings about compliance across different files. Similar to the C compiler option, the `-Wall` option outputs warnings from the compilation phase.

## Limitations

Specifying the `-jsf-coding-rules` option disables the `-Wall` option.

## Examples

Use the `-Wall` flag to print compilation warnings.

```
polyspace-bug-finder-nodesktop -Wall ...
```

## See Also

`polyspaceBugFinder`

## Related Examples

- "Run Analysis from the Command Line"

4

# Checks

# Assertion

## Purpose

Failed assertion statement

## Description

**Assertion** occurs when you use an `assert`, and the asserted expression is or might 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" on page 1-43

## Concepts

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

# Invalid deletion of pointer

## Purpose

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 elements of the array `p` 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" on page 1-43

## Concepts

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

# Invalid use of == operator

## Purpose

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 equals 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
**Command-Line Syntax:** `bad_equal_equal_use`

## See Also

`Invalid use of = operator` | "Find defects" on page 1-43

## Concepts

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

# Invalid use of = operator

## Purpose

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 equals 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:** off
**Command-Line Syntax:** `bad_equal_use`

## See Also

`Invalid use of == operator` | "Find defects" on page 1-43

## Concepts

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

# Invalid use of floating point operation

## Purpose

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 might be imprecise.

## 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

One 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:** on
**Command-Line Syntax:** `bad_float_op`

## See Also

"Find defects" on page 1-43

## Concepts

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

# Invalid free of pointer

## Purpose

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" on page 1-43

## Concepts

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

# Code deactivated by constant false condition

## Purpose

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 portion within the directive is treated as a code comment 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

`Dead code` | "Find defects" on page 1-43

# Code deactivated by constant false condition

**Concepts**

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

## Purpose

Code does not execute

## Description

**Dead code** occurs when a block of code cannot be reached by execution path. This error excludes directives such as `#if 0`, which you can deliberately use to deactivate a code segment.

## Examples

### Dead code error

```
#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;

  /* Fix: Remove dead code */
  return table[ch];
}
```

## Check Information

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

## See Also

`Code deactivated by constant false condition` | "Find defects" on page 1-43

## Concepts

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

## Purpose

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

### Inconsistency Between Files

*file1.c*

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

*file2.c*

```
double foo(void);

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

#### Correction — Align the Function Declarations

One possible correction is to change the function declarations so 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();
}
```

# Declaration mismatch

**Check Information**

**Category:** Programming
**Language:** C | C++
**Default:** on
**Command-Line Syntax:** `decl_mismatch`

**See Also** "Find defects" on page 1-43

**Concepts**

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

# Deallocation of previously deallocated pointer

## Purpose

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;
```

# Deallocation of previously deallocated pointer

```
    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" on page 1-43

## Concepts

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

## Purpose

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 target operating system. See "Predefined Target Processor Specifications".

## 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 is converted to a variable of type float. However, the value 1^100 requires more than the 32-bits of a float 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" on page 1-43

## Concepts

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

# Float overflow

## Purpose

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 target operating system. See "Predefined Target Processor Specifications".

## 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 operation's result 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" on page 1-43

## Concepts

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

# Invalid use of standard library floating point routine

## Purpose

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" on page 1-43

## Concepts

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

# Float division by zero

## Purpose

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 an Integer 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" on page 1-43

## Concepts

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

# Use of previously freed pointer

## Purpose

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, the dereference of `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" on page 1-43

## Concepts

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

# Unreliable cast of function pointer

## Purpose

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" on page 1-43

## Concepts

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

## Purpose

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 operating system. See "Predefined Target Processor Specifications".

## 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, 1000000 cannot be represented by an 8-bit or 16-bit character 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_overflow`

# Integer conversion overflow

## See Also

`Float conversion overflow` | `Unsigned integer conversion overflow` | `Sign change integer conversion overflow` | "Find defects" on page 1-43

## Concepts

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

## Purpose

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 operating system. See "Predefined Target Processor Specifications".

## 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 one plus the maximum integer value cannot be represented by an `int`.

#### Correction — Different storage type

One possible correction is to change data types. Store the operation's result 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`

# Integer overflow

## See Also

`Unsigned integer overflow` | `Float overflow` | "Find defects" on page 1-43

## Concepts

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

# Invalid use of standard library integer routine

## Purpose

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;
```

# Invalid use of standard library integer routine

```
    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" on page 1-43

## Concepts

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

## Purpose

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.

# Integer division by 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" on page 1-43

## Concepts

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

## Purpose

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

The memory allocated through `malloc` and referenced by `pi` is neither freed nor returned by the function `assign_memory`.

```
#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 */
  }
```

#### 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" on page 1-43

## Concepts

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

# Invalid use of standard library memory routine

## Purpose

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 6 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" on page 1-43

## Concepts

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

# Missing null in string array

## Purpose

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
**Command-Line Syntax:** `missing_null_char`

## See Also

"Find defects" on page 1-43

## Concepts

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

# Missing or invalid return statement

## Purpose

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" on page 1-43

## Concepts

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

# Non-initialized pointer

## Purpose

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" on page 1-43

## Concepts

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

# Pointer to non-initialized value converted to const pointer

## Purpose

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" on page 1-43

## Concepts

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

# Non-initialized variable

## Purpose

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" on page 1-43

## Concepts

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

# Null pointer

## Purpose

`NULL` pointer derefenced

## 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" on page 1-43

## Concepts

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

# Arithmetic operation with NULL pointer

## Purpose

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" on page 1-43

## Concepts

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

# Invalid use of standard library routine

## Purpose

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” on page 1-43

**Concepts**

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

# Array access out of bounds

## Purpose

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" on page 1-43

## Concepts

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

# Pointer access out of bounds

## Purpose

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" on page 1-43

## Concepts

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

# Partially accessed array

## Purpose

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:** off
**Command-Line Syntax:** `partially_accessed_array`

## See Also

"Find defects" on page 1-43

## Concepts

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

# Large pass-by-value argument

## Purpose

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" on page 1-43

## Concepts

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

# Unreliable cast of pointer

## Purpose

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" on page 1-43

## Concepts

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

# Wrong type used in sizeof

## Purpose

`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
**Command-Line Syntax:** `ptr_sizeof_mismatch`

**See Also** "Find defects" on page 1-43

**Concepts**

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

# Qualifier removed in conversion

## Purpose

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" on page 1-43

## Concepts

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

# Race conditions

## Purpose

Race conditions between multiple instances of the same variable

## Description

**Race conditions** occur in multitasking code when two parallel task change the same variable. A race condition occurs because both tasks are racing to be the first to use the variable.

You must use the **Multitasking** analysis option and specify multiple entry points to find **Race condition** defects.

## Example

### Simple Function Race

1. Create a C file with the following text:

```
int var_for_rc;
void race_condition(void) {
    var_for_rc++;
}
void task1(void) { race_condition(); }
void task2(void) { race_condition(); }
```

2. In Bug Finder, create a project with your new source file.

3. In the **Multitasking** pane of the configuration options, select the **Multitasking** option.

4. In the **Entry points** portion of the same pane, use the  button to add two entry point rows, one for `task1` and the other for `task2`.

5. In the **Bug Finder Analysis** pane of the configuration options, select **Find defects > All**.

6. Run the analysis.

   The software finds one defect:

```
int var_for_rc;
void race_condition(void) {
    var_for_rc++;
```

```
}
void task1(void) { race_condition(); }
void task2(void) { race_condition(); }
```

In this example, the tasks `task1` and `task2` both call the same function which uses an external variable `var_for_rc`. A race condition occurs because `var_for_rc` is changing in two parallel tasks.

**Correction — Exclusive Tasks**

One possible correction is to change which tasks are parallel and which are temporally exclusive. Set `task1` and `task2` to be nonparallel multitasking tasks.

1

Follow the previous steps to create your project and set up the multitasking options.

2

From the Project Manager perspective, select the **Multitasking** pane.

3

In the **Temporally exclusive tasks** section, use the  button to add one row. List both tasks in the same line: `task1 task2`.

4

Rerun the analysis. The software does not find a race condition defect.

## Check Information

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

## See Also

"Multitasking" | "Find defects" on page 1-43

# Race conditions

**Concepts**

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

## Purpose

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” on page 1-43

# Shift of a negative value

**Concepts**

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

## Purpose

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 operating system. See "Predefined Target Processor Specifications".

## 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" on page 1-43

# Shift operation overflow

**Concepts**

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

## Purpose

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 operating system. See “Predefined Target Processor Specifications”.

## 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`

# Sign change integer conversion overflow

## See Also

`Float conversion overflow` | `Unsigned integer conversion overflow` | `Integer conversion overflow` | “Find defects” on page 1-43

## Concepts

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

# Invalid use of standard library string routine

## Purpose

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);
```

# Invalid use of standard library string routine

```
  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" on page 1-43

## Concepts

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

# Format string specifiers and arguments mismatch

## Purpose

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" on page 1-43

## Concepts

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

## External Web Sites

- Standard library output functions

## Purpose

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 operating system. See "Predefined Target Processor Specifications".

## 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 possible 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++

# Unsigned integer conversion overflow

**Default:** on
**Command-Line Syntax:** `uint_conv_ovfl`

## See Also

`Float conversion overflow` | `Integer conversion overflow` | `Sign change integer conversion overflow` | "Find defects" on page 1-43

## Concepts

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

## Purpose

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 operating system. See "Predefined Target Processor Specifications".

## 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 possible 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;
}
```

# Unsigned integer overflow

**Check Information**

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

**See Also**

`Integer overflow` | `Float overflow` | "Find defects" on page 1-43

**Concepts**

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

## Purpose

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>
```

# Uncalled function

```
#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" on page 1-43

## Concepts

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

## Purpose

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" on page 1-43

## Concepts

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

## Purpose

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 assingment */
    printf('The value is %d', level)

}
```

The variable `level` is printed, reading the new value.

# Write without further read

**Check Information**

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

**See Also** "Find defects" on page 1-43

**Concepts**

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

## Purpose

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
**Command-Line Syntax:** `var_shadowing`

**See Also** "Find defects" on page 1-43

**Concepts**

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

## Purpose

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 a = 4; int b = 0; //defect

    return a*b;
}
```

In this example, `a` and `b` 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 statement in the loop

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 conditionals with more than one statement

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:** Yes
**Command-Line Syntax:** `-more-than-one-statement`

## See Also

"Find defects" on page 1-43

## Concepts

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

## Purpose

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. a pointer is converted to a different pointer, but the size of the allocated memory is not 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 multiplier 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 multiplier 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 multiplier 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” on page 1-43

# Wrong allocated object size for cast

**Concepts**

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

5

# Functions

# pslinkfun

## Purpose

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('queuemanager')
pslinkfun('stop')
```

## Description

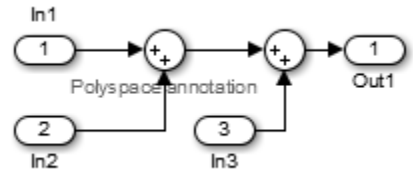
`pslinkfun('annotations','type',typeValue,'kind',kindValue,Name,Value)` adds an annotation of type `typeValue` and kind `kindValue` to the selected block in the model. You can specify a different block using a `Name,Value` pair argument. You can also add notes about a priority classification, an action 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 Manager perspective.

`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('queuemanager')` opens the Polyspace Queue Manager to display remote verifications in the queue.

`pslinkfun('stop')` kills the code analysis that is currently running. Use this option for local analyses only.

## 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.<br>For the list of possible checks, see: "Polyspace Bug Finder Defects". |
| `'MISRA-C'` | Use the rule number that you want to annotate. For example, `'2.2'`.<br>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'`.<br>For the list of supported MISRA C rules and their numbers, see "MISRA C:2004 Coding Rules". |
| `'MISRA-CPP'` | Use the rule number that you want to annotate. For example, `'0-1-1'`.<br>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'`.<br>For the list of supported JSF C++ rules and their numbers, see "JSF C++ Coding Rules". |

**Example:**

```
pslinkfun('annotation','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.`

## 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'`

## 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:

```
open(WhereAreTheErrors_v2);
```

Add an annotation to the switch block to annotate violations to MISRA C rule 13.7. Add to the annotation a comment, a classification, and a status.

```
pslinkfun('annotation','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.

At the command line, generate code for the model and run an analysis. After the analysis is finished, open the results in the Polyspace environment:

```
slbuild('WhereAreTheErrors_v2');
pslinkrun('WhereAreTheErrors_v2');
pslinkfun('openresults','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 they are from the switch block.

### Add Remote Verification Options to 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 Queue Manger to monitor your remote job.

```
pslinkrun(model,opts);
pslinkfun('queuemanager');
```

After your job is finished, open the metrics server to see your job in the repository.

```
pslinkfun('metrics');
```

## See Also

`pslinkrun` | `pslinkoptions` | `gcb`

# pslinkoptions

## Purpose

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 `model`.

## 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 configuration options specific to that code generator.

For a description of all configuration options and their values, see Configuration Options on page 5-11.

**Example:** `embedded_coder_opt = pslinkoptions('ec')`

**Example:** `target_link_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 Configuration Options on page 5-11.

**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 a Polyspace from the MATLAB command line.

**Example:**
```
myOptions = pslinkoptions('ec')
myOptions.VerificationSettings = 'Misra'
```

The following table provides possible values and a description for each configuration option. Depending on the code generator, the object will have different configuration options. The value in curly brackets {} is the default.

### Configuration Options

| Configuration Option | Values | Description |
|---|---|---|
| `ResultDir` | `{'C:\Polyspace_Results\results_$ModelName$'}` | Specify location of results folder. Can be either an absolute path or a path relative to the current folder. |
| `VerificationSettings` | `{'PrjConfig'} \| 'PrjConfigAndMisraAGC' \| 'PrjConfigAndMisra' \| 'MisraAGC' \| 'Misra'` | Specify checking of coding rules for C: **`'PrjConfig'`** – Inherit all options from project configuration and run complete analysis.<br>**`'PrjConfigAndMisraAGC'`** – Inherit all options from |

**Configuration Options (Continued)**

| Configuration Option | Values | Description |
|---|---|---|
| | | project configuration, enable MISRA AC AGC rule checking, and run complete analysis.<br>**`'PrjConfigAndMisra'`** – Inherit all options from project configuration, enable MISRA C rule checking, and run complete analysis.<br>**`'MisraAGC'`** – Enable MISRA AC AGC rule checking, and run compilation phase only.<br>**`'Misra'`** – Enable MISRA C rule checking, and run compilation phase only. |
| `OpenProjectManager` | `{false} \| true` | Open Polyspace Metrics or Project Manager to monitor the progress. Afterward, you can switch to the Results Manager perspective to review the results. |
| `AddSuffixToResultDir` | `{false} \| true` | Modify location of results folder by appending a unique number to the folder name instead of overwriting an existing folder. |

**Configuration Options (Continued)**

| Configuration Option | Values | Description |
|---|---|---|
| `EnableAdditionalFileList` | `{false} \| true` | Specify whether additional files must be analyzed. You can specify these additional files with the `AdditionalFileList` option |
| `AdditionalFileList` | `{0x1 cell}` | List additional files to analyze. |
| `InputRangeMode` | `{'DesignMinMax'} \| 'FullRange'` | Specify whether to use data ranges defined in blocks and workspace or treat inputs as full-range values. |
| `ParamRangeMode` | `{'None'} \| 'DesignMinMax'` | Specify whether to use constant values of parameters specified in the code, or use a range defined in blocks and workspace. |
| `OutputRangeMode` | `{'None'} \| 'DesignMinMax'` | Specify whether to apply assertions to outputs (using a range defined in blocks and workspace). |
| `VerificationMode` | `{'BugFinder'} \| 'CodeProver'` | Specify whether to run a Bug Finder analysis or Code Prover verification. |

### Configuration Options (Continued)

| Configuration Option | Values | Description |
|---|---|---|
| `AutoStubLUT`<br>*Only for TargetLink* | `{false} \| true` | Specify whether to include Lookup Table code in the analysis. |
| `ModelRefVerifDepth`<br>*Only for Embedded Coder* | `{'Current model only'} \| '1' \| '2' \| '3' \| 'All'` | Specify analysis of generated code with respect to model reference hierarchy levels. |
| `ModelRefByModelRefVerif`<br>*Only for Embedded Coder* | `{false} \| true` | Specify whether to analyze code from models within model reference hierarchies jointly or separately. |

**Configuration Options (Continued)**

| Configuration Option | Values | Description |
|---|---|---|
| `CxxVerificationSettings`<br>*Only for Embedded Coder* | `{'PrjConfig'} | 'PrjConfigAndMisraCxx' | 'PrjConfigAndJSF' | 'MisraCxx' | 'JSF'` | Specify checking of coding rules for C++: **`'PrjConfig'`** – Inherit all options from project configuration and run complete analysis.<br>**`'PrjConfigAndMisraCxx'`** – Inherit all options from project configuration, enable MISRA C++ rule checking, and run complete analysis.<br>**`'PrjConfigAndJSF'`** – Inherit all options from project configuration, enable JSF rule checking, and run complete analysis.<br>**`'MisraCxx'`** – Enable MISRA C++ rule checking, and run compilation phase only.<br>**`'JSF'`** – Enable JSF rule checking, and run compilation phase only. |
| `CheckConfigBeforeAnalysis` | `'Off' | {'OnWarn'} | 'OnHalt'` | Select the level of configuration checking before running the verification:**`'Off'`** — |

**Configuration Options (Continued)**

| Configuration Option | Values | Description |
|---|---|---|
| | | Check only for errors. Halts if errors are found<br>`'OnWarn'` — Halt for errors, display a message for warnings<br>`'OnHalt'` — Halt for errors and warnings. |

## Examples

### Use a Simulink model to create and edit an options objects

Load the Simulink model `psdemo_model_link_sl`:

```
load_system('psdemo_model_link_sl_v2')
```

From the MATLAB command line, create a Polyspace options object from the model:

```
model_opt = pslinkoptions('psdemo_model_link_sl_v2')
```

```
model_opt =

                    ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfig'
          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'
CheckConfigBeforeAnalysis: 'OnWarn'
```

The model is already configured for Embedded Coder, so only the Embedded Coder configuration options appear.

Change the results folder name option:

```
model_opt.ResultDir = 'results_v1_$ModelName$';
```

```
model_opt =

                   ResultDir: 'results_v1_$ModelName$'
        VerificationSettings: 'PrjConfig'
          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'
CheckConfigBeforeAnalysis: 'OnWarn'
```

Set the `OpenProjectManager` to true, to monitor progress in the Polyspace interface.

```
model_opt.OpenProjectManager = true
```

```
model_opt =

                   ResultDir: 'results_v1_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 1
```

```
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'BugFinder'
          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: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'BugFinder'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
     CxxVerificationSettings: 'PrjConfig'
CheckConfigBeforeAnalysis: 'OnWarn'
```

Set the `OpenProjectManager` option to true to follow the progress in the Polyspace interface:

```
new_opt.OpenProjectManager = true

new_opt =

                   ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfig'
          OpenProjectManager: 1
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
            VerificationMode: 'BugFinder'
          ModelRefVerifDepth: 'Current model only'
     ModelRefByModelRefVerif: 0
     CxxVerificationSettings: 'PrjConfig'
CheckConfigBeforeAnalysis: 'OnWarn'
```

Change the configuration to check for both run-time errors and MISRA C coding rule violations:

```
new_opt.VerificationSettings = 'PrjConfigAndMisra'

new_opt =

                   ResultDir: 'results_$ModelName$'
        VerificationSettings: 'PrjConfigAndMisra'
          OpenProjectManager: 1
        AddSuffixToResultDir: 0
    EnableAdditionalFileList: 0
          AdditionalFileList: {0x1 cell}
              InputRangeMode: 'DesignMinMax'
              ParamRangeMode: 'None'
             OutputRangeMode: 'None'
```

```
                  VerificationMode: 'BugFinder'
                ModelRefVerifDepth: 'Current model only'
          ModelRefByModelRefVerif: 0
          CxxVerificationSettings: 'PrjConfig'
     CheckConfigBeforeAnalysis: 'OnWarn'
```

## See Also

pslinkfun | pslinkoptions | pslinkrun

## Purpose

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.

## 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`

## 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.

## See Also

`pslinkfun` | `pslinkoptions`

## Purpose

Run Polyspace Bug Finder analysis from MATLAB

## Syntax

```
polyspaceBugFinder

polyspaceBugFinder(projectFile)
polyspaceBugFinder(resultsFile)
polyspaceBugFinder('-results-dir',resultsFolder)

polyspaceBugFinder('-help')

polyspaceBugFinder('-sources',sourceFile1,...,sourceFileN)
polyspaceBugFinder('-sources',sourceFile1,...,sourceFileN,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',sourceFile1,...,sourceFileN)` runs a Polyspace Bug Finder analysis on `sourceFile1,...,sourceFileN`.

`polyspaceBugFinder('-sources',sourceFile1,...,sourceFileN,Name,Value)` runs a Polyspace Bug Finder analysis on the source files with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### projectFile - Project file path

full path to project file with extension `.psprj` | relative path to project file with extension `.psprj`

Path to project file with extension `.psprj`, specified as a string.

If you use just the file name, the project file must reside in the current folder. Use `pwd` to identify the current folder and `cd` to change the current folder.

**Example:** `'C:\Polyspace_Projects\myProject.psprj'`

### resultsFile - Result file path

full path to results file with extension `.psbf` | relative path to results file with extension `.psbf`

Path to results file with extension `.pscp`, specified as a string.

If you use just the file name, the results file must reside in the current folder. Use `pwd` to identify the current folder and `cd` to change the current folder.

**Example:** `'myResults.psbf'`

### resultsFolder - Result folder name

full path to folder | relative path from current folder

Path to 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.

You can specify either the full path to the folder or the relative path from the current folder. Use `pwd` to identify the current folder and `cd` to change the current folder.

**Example:** `'C:\Polyspace\Results\'`

#### sourceFile1,...,sourceFileN - Source file name

path to source files with extension `.c` | path to source files with extension `.cpp`

Comma-separated source file paths with extension `.c` or `.cpp`, specified as a single string.

If you use just the file names, the source files must reside in the current folder. 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".

## 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 5-27. Here you specify checking of MISRA C coding rules using the option `-misra2`. For more information on this option, see "Check MISRA C:2004 rules" on page 1-34.

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)
```

## Related Examples

- "Specify Options from MATLAB Command Line"

# polyspaceConfigure

## Purpose

Create Polyspace project from your build system

## 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`.

## 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`

### -option value - Options for changing default operation of `polyspaceConfigure`

single option starting with -, followed by argument | multiple space-separated option-argument pairs

| Option | Argument | Description |
|---|---|---|
| `-author` | Author name | Name of project author.<br>**Example:** `-author jsmith` |
| `-build-trace` | Path and file name | Location and name of file where build information is stored. The default is `./polyspace_configure_build_trace.log`.<br>**Example:** `-build-trace ../build_info/trace.log` |

| Option | Argument | Description |
|---|---|---|
| `-cache-all-files` | None | Option to cache all files read by `polyspaceConfigure` including binaries |
| `-cache-path` | Path | Location of folder where cache information is stored.<br>**Example:** `-cache-path ../cache` |
| `-compiler-configuration` | Path and file name | Location and name of compiler configuration file.<br>The file must be in a specific format. For guidance, see the existing configuration files in *`matlabroot`*`\polyspace\configure\compiler_configuration\`.<br>**Example:** `-compiler-configuration myCompiler.xml` |
| `-debug` | None | Option used by MathWorks technical support |
| `-help` | None | Option to display the full list of `polyspaceConfigure` commands |
| `-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.<br>To use this option, you must have the build trace information saved from an earlier run of `polyspaceConfigure` with the `-no-project` option. |
| `-no-cache` | None | Option to specify that a cache of your files must not be created. |

| Option | Argument | Description |
|---|---|---|
| `-no-project` | None | Option to trace your build system without creating a Polyspace project and save the build trace information.<br>Use this option to save your build trace information for a later run of `polyspaceConfigure` with the `-no-build` option. |
| `-output-dump-file` | None | Option to save build trace information in a text file. |
| `-output-options-file` | None | Option to create a Polyspace analysis options file. Use this file for command-line analysis using `polyspaceBugFinder`. |
| `-output-project` | Path | Location for saving Polyspace project. The default is the current folder.<br>**Example:** `-output-project ../myProjects/` |
| `-prog` | Project name | Name of project. The default is `polyspace.psprj`.<br>**Example:** `-output-project myProject` |

## 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` 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` 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` option with `make` so that all prerequisite targets in the makefile are remade.
- 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` 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.

## Related Examples

- "Create Projects Automatically from Your Build System"

## Concepts

- "Requirements for Project Creation from Build Systems"

## Purpose

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.

## 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'`

## 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 preform 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 CLUSTE
...
19 user Polyspace C:\psdemo\res1  queued Wed Mar 16 16:48:38 EST 2014
20 user Polyspace C:\psdemo\res2  queued Wed Mar 16 16:48:38 EST 2014
```

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_M
...
19 user Polyspace C:\psdemo\res1  cancelled Wed Mar 16 16:48:38 EST 2014
20 user Polyspace C:\psdemo\res2  running Wed Mar 16 16:48:38 EST 2014 C
```

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_M
...
20 user Polyspace C:\psdemo\res2  completed Wed Mar 16 16:48:38 EST 2014
```

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');
```

## See Also

`polyspaceBugFinder`

## Concepts

- “Clusters and Cluster Profiles”
- “Manage Remote Analyses at the Command Line”

## Purpose

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.

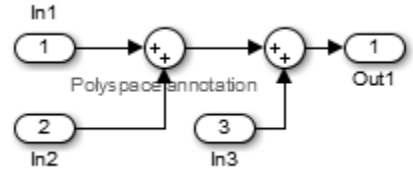
In the generated code associated with the annotated block, code comments are added before and after the lines of code. Polyspace reads these comments and marks Polyspace results of the specified `kind` with the annotated information.

When you add annotations, you can identify known errors and coding rule violations to focus on new results.

## 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

## 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'.<br>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'.<br>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'.<br>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'`

## 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
```

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.

## See Also

`pslinkoptions` | `pslinkrun` | `PolySpaceViewer` | `gcb`

# PolySpaceViewer

## Purpose

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 Manager perspective.

## Input Arguments

### system - Simulink model

system | subsystem

Simulink model specified by the system or subsystem name.

**Example:** `PolySpaceViewer(`myModel')`

## 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 Manager page 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 environment:

```
PolySpaceViewer('WhereAreTheErrors_v2');
```

The analysis results of `WhereAreTheErrors_v2` appear in the Polyspace Results Manager.

## See Also

`pslinkoptions` | `pslinkrun` | `PolyspaceAnnotation`