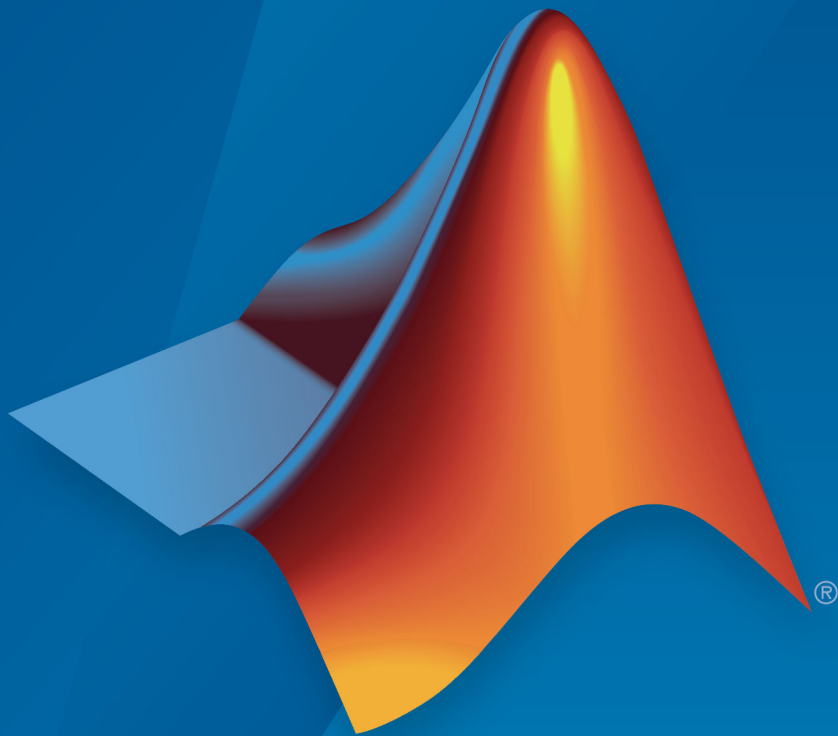


Polyspace[®] Bug Finder[™] Server[™]

User's Guide



R2019a

How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Polyspace® Bug Finder™ Server™ User's Guide

© COPYRIGHT 2019 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

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

March 2019 Online only New for Version 3.0 (R2019a)

1 Polyspace Analysis on Server After Code Submission

Prepare Scripts for Polyspace Analysis	1-2
Options Related to Source Code and Target	1-3
Options Related to Polyspace Analysis	1-5
Configure Polyspace Analysis Options in User Interface and Generate Scripts	1-7
Prerequisites	1-9
Generate Scripts from Configuration	1-9
Run Analysis with Generated Scripts	1-10
Sample Scripts for Polyspace Analysis with Jenkins	1-13
Extending Sample Scripts to Your Development Process	1-14
Prerequisites	1-15
Set Up Polyspace Plugin in Jenkins	1-15
Script to Run Bug Finder, Upload Results and Send Common Notification	1-20
Script to Run Bug Finder, Upload Results and Send Personalized Notification	1-22

Use Existing Software Development Specifications for Polyspace Analysis

2 Create Polyspace Analysis Configuration from Build Command	2-2
polyspace-configure Source Files Selection Syntax	2-5

Modularize Polyspace Analysis by Using Build Command	2-8
Build Source Code	2-8
Create One Polyspace Options File for Full Build	2-10
Create Options File for Specific Binary in Build Command . .	2-11
Create One Options File Per Binary Created in Build Command	2-12

Offload Polyspace Analysis to Remote Servers from Desktop

3

Send Polyspace Analysis from Desktop to Remote Servers . . .	3-2
Client-Server Workflow for Running Analysis	3-2
Prerequisites	3-4
Offload Analysis in Polyspace User Interface	3-4

Send Polyspace Analysis from Desktop to Remote Servers Using Scripts	3-7
Client-Server Workflow for Running Analysis	3-7
Prerequisites	3-8
Run Remote Analysis	3-8
Manage Remote Analysis	3-10
Sample Scripts for Remote Analysis	3-12

Run Polyspace Analysis on Server with MATLAB Scripts

4

Integrate Polyspace Server Products with MATLAB and Simulink	4-2
Integrate Polyspace with MATLAB Installation from Same Release	4-2
Integrate Polyspace with MATLAB Installation from Different Release	4-3
Check Integration Between MATLAB and Polyspace	4-3
Run Polyspace Server Products with MATLAB Scripts	4-4

Configure Target and Compiler Options

5

Specify Target Environment and Compiler Behavior	5-2
Extract Options from Build Command	5-3
Specify Options Explicitly	5-4
C/C++ Language Standard Used in Polyspace Analysis	5-7
Supported Language Standards	5-7
Default Language Standard	5-7
C11 Language Elements Supported in Polyspace	5-10
C++11 Language Elements Supported in Polyspace	5-12
C++14 Language Elements Supported in Polyspace	5-15
Provide Standard Library Headers for Polyspace Analysis ...	5-19
Requirements for Project Creation from Build Systems	5-21
Compiler Requirements	5-21
Build Command Requirements	5-22
Supported Keil or IAR Language Extensions	5-24
Special Function Register Data Type	5-24
Keywords Removed During Preprocessing	5-25
Remove or Replace Keywords Before Compilation	5-26
Remove Unrecognized Keywords	5-26
Remove Unrecognized Function Attributes	5-28
Gather Compilation Options Efficiently	5-30

Configure Inputs and Stubbing Options

6

Specify External Constraints	6-2
Create Constraint Template	6-2

Create Constraint Template from Code Prover Analysis Results	6-4
Update Existing Template	6-5
Specify Constraints in Code	6-6
External Constraints for Polyspace Analysis	6-8
Constraint Specification Limitations	6-15
Constrain Global Variable Range	6-17
User Interface (Desktop Products Only)	6-17
Command Line	6-18
Constrain Function Inputs	6-20
User Interface (Desktop Products Only)	6-20
Command Line	6-21
XML File Format for Constraints	6-23
Syntax Description — XML Elements	6-23
Valid Modes and Default Values	6-28

Configure Multitasking Analysis

7

Analyze Multitasking Programs in Polyspace	7-2
Configure Analysis	7-3
Review Analysis Results	7-4
Auto-Detection of Thread Creation and Critical Section in Polyspace	7-6
Multitasking Routines that Polyspace Can Detect	7-6
Example of Automatic Thread Detection	7-8
Naming Convention for Automatically Detected Threads	7-12
Limitations of Automatic Thread Detection	7-13
Configuring Polyspace Multitasking Analysis Manually	7-19
Specify Options for Multitasking Analysis	7-19
Adapt Code for Code Prover Multitasking Analysis	7-20
Protections for Shared Variables in Multitasking Code	7-24
Detect Unprotected Access	7-24

Protect Using Critical Sections	7-25
Protect Using Temporally Exclusive Tasks	7-27
Protect Using Priorities	7-27
Protect By Disabling Interrupts	7-28
Define Atomic Operations in Multitasking Code	7-29
Nonatomic Operations	7-29
What Polyspace Considers as Nonatomic	7-29
Define Specific Operations as Atomic	7-30
Define Preemptable Interrupts and Nonpreemptable Tasks .	7-33
Emulating Task Priorities	7-33
Examples of Task Priorities	7-33
Further Explorations	7-35
Define Critical Sections with Functions That Take Arguments	
.....	7-36
Polyspace Assumption on Functions Defining Critical Sections	
.....	7-36
Adapt Polyspace Analysis to Lock and Unlock Functions with	
Arguments	7-37

Configure Coding Rules Checking and Code Metrics Computation

8

Check for Coding Standard Violations	8-2
Configure Coding Rules Checking	8-2
Review Coding Rule Violations	8-7
Generate Reports	8-9
Avoid Violations of MISRA C 2012 Rules 8.x	8-10
Software Quality Objective Subsets (C:2004)	8-14
Rules in SQO-Subset1	8-14
Rules in SQO-Subset2	8-15
Software Quality Objective Subsets (AC AGC)	8-20
Rules in SQO-Subset1	8-20
Rules in SQO-Subset2	8-21

Software Quality Objective Subsets (C:2012)	8-24
Guidelines in SQO-Subset1	8-24
Guidelines in SQO-Subset2	8-25
Software Quality Objective Subsets (C++)	8-29
SQO Subset 1 - Direct Impact on Selectivity	8-29
SQO Subset 2 - Indirect Impact on Selectivity	8-31
Coding Rule Subsets Checked Early in Analysis	8-36
MISRA C: 2004 and MISRA AC AGC Rules	8-36
MISRA C: 2012 Rules	8-46
Create Custom Coding Rules	8-56
User Interface (Desktop Products Only)	8-56
Command Line	8-57
Compute Code Complexity Metrics	8-59
Impose Limits on Metrics (Desktop Products Only)	8-59
Impose Limits on Metrics (Server and Access products)	8-61
HIS Code Complexity Metrics	8-62
Project	8-62
File	8-62
Function	8-62

Configure Bug Finder Checkers

9

Choose Specific Bug Finder Defect Checkers	9-2
User Interface (Desktop Products Only)	9-2
Command Line	9-2
Short Names of Bug Finder Defect Checkers	9-4
Bug Finder Defect Groups	9-21
Concurrency	9-21
Cryptography	9-22
Data flow	9-22
Dynamic Memory	9-23
Good Practice	9-23

Numerical	9-23
Object Oriented	9-24
Programming	9-24
Resource Management	9-24
Static Memory	9-25
Security	9-25
Tainted data	9-25
Polyspace Bug Finder Defects Checkers Enabled by Default	9-27
Bug Finder Results Found in Fast Analysis Mode	9-33
Polyspace Bug Finder Defects	9-33
MISRA C: 2004 and MISRA AC AGC Rules	9-37
MISRA C: 2012 Rules	9-45
MISRA C++ 2008 Rules	9-53
CWE Coding Standard and Polyspace Results	9-66
CWE and Polyspace Bug Finder	9-66
Find CWE IDs from Polyspace Results	9-66
Mapping Between CWE Identifiers and Polyspace Results ...	9-67
Mapping Between CWE-658 or 659 and Polyspace Results	9-100
CWE-658: Weaknesses in Software Written in C	9-100
CWE-659: Weaknesses in Software Written in C++	9-109

10

Configure Comment Import from Previous Results

Import Comments from Previous Polyspace Analysis	10-2
Automatic Comment Import from Last Analysis	10-2
Import Comments from Another Analysis Result	10-3
Comment Import Algorithm	10-3
View Imported Comments That Do Not Apply	10-4
Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results	10-6
Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result	10-7

License Error -4,0	11-3
Issue	11-3
Possible Cause: Another Polyspace Instance Running	11-3
Possible Cause: Prior Polyspace Run in Simulink or MATLAB Coder	11-3
Read Error Information When Polyspace Analysis Stops	11-4
Contact Technical Support	11-5
Provide System Information	11-5
Provide Information About the Issue	11-5
Compiler Not Supported for Project Creation from Build Systems	11-8
Issue	11-8
Cause	11-8
Solution	11-8
Slow Build Process When Polyspace Traces the Build	11-18
Issue	11-18
Cause	11-18
Solution	11-18
Check if Polyspace Supports Build Scripts	11-19
Issue	11-19
Possible Cause	11-19
Solution	11-19
Troubleshooting Project Creation from MinGW Build	11-21
Issue	11-21
Cause	11-21
Solution	11-21
Troubleshooting Project Creation from Visual Studio Build	11-22
Cannot Create Project from Visual Studio Build	11-22
Compilation Error After Creating Project from Visual Studio Build	11-22

Error Processing Macro with Semicolon in Build System . .	11-24
Issue	11-24
Cause	11-24
Solution	11-24
Polyspace Cannot Find the Server	11-25
Message	11-25
Possible Cause	11-25
Solution	11-25
Job Manager Cannot Write to Database	11-26
Message	11-26
Possible Cause	11-26
Workaround	11-26
Undefined Identifier Error	11-28
Issue	11-28
Possible Cause: Missing Files	11-28
Possible Cause: Unrecognized Keyword	11-28
Possible Cause: Declaration Embedded in #ifdef Statements	11-29
Possible Cause: Project Created from Non-Debug Build . . .	11-30
Unknown Function Prototype Error	11-32
Issue	11-32
Cause	11-32
Solution	11-32
Error Related to #error Directive	11-34
Issue	11-34
Cause	11-34
Solution	11-34
Large Object Error	11-36
Issue	11-36
Cause	11-36
Solution	11-36
Errors Related to Generic Compiler	11-39
Issue	11-39
Cause	11-39
Solution	11-39

Errors Related to Keil or IAR Compiler	11-41
Missing Identifiers	11-41
Errors Related to Diab Compiler	11-42
Issue	11-42
Cause	11-42
Solution	11-42
Errors Related to Green Hills Compiler	11-45
Issue	11-45
Cause	11-45
Solution	11-46
Errors Related to TASKING Compiler	11-47
Issue	11-47
Cause	11-47
Solution	11-48
Errors from Conflicts with Polyspace Header Files	11-49
Issue	11-49
Cause	11-49
Solution	11-49
Errors from Assertion or Memory Allocation Functions ...	11-51
Issue	11-51
Cause	11-51
Solution	11-51
Error from Special Characters	11-52
Issue	11-52
Cause	11-52
Workaround	11-52
Errors from In-Class Initialization (C++)	11-53
Errors from Double Declarations of Standard Template Library Functions (C++)	11-54
Errors Related to GNU Compiler	11-55
Issue	11-55
Cause	11-55
Solution	11-55

Errors Related to Visual Compilers	11-56
Import Folder	11-56
pragma Pack	11-56
C++/CLI	11-57
Error or Slow Runs from Disk Defragmentation and Anti-virus	
Software	11-58
Issue	11-58
Possible Cause	11-58
Solution	11-58
SQLite I/O Error	11-60
Issue	11-60
Cause	11-60
Solution	11-60

Polyspace Analysis on Server After Code Submission

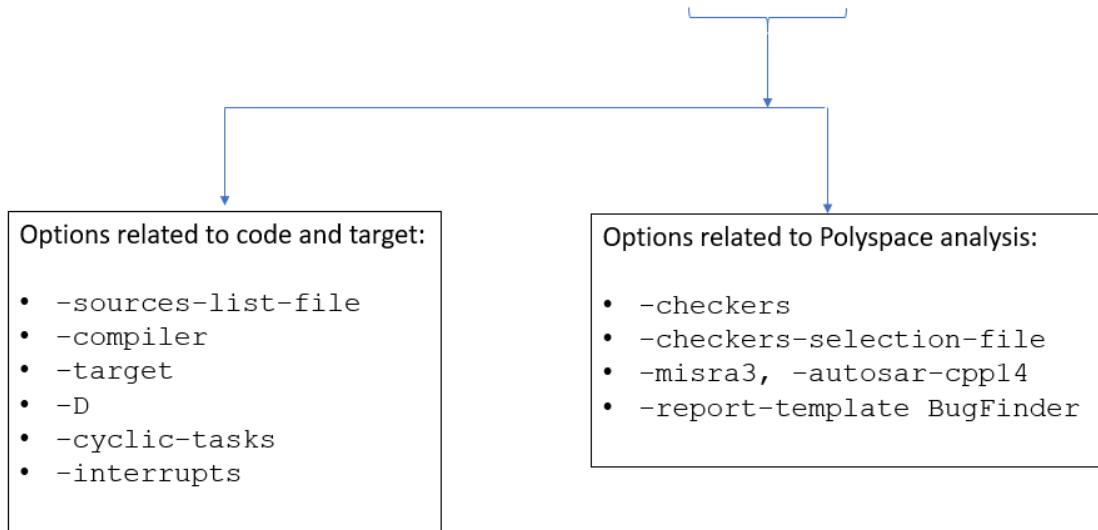
Prepare Scripts for Polyspace Analysis

When you run Polyspace as part of your software development processes, your analysis scripts must be preconfigured for new code submissions. For instance, new source files must be automatically included in the Polyspace analysis. To keep the analysis configuration updated with new submissions, you can leverage existing artifacts such as your build command (makefiles) and create your analysis configuration on the fly when new submissions occur.

The analysis configuration consists of two parts:

- Options related to the source code and target, such as data type sizes, macro definitions, cyclic tasks and interrupts, and so on.
- Options related to the analysis, such as checkers, code verification assumptions, and so on.

```
polyspace-bug-finder-server -options-file file.opts
```



Options Related to Source Code and Target

The most common options related to the source code and target are:

- `-sources-list-file`: Specify a text file containing one source file per line.
- `-I`: Specify the folders containing included header files.
- `Compiler (-compiler)`: Specify the compiler used for building your source code.
- `Target processor type (-target)`: Specify sizes of data types and endianness by selecting a predefined target processor.
- `Preprocessor definitions (-D)`: Replace unrecognized code for the purposes of Polyspace analysis. You typically use this option if the analysis shows compilation errors from compiler-specific keywords and macros.
- `Constraint setup (-data-range-specifications)`: Define external constraints on global variables and function interfaces. The option is typically useful for a more precise Code Prover analysis.

For the full list of options, see:

- “Analysis Options”
- “Analysis Options” (Polyspace Code Prover Server)

Extract Options from Build Command

In a continuous integration workflow, you typically do not specify the option arguments explicitly. Your build command contains the specifications for sources, compiler, macro definitions and so on. Run the `polyspace-configure` command to extract these specifications from your build command and create an options file. For instance, if you use `make` to build your source code, run the analysis as follows:

```
polyspace-configure -output-options-file polyspace_opts make
polyspace-bug-finder-server -options-file polyspace_opts
polyspace-code-prover-server -options-file polyspace_opts
```

The first command extracts source and target specifications by executing the instructions in the makefile and creates an analysis options file. The second and third commands runs a Bug Finder and Code Prover analysis with the options file. See “Create Polyspace Analysis Configuration from Build Command” on page 2-2.

Specify Options Explicitly in Options File

If you cannot extract the options from your build command, specify the options explicitly. You can create some of the option arguments on the fly from new submissions. For instance, the argument for the option `-sources-list-file` is a text file that lists the sources. You can update this text file based on any new source file added to the source code repository.

If you have to specify the target and compiler options explicitly, you might not get all the options right in the first run. To find the right combination of options:

- 1 Specify the options `Compiler` (`-compiler`) and `Target processor type` (`-target`) in your options file.
- 2 Compile the code with your compiler and fix all compilation errors. Then, run only the compilation part of the Polyspace analysis.
 - In Bug Finder, disable all checkers. Specify `-checkers none` in the options file. See `Find defects` (`-checkers`).
 - In Code Prover, stop the analysis after compilation. Specify `-to compile` in the options file. See `Verification level` (`-to`).

If you run into compilation errors, you might have to work around the errors with Polyspace options. For instance, if you see a compilation error because the macro `_WIN32` is defined with a compiler option but Polyspace considers the macro as undefined by default, emulate your compiler option with the Polyspace option `-D _WIN32`. See “Target and Compiler”, “Macros” and “Environment Settings” for the target and compiler options.

Once you fix all compilation errors with Polyspace analysis options, your options file is prepared with the right set of Polyspace options for the analysis.

If you have an installation of the desktop products, Polyspace Bug Finder and/or Polyspace Code Prover™, you can perform the trial runs in the user interface of the desktop products. You can then generate an options file from the configuration defined in the user interface. The user interface provides various features such as:

- Compilation assistant that suggests workarounds for some compilation errors,
- Auto-generation of XML file for constraint specification,

- Context-sensitive help for options,

See “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 1-7.

Options Related to Polyspace Analysis

Some options related to the Polyspace analysis are:

Bug Finder

- `Find defects (-checkers)`: Specify checkers to enable for the Bug Finder analysis.
- `Check MISRA C:2012 (-misra3)` and other options related to external standards: Specify an external standard and a predefined subset of that standard.
- `Set checkers by file (-checkers-selection-file)`: Specify a custom subset of rules from external standards.
- `Bug Finder and Code Prover report (-report-template)`: Specify that a PDF, Word or HTML report must be generated along with the analysis results and specify a template for the report.

Code Prover

- `Overflow mode for signed integer (-signed-integer-overflows)`: Specify the behavior following an overflow: stop analysis or continue with wrap-around.
- `Detect stack pointer dereference outside scope (-detect-pointer-escape)`: Specify if the analysis must find cases where a function returns a pointer to one of its local variables.
- `Detect uncalled functions (-uncalled-function-checks)`: Specify if the analysis must flag functions that are not called directly or indirectly from main or another entry point function.
- `Bug Finder and Code Prover report (-report-template)`: Specify that a PDF, Word or HTML report must be generated along with the analysis results and specify a template for the report.

The checkers and other options related to the Polyspace analysis can be applicable to more than one project. To maintain uniform standards across projects, you can reuse this

subset of analysis options. When running the analysis, specify two options files, one containing the options specific to the current project and the other containing the reusable options. You can extract the first options file from your build command but explicitly create the second options file.

For instance, in this example, the `polyspace-bug-finder-server` command uses two options files: `compile_opts` generated from a makefile and `runbf_opts` created manually. All reusable options can be specified in `runbf_opts`.

```
polyspace-configure -output-options-file compile_opts make
polyspace-bug-finder-server -options-file compile_opts -option-file runbf_opts
polyspace-code-prover-server -options-file compile_opts -option-file runcp_opts
```

If the same option appears in two options files, the last instance of the option is considered. In the preceding example, if an option occurs in both `compile_opts` and `runbf_opts`, the occurrence in `runbf_opts` is considered. If you want to override previous occurrences of an option, use an additional options file with your overrides. Append this options file to the end of the analysis command.

See Also

`polyspace-bug-finder-server` | `polyspace-configure`

More About

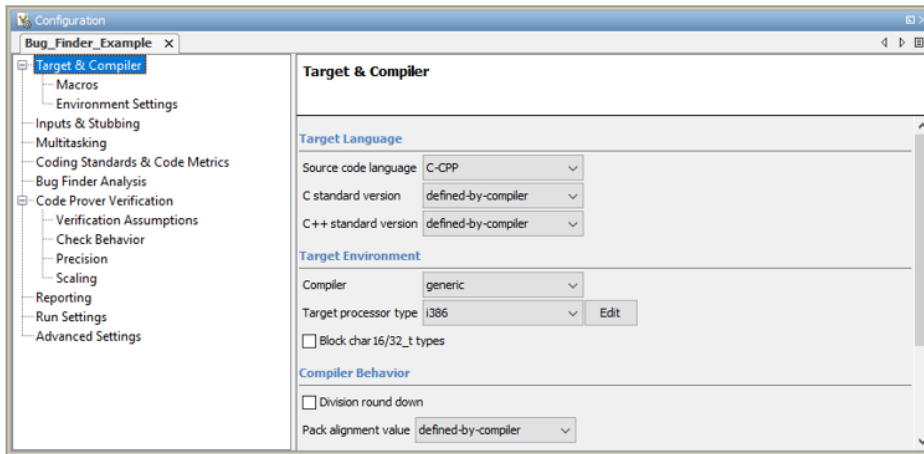
- “Run Polyspace Bug Finder on Server and Upload Results to Web Interface”
- “Create Polyspace Analysis Configuration from Build Command” on page 2-2
- “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 1-7

Configure Polyspace Analysis Options in User Interface and Generate Scripts

In this section...
“Prerequisites” on page 1-9
“Generate Scripts from Configuration” on page 1-9
“Run Analysis with Generated Scripts” on page 1-10

If you have an installation of the desktop products, Polyspace Bug Finder and/or Polyspace Code Prover, you can configure your project in the user interface of the desktop products. You can then generate a script or an options file from the configuration defined in the user interface and use the script or options file for automated runs with the desktop or server products.

1 Polyspace Analysis on Server After Code Submission



```
polyspace -generate-launching-script-for Bug_Finder_Example.psrj -bug-finder  
polyspace -generate-launching-script-for Code_Prover_Example.psrj
```

```
-target x86_64  
-c-version c11  
-compiler gnu4.6  
-dos  
-sources-list-file source_command.txt  
...
```

Unless you create a Polyspace project from existing specifications such as a build command, when setting up the project, you might have to perform a few trial runs first. In these trial runs, if you run into compilation errors or unchecked code, you might have to modify your analysis configuration. It is easier performing this initial setup in the user interface of the desktop products. The user interface provides various features such as:

- Compilation assistant that suggests workarounds for some compilation errors,
- Auto-generation of XML file for constraint specification,
- Context-sensitive help for options.

Prerequisites

You must have at least one license of Polyspace Bug Finder and/or Polyspace Code Prover to open the Polyspace user interface and configure the options.

After generating the scripts, you can run the analysis using either the desktop products (Polyspace Bug Finder and Polyspace Code Prover) or the server products (Polyspace Bug Finder Server and/or Polyspace Code Prover Server).

Generate Scripts from Configuration

This example shows how to generate a script from a Bug Finder configuration. The same steps apply to a Code Prover configuration.

- 1 Add source files to a new project in the Polyspace user interface.

Navigate to *polyspaceroot*\polyspace\bin, where *polyspaceroot* is the Polyspace installation folder; for instance, C:\Program Files\Polyspace\R2019a. Open the Polyspace user interface using the *polyspace* executable and create a new project.

See “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Bug Finder).

- 2 Specify the analysis options on the **Configuration** pane in the Polyspace project. To open this pane, in the project browser, click the configuration node in your Polyspace project.

See “Specify Polyspace Analysis Options” (Polyspace Bug Finder).

- 3 Run the analysis. Based on compilation errors and analysis results, modify options as needed.

See “Run Polyspace Analysis on Desktop” (Polyspace Bug Finder).

- 4 Once your analysis options are set, generate a script from the project (.psprj file).

To generate a script from the demo project, Bug_Finder_Example:

- a Load the project. Select **Help > Examples > Bug_Finder_Example.psprj**. A copy of this project is loaded in the Examples folder in your default workspace.

To find the project location, place your cursor on the project name in the **Project Browser** pane.

- b** Navigate to the project location and enter:

```
polyspace -generate-launching-script-for Bug_Finder_Example.psprj -bug-finder
```

To generate Code Prover scripts, use the same command without the `-bug-finder` option.

These files are generated for scripting the analysis:

- `source_command.txt`: Lists source files. This file can be provided as argument to the `-sources-list-file` option.
- `options_command.txt`: Lists analysis options. This file can be provided as argument to the `-options-file` option.
- `launchingCommand.bat` or `launchingCommand.sh`, depending on your operating system. The file uses the `polyspace-bug-finder` or `polyspace-code-prover` executable to run the analysis. The analysis runs on the source files listed in `source_command.txt` and uses the options listed in `options_command.txt`.

Run Analysis with Generated Scripts

After configuring your analysis and generating scripts, you can use the generated files to automate the subsequent analysis. You can automate the subsequent analysis using either the desktop or server products.

To automate a Bug Finder analysis with the desktop product, Polyspace Bug Finder:

- 1** Generate scripts as mentioned in the previous section.
- 2** Execute the script `launchingCommand.bat` or `launchingCommand.sh` at periodic intervals or based on predefined triggers.

To automate a Bug Finder analysis with the server product, Polyspace Bug Finder Server:

- 1** After specifying options in the user interface and before generating scripts, move the Polyspace project (`.psprj` file) to the server where the server product is running.
- 2** Generate scripts as mentioned in the previous section.

The scripts refer to the server product executable instead of the desktop products.

- 3 Execute the script `launchingCommand.bat` or `launchingCommand.sh` at periodic intervals or based on predefined triggers.

Alternatively, you can modify the script generated for the desktop product so that the server product is executed. The script refers to the path to a desktop product executable, for instance:

```
"C:\Program Files\Polyspace\R2019a\polyspace\bin\polyspace-code-prover.exe"
```

Replace this with the path to a server product executable, for instance:

```
"C:\Program Files\Polyspace Server\R2019a\polyspace\bin\  
polyspace-code-prover-server.exe"
```

Sometimes, you might want to override some of the options in the options file. For instance, the option to specify a results folder is hardcoded in the script. You can remove this option or override it when launching the scripts:

```
launchingCommand -results-dir newResultsFolder
```

where *newResultsFolder* is the new results folder. This folder can even be dynamically generated for each run.

If you override multiple options in `options_command.txt`, you can save the overrides in a second options file. Modify the script `launchingCommand.bat` or `launchingCommand.sh` so that both options files are used. The script uses the option `-options-file` to use an options file, for instance:

```
-options-file options_command.txt
```

If you place your option overrides in a second options file `overrides.txt`, modify the script to append a second `-options-file` option:

```
-options-file options_command.txt -options-file overrides.txt
```

See Also

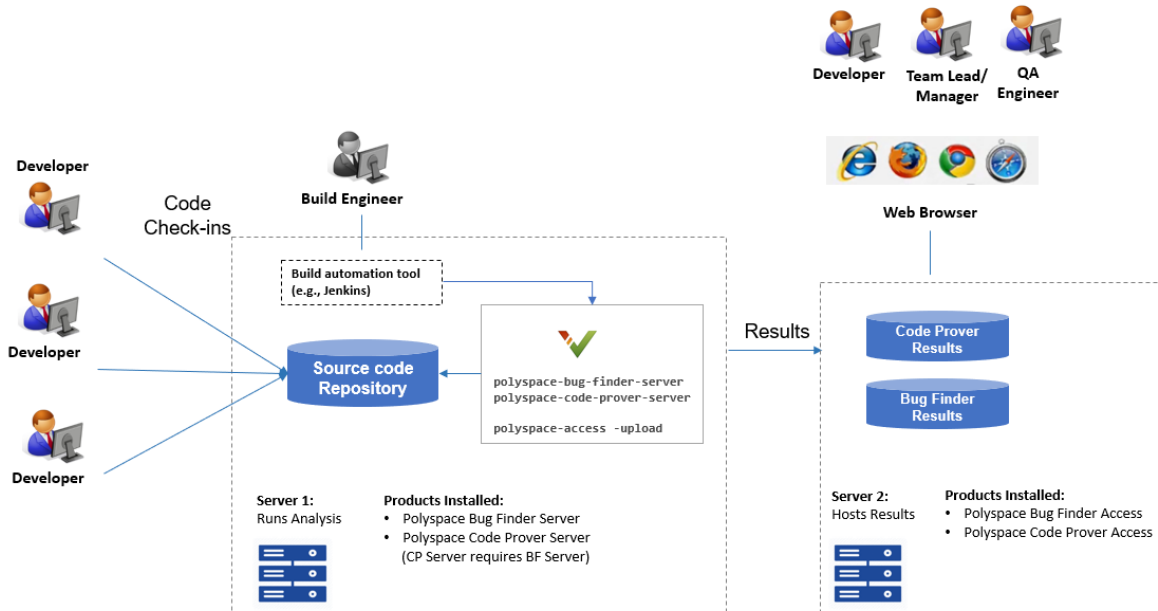
`-generate-launching-script-for`

Related Examples

- “Run Polyspace Bug Finder on Server and Upload Results to Web Interface”
- “Prepare Scripts for Polyspace Analysis” on page 1-2

Sample Scripts for Polyspace Analysis with Jenkins

In a continuous integration process, developers submit code to a shared repository. An automated build system using a tool such as Jenkins builds and tests each submission at regular intervals or based on predefined triggers and integrates the code. You can run a Polyspace analysis as part of this process.



Note:

- Depending on the specifications, the same computer can serve as both Server 1 and Server 2.
- Though a server hosts the components for Polyspace web interface, each reviewer requires a Polyspace (BF/CP) Access license to login to the interface.

This topic provides sample Shell scripts that run a Polyspace analysis using Polyspace Bug Finder Server and upload the results for review in the Polyspace Access web interface. The script also sends e-mail notifications to potential reviewers. Notified reviewers can login to the Polyspace Access web interface (if they have a Polyspace Bug Finder Access™ license) and review the results.

Extending Sample Scripts to Your Development Process

The scripts are written for a specific development toolchain but can be easily extended to the processes used in your project, team or organization.

In particular, the scripts:

- *Run on Linux[®] only.*

The scripts use some Linux-specific commands such as `export`. However, these commands are not an integral part of the Polyspace workflow. If you write Windows[®] scripts (`.bat` files), use the equivalent Windows commands instead.

- *Work only with Jenkins after you install the Polyspace plugin.*

The scripts are designed for the Jenkins plugin in these two ways:

- The scripts uses helper functions `$ps_helper` and `$ps_helper_access` for simpler scripting. The helper functions export Polyspace results for e-mail attachments and use command-line utilities to filter the results.

These helper functions are available only with the Jenkins plugin. However, the underlying commands come with a Polyspace Bug Finder Server installation. On build automation tools other than Jenkins, you can create these helper functions using the `polyspace-report-generator` command or `polyspace-access` command (with the `-export` option). See “Send E-mail Notifications with Polyspace Bug Finder Results”.

- The scripts create text files for e-mail attachments and mail subjects and bodies for personalized e-mails. If you install the Polyspace plugin in Jenkins, an extension of an e-mail plugin is available for use in your Jenkins projects. The e-mail plugin allows you to easily send the personalized e-mails with the previously created subjects, bodies and attachments. Without the Polyspace plugin, you have to find an alternative way to send the e-mails.
- *Run a Bug Finder analysis.*

The scripts run Bug Finder on the demo example `Bug_Finder_Example`. If you install the product Polyspace Bug Finder Server, the folder containing the demo example is `polyspaceserverroot/polyspace/examples/cxx/Bug_Finder_Example`. Here, `polyspaceserverroot` is the installation folder for Polyspace Server products, for instance, `/usr/local/Polyspace Server/R2019a/`.

You can easily adapt the script to run Code Prover. Replace `polyspace-bug-finder-server` with `polyspace-code-prover-server`. You can use the demo example `Code_Prover_Example` specifically meant for Code Prover.

Prerequisites

To run a Bug Finder analysis on a server and review the results in the Polyspace Access web interface, you must perform a one-time setup.

- To run the analysis, you must install one instance of the Polyspace Bug Finder Server product.
- To upload results, you must set up the components required to host the web interface of Polyspace Access.
- To view the uploaded results, you (and each developer reviewing the results) must have one Polyspace Bug Finder Access license.

See “Install Polyspace Server and Access Products”.

To install the Polyspace plugin, in the Jenkins interface, select **Manage Jenkins** on the left. Select **Manage Plugin**. Search for the Polyspace plugin and then download and install the plugin.

Set Up Polyspace Plugin in Jenkins

The following steps outline how to set up a Polyspace analysis in Jenkins after installing the Polyspace plugin. Note that the steps refer to Jenkins version 2.150.1. The steps in your Jenkins version and your Polyspace plugin installation might be slightly different.

If you use a different build automation tool, you can perform similar setup steps.

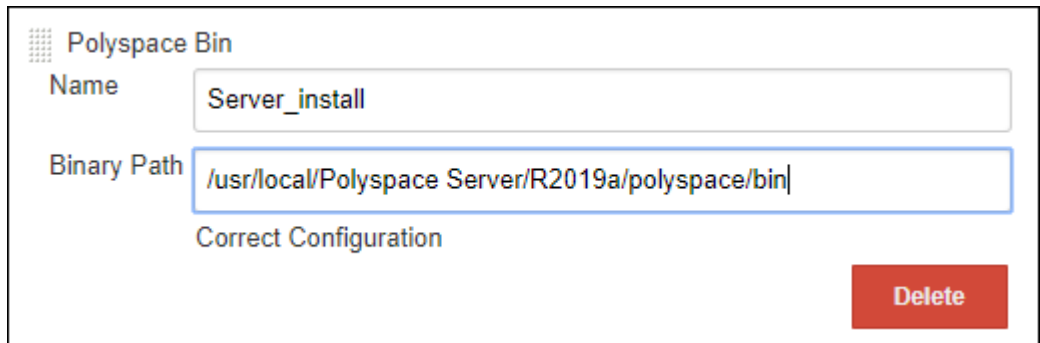
Specify Paths to Polyspace Commands and Server Details for Polyspace Access Web Interface

Specify the full paths of the folder containing the Polyspace commands and host name and port number of the server hosting the Polyspace Access web interface. After you

specify the paths, in your scripts, you do not have to use the full paths to the commands or the server details for uploading results.

- 1 In the Jenkins interface, select **Manage Jenkins** on the left. Select **Configure System**.
- 2 In the **Polyspace** section, specify the following:
 - Paths to Polyspace commands.

The path refers to *polyspaceserverroot/polyspace/bin*, where *polyspaceserverroot* is the installation folder for Polyspace Server products, for instance, */usr/local/Polyspace Server/R2019a/*.



The screenshot shows a configuration form titled "Polyspace Bin" with a grid icon on the left. It contains two input fields: "Name" with the value "Server_install" and "Binary Path" with the value "/usr/local/Polyspace Server/R2019a/polyspace/bin". Below the "Binary Path" field, the text "Correct Configuration" is displayed. A red "Delete" button is located in the bottom right corner of the form.

- The host name, port number and protocol (<http> or <https>) used by the server hosting the Polyspace Access web interface.



Polyspace Access

Name Polyspace_Access

Protocol https

Host doc-server

Port 9443

Delete

The **Name** field allows you to define a convenient shorthand that you use later in Jenkins projects.

- 3** In the **E-mail Notification** section, specify your company's SMTP server (and other details needed for sending e-mails).

E-mail Notification	
SMTP server	<input type="text" value="mail.companyname.com"/>
Default user e-mail suffix	<input type="text"/>
<input type="checkbox"/> Use SMTP Authentication	
Use SSL	<input type="checkbox"/>
SMTP Port	<input type="text" value="25"/>
Reply-To Address	<input type="text"/>
Charset	<input type="text" value="UTF-8"/>
<input type="checkbox"/> Test configuration by sending test e-mail	

Create Jenkins Project for Running Polyspace

When you create a Jenkins project (for instance, a Freestyle project), you can refer to the Polyspace paths by the global shorthands that you defined earlier.

To create a Jenkins project for running Polyspace:

- 1 In the Jenkins interface, select **New Item** on the left. Select **Freestyle Project**.
- 2 In the **Build Environment** section of the project, enter the two shorthand names you defined earlier:
 - The name for the path to the folder containing the Polyspace commands
 - The name for the details of the server hosting the Polyspace Access web interface.

Also, enter a login and password that can be used to upload to the Polyspace Access web interface. The login and password must be associated with a Polyspace Bug Finder Access license.

Build Environment

Polyspace - Configuration to use


Polyspace Bin Configuration ⓘ

Polyspace Access Configuration ⓘ

Polyspace Access Credentials Add ⓘ

- 3 In the **Build** section of the project, you can enter scripts that use the Polyspace commands and details of the server hosting the Polyspace Access web interface.

Build

 **Execute shell**

Command

```
set -e
export RESULT=ResultBF
export PROG=Bug_Finder_Example_2
export PARENT_PROJECT=testProject
rm -rf Notification && mkdir -p Notification

build_cmd="gcc -c sources/*.c"
polyspace-configure \
  -allow-overwrite \
  -allow-build-error \
  -prog $PROG \
  -author jenkins \
  -output-options-file $PROG.psopts \
  $build_cmd

polyspace-bug-finder-server -options-file $PROG.psopts -results-dir $RESULT
```

The scripts run a Polyspace analysis and upload results to the Polyspace Access web interface.

- 4 In the **Post-build Actions** section of the project, configure e-mail addresses and attachments to be sent after the analysis.

Post-build Actions

Polyspace Notification X

Send to Recipients ?

Recipients

File to attach

Mail Subject

Mail Body

Script to Run Bug Finder, Upload Results and Send Common Notification

This script runs a Bug Finder analysis, uploads the results and exports defects with high impact for a common notification email to all recipients.

The script assumes that the current folder contains a folder `sources` with `.c` files. Otherwise modify the line `gcc -c sources/*.c` with the full path to the sources.

```
set -e
export RESULT=ResultBF
export PROG=Bug_Finder_Example
export PARENT_PROJECT=/public/BugFinderExample_PRS_01

# =====
# Trace build command and create an options file

build_cmd="gcc -c sources/*.c"
polyspace-configure \
    -allow-overwrite \
    -allow-build-error \
    -prog $PROG \
    -author jenkins \
    -output-options-file $PROG.psopts \
    $build_cmd

# =====
# Run Bug Finder on the options file

polyspace-bug-finder-server -options-file $PROG.psopts -results-dir $RESULT

# =====
# Upload results to Polyspace Access web interface

$ps_helper_access -create-project $PARENT_PROJECT
$ps_helper_access \
    -upload $RESULT \
    -parent-project $PARENT_PROJECT \
    -project $PROG

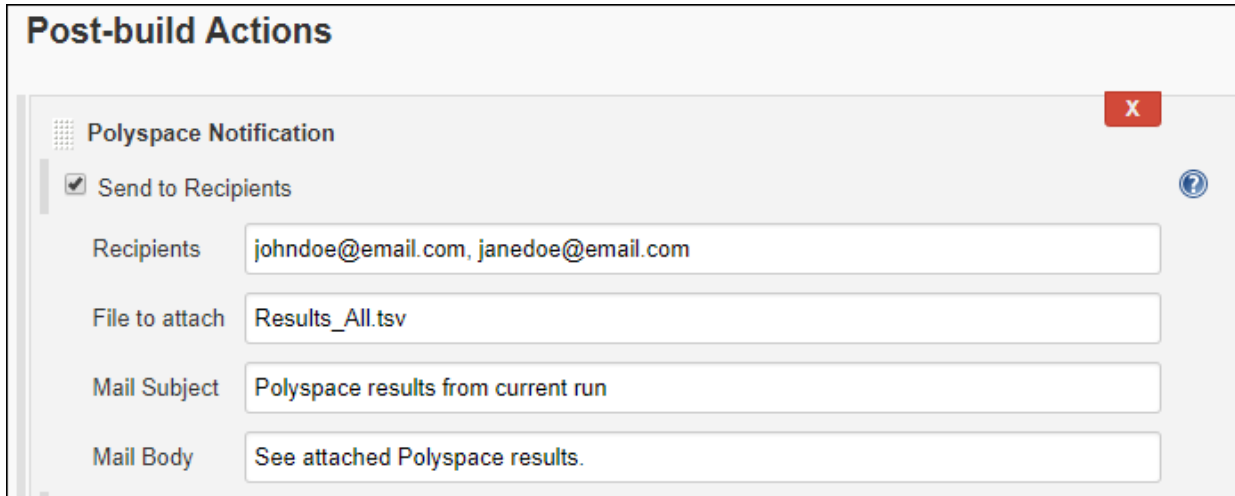
# =====
# Export results filtered for defects with "High" impact

$ps_helper_access \
    -export $PARENT_PROJECT/$PROG \
    -output Results_All.tsv \
    -defects High

# =====
# Finalize Jenkins status

exit 0
```

After the script is run, you can create a post-build action to send an e-mail to all recipients with the exported file `Results_All.tsv`.



The screenshot shows a configuration window titled "Post-build Actions" with a sub-section "Polyspace Notification". The "Send to Recipients" checkbox is checked. The "Recipients" field contains "johndoe@email.com, janedoe@email.com". The "File to attach" field contains "Results_All.tsv". The "Mail Subject" field contains "Polyspace results from current run". The "Mail Body" field contains "See attached Polyspace results.".

In this script, `$ps_helper_access` is a shorthand for the `polyspace-access` command with the options specifying host name, port, login and encrypted password included. The other `polyspace-access` options are explicitly written in the script.

Script to Run Bug Finder, Upload Results and Send Personalized Notification

This script runs the previous Bug Finder analysis and uploads the results. However, the script differs from the previous script in these ways:

- The script uses a `run_command` function that prints a message when running a command. The function helps determine from the console output which part of the script is running.

- When exporting the results, the script creates a separate results file for different owners.

- A master file `Results_All.tsv` contains all results. This file is sent in e-mail attachment to a manager. The manager email is configured in the post-build step.

If the file contains more than 10 defects, the build status is considered as a failure. The script sends a status `UNSTABLE` in the e-mail notification.

- The results file `Results_Users_userA.tsv` exported for `userA` contains defects from the group Programming and with impact High.

This result file is sent in e-mail attachment to `userA`.

- The results file `Results_Users_userB.tsv` exported for `userB` contains defects from the function `bug_memstdlib()`.

This result file is sent in e-mail attachment to `userB`.

- A separate mail subject is created for the manager in the file `mailsubject_manager.txt` and for users `userA` and `userB` in the files `mailsubject_user_userA.txt` and `mailsubject_user_userB.txt` respectively.

A mail body is created for the email to the manager in the file `mailbody_manager.txt`.

The script:

- Assumes that the current folder contains a folder `sources` with `.c` files.

Otherwise, modify the line `gcc -c sources/*.c` with the full path to the sources.

- Assumes users named `userA` and `userB`. In particular, the email addresses `userA@companyname.com` and `userB@companyname.com` (determined from the user name and SMTP server configured earlier) must be real e-mail addresses.

Replace the names with real user names.

```
set -e
export RESULT=ResultBF
export PROG=Bug_Finder_Example
export REPORT=Results_List.tsv

# =====
# Define function to print message while running command
run_command()
{
# $1 is a message
# $2 $3 ... is the command to dump and to run
message=$1
shift
cat >> mailbody_manager.txt << EOF
$(date): $message

EOF
"$@"
}

# =====
# Initialize mail body
cat > mailbody_manager.txt << EOF
Dear Manager(s)

Here is the report of the Jenkins Job ${JOB_NAME} #${BUILD_NUMBER}
It contains all Red Defect found in Bug Finder Example project

EOF

# =====
# Trace build command and create options file

build_cmd="gcc -c sources/*.c"
run_command "Tracing build command", \
    polyspace-configure \
    -allow-overwrite \
    -allow-build-error \
    -prog $PROG \
    -author jenkins \
    -output-options-file $PROG.psopts \
    $build_cmd
```

```
# =====
# Run Bug Finder on the options file

run_command "Running Bug finder" \
    polyspace-bug-finder-server -options-file $PROG.psopts\
    -results-dir $RESULT

# =====
# Upload results to Polyspace Access web interface

run_command "Creating Project $PARENT_PROJECT" \
    $ps_helper_access -create-project $PARENT_PROJECT

run_command "Uploading on $PARENT_PROJECT/$PROG" \
    $ps_helper_access \
    -upload $RESULT \
    -parent-project $PARENT_PROJECT \
    -project $PROG \
    -output upload.output
PROJECT_RUNID=$(($ps_helper prs_print_runid upload.output)
PROJECT_URL=$(($ps_helper prs_print_projecturl upload.output $POLYSPACE_ACCESS_URL)

# =====
# Export report

run_command "Exporting report from $PARENT_PROJECT/$PROG" \
    $ps_helper_access \
    -export $PROJECT_RUNID \
    -output $REPORT \
    -defects High

# =====
# Filter Reports

run_command "Filtering reports for defects" \
    $ps_helper report_filter \
    $REPORT \
    Results_All.tsv \
    Family Defect \
```

```
# =====
# Filter Reports for userA and userB

run_command "Filtering Reports for userA based on Group and Information" \
    $ps_helper report_filter \
        $REPORT \
        Results_Users.tsv \
        userA \
        Group Programming \
        Information "Impact: High"
run_command "Filtering Reports for userB based on Function" \
    $ps_helper report_filter \
        $REPORT \
        Results_Users.tsv \
        userB \
        Function "bug_memstdlib()"

# =====
# Update Jenkins status
# Jenkins build status is unstable when there are more than 10 Defects

BUILD_STATUS=$(ps_helper report_status Results_All.tsv 10)

# =====
# Update mail body and mail subject

NB_FINDINGS_ALL=$(ps_helper report_count_findings Results_All.tsv)
NB_FINDINGS_USERA=$(ps_helper report_count_findings Results_Users_userA.tsv)
NB_FINDINGS_USERB=$(ps_helper report_count_findings Results_Users_userB.tsv)
cat >> mailbody_manager.txt << EOF

Number of defects: $NB_FINDINGS_ALL
Number of findings owned by userA: $NB_FINDINGS_USERA
Number of findings owned by userB: $NB_FINDINGS_USERB

All results are uploaded in: $PROJECT_URL

Build Status: $BUILD_STATUS

EOF

cat >> mailsubject_manager.txt << EOF
Polyspace run completed with status $BUILD_STATUS and $NB_FINDINGS_ALL findings
```



```
EOF

for user in userA userB
do
echo "$user - $($ps_helper report_count_findings Results_Users_$user.tsv) findings\"
    > mailsubject_user_$user.txt
done

# =====
# Exit with correct build status

[ "$BUILD_STATUS" != "SUCCESS" ] && exit 129
exit 0
```

After the script is run, you can create a post-build action to send an e-mail to a manager with the exported file `Results_All.tsv`. Specify the e-mail address in the **Recipients** field, the email subject in the **Mail Subject** field and the email body in the **Mail Body** field.

In addition, a separate e-mail is sent to `userA` and `userB` with the files `Results_Users_userA.tsv` and `Results_Users_userB.tsv` in attachment (and the content of `mailsubject_user_userA.txt` and `mailsubject_user_userB.txt` as mail subjects). The e-mail addresses are `userA@companyname.com` and `userB@companyname.com` (determined from the user name and SMTP server configured earlier).

Post-build Actions

Polyspace Notification X

Send to Recipients ?

Recipients

File to attach

Mail Subject

Mail Body

Send to Owners ?

Query Base Name

Mail Subject Base Name

Mail Body Base Name

Unique recipients - Debug only

Add post-build action ▼

The script uses the helper function `$ps_helper` to filter the results based on group, impact and function. The helper function uses command-line utilities to filter the master file for results and perform actions such as create a separate results file for each owner. The function takes these actions as arguments:

- `report_filter`: Filters results from exported text file based on contents of the text file.

For instance:

```
$ps_helper report_filter \
    Results_List.tsv \
    Results_Users.tsv \
    userA \
    Group Programming \
    Information "Impact: High"
```

reads the file `Results_List.tsv` and writes to the file `Results_Users_userA.tsv`. The text file `Results_List.tsv` contains columns for `Group` and `Information`. Only those rows where the `Group` column contains `Programming` and the `Information` column contains `Impact: High` are written to the file `Results_Users_userA.tsv`.

- `report_status`: Returns `UNSTABLE` or `SUCCESS` based on the number of results in a file.

For instance:

```
BUILD_STATUS=$(ps_helper report_status Results_All.tsv 10)
```

returns `UNSTABLE` if the file `Results_All.tsv` contains more than 10 results (10 rows).

- `report_count_findings`: Reports number of results in a file.

For instance:

```
NB_FINDINGS_ALL=$(ps_helper report_count_findings Results_All.tsv)
```

returns the number of results (rows) in the file `Results_All.tsv`.

- `prs_print_projecturl`: Uses the host name and port number to create the URL of the Polyspace Access web interface.

For instance:

```
PROJECT_URL=$(ps_helper prs_print_projecturl Results_All.tsv $POLYSPACE_ACCESS_URL)
```

reads the file `Results_All.tsv` (exported by the `polyspace-access` command) and extracts the URL of the Polyspace Access web interface in `$POLYSPACE_ACCESS_URL` and the URL of the current project in `$PROJECT_URL`.

See Also

[polyspace-access](#) | [polyspace-bug-finder-server](#) | [polyspace-code-prover-server](#) | [polyspace-configure](#) | [polyspace-report-generator](#)

More About

- “Run Polyspace Bug Finder on Server and Upload Results to Web Interface”
- “Send E-mail Notifications with Polyspace Bug Finder Results”

Use Existing Software Development Specifications for Polyspace Analysis

- “Create Polyspace Analysis Configuration from Build Command” on page 2-2
- “polyspace-configure Source Files Selection Syntax” on page 2-5
- “Modularize Polyspace Analysis by Using Build Command” on page 2-8

Create Polyspace Analysis Configuration from Build Command

To run Polyspace on a server during continuous integration, you must configure all analysis options beforehand so that the analysis completes without errors. These options must be updated as necessary to keep up with new code submissions. If you use existing artifacts such as a build command (makefile) to build new code submissions, you can reuse the build command to configure a Polyspace analysis and stay updated with new submissions. With the `polyspace-configure` command, you can monitor the execution of a build command and create an options file for analysis with Polyspace.

This topic shows a simple tutorial illustrating how to create an options file from a build command and use the file for the subsequent analysis. The topic uses a Linux makefile and the GCC compiler, but you can adapt the commands to other operating systems such as Windows and other compilers such as Microsoft® Visual Studio®.

- 1 Copy the demo source files from `polyspaceserverroot\polyspace\examples\cxx\Bug_Finder_Example\sources` to a folder with write permissions. Here, `polyspaceserverroot` is the root installation folder of the Polyspace server products, for instance, `C:\Program Files\Polyspace Server\R2019a`.
- 2 Create a simple makefile that compiles the demo source files. Save the makefile in the same folder as the source files.

For instance, create a file named `makefile` and add this content:

```
CC := gcc
SOURCES := $(wildcard *.c)

all: $(CC) -c $(SOURCES)
```

Check that the makefile builds the source files successfully. Open a command terminal, navigate to the folder (using `cd`) and enter:

```
make
```

The `make` command should complete execution without errors.

- 3 Trace the build command with `polyspace-configure` and create an options file `compile_opts`.

```
polyspace-configure -output-options-file compile_opts make
```

- 4 Create a second options file with additional options. For instance, create a file `run_opts` with this content:

```
-checkers numerical  
-report-template BugFinder  
-output-format pdf
```

The options run all numerical checkers in Bug Finder and creates a PDF report after analysis using the `BugFinder` template.

- 5 Run a Bug Finder analysis with the two options files: `compile_opts` created from your build command and `run_opts` created manually.

```
polyspace-bug-finder-server -options-file compile_opts -options-file run_opts
```

The analysis should complete without errors. You can open the results in the Polyspace user interface or upload the results to the Polyspace Access web interface (using the `polyspace-access` command).

To run Code Prover instead of Bug Finder, use the `polyspace-code-prover-server` command instead of the `polyspace-bug-finder-server` command.

You can run a similar analysis using MATLAB scripts. Replace `polyspace-bug-finder-server` with the function `polyspaceBugFinderServer` and `polyspace-configure` with the function `polyspaceConfigure`.

See Also

`polyspace-bug-finder-server` | `polyspace-configure`

See Also

More About

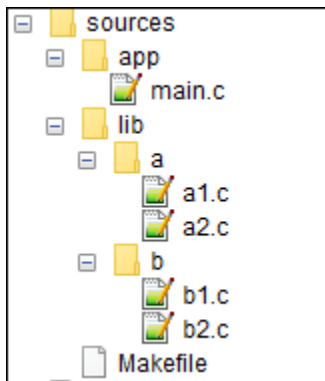
- “Prepare Scripts for Polyspace Analysis” on page 1-2
- “Specify Target Environment and Compiler Behavior” on page 5-2
- “polyspace-configure Source Files Selection Syntax” on page 2-5

- “Modularize Polyspace Analysis by Using Build Command” on page 2-8

polyspace-configure Source Files Selection Syntax

When you create projects by using `polyspace-configure`, you can include or exclude source files whose paths match the pattern that you pass to the options `-include-sources` or `-exclude-sources`. You can specify these two options multiple times and combine them at the command line.

This folder structure applies to these examples.



To try these examples, use the demo files in `polyspaceroot\help\toolbox\polyspace_bug_finder_server\examples\sources-select`. `polyspaceroot` is the Polyspace installation folder.

Run this command:

```
polyspace-configure -allow-overwrite -include-sources "glob_pattern" \
-print-excluded-sources -print-included-sources make -B
```

glob_pattern is the glob pattern that you use to match the paths of the files you want to include or exclude from your project. To ensure the shell does not expand the glob patterns you pass to `polyspace-configure`, enclose them in double quotes.

In the table, the examples assume that `sources` is a top-level folder.

Glob Pattern Syntax	Example
<p>No special characters, slashes ('/'), or backslashes ('\').</p> <p>Pattern matches corresponding files, but not folders.</p>	<p>-include-sources "main.c" matches: /sources/app/main.c</p>
<p>Pattern contains '*' or '?' special characters.</p> <p>'*' matches zero or more characters in file or folder name.</p> <p>'?' matches one character in file or folder name.</p> <p>The matches do not include path separators.</p>	<p>-include-sources "b?.c" matches: /sources/lib/b/b1.c /sources/lib/b/b2.c</p> <p>-include-sources "app/*.c" matches: /sources/app/main.c</p>
<p>Pattern starts with slash '/' (UNIX®) or drive letter (Windows).</p> <p>Pattern matches absolute path only.</p>	<p>-include-sources "/a" does not match anything.</p> <p>-include-sources "/sources/app" matches: /sources/app/main.c</p>
<p>Pattern ends with a slash (UNIX), backslash (Windows), or '**'.</p> <p>Pattern matches all files under specified folder.</p> <p>'**' is ignored if it is at the start of the pattern.</p>	<p>-include-sources "a/" matches /sources/lib/a/a1.c /sources/lib/a/a2.c</p>
<p>Pattern contains '**/' (UNIX) or '**\' (Windows). Pattern matches zero or more folders in the specified path.</p>	<p>-include-sources "lib/**/?1.c" matches: /sources/lib/a/a1.c /sources/lib/b/b1.c</p>

Glob Pattern Syntax	Example
<p>Pattern starts with '.' or '..'.</p> <p>Pattern matches paths relative to the path where you run the command.</p>	<p>If you start polyspace-configure from /sources/lib/a,</p> <pre>-include-sources "../lib/**/b?.c"</pre> <p>matches:</p> <pre>/sources/lib/b/b1.c</pre> <pre>/sources/lib/b/b2.c</pre>
<p>Pattern is a UNC path on Windows .</p>	<p>If your files are on server myServer:</p> <pre>\\myServer\sources\lib\b**</pre> <p>matches:</p> <pre>\\myServer\sources\lib\b\b1.c</pre> <pre>\\myServer\sources\lib\b\b2.c</pre>

polyspace-configure does not support these glob patterns:

- Absolute paths relative to the current drive on Windows.
For instance, \foo\bar.
- Relative paths to the current folder.
For instance, C:foo\bar.
- Extended length paths in Windows.
For instance, \\?\foo.
- Paths that contain '.' or '..' except at the start of the pattern.
For instance, /foo/bar/.. /a?.c.
- The '*' character by itself.

Modularize Polyspace Analysis by Using Build Command

To configure the Polyspace analysis, you can reuse the compilation options in your build command such as `make`. First, you trace your build command with `polyspace-configure` (or `polyspaceConfigure` in MATLAB®) and create a Polyspace options file. You later specify this options file for the subsequent Polyspace analysis.

If your build command creates several binaries, by default `polyspace-configure` groups the source files for all binaries into one Polyspace options file. If binaries that use the same source files or functions are compiled with different options, you lose this distinction in the subsequent Polyspace analysis. The presence of the same function multiple times can lead to link errors during the Polyspace analysis and sometimes to incorrect results.

This topic shows how to create a separate Polyspace options file for each binary created in your makefile. Suppose that a makefile creates four binaries: two executable (target `cmd1` and `cmd2`) and two shared libraries (target `liba` and `libb`). You can create a separate Polyspace options file for each of these binaries.

To try this example, use the files in `polyspaceroot\help\toolbox\polyspace_bug_finder_server\examples\multiple_modules`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a` or `C:\Program Files\Polyspace Server\R2019a`.

Build Source Code

Inspect the makefile. The makefile creates four binaries:

```
CC := gcc
LD := ld

LIBA_SOURCES := $(wildcard src/liba/*.c)
LIBB_SOURCES := $(wildcard src/libb/*.c)
CMD1_SOURCES := $(wildcard src/cmd1/*.c)
CMD2_SOURCES := $(wildcard src/cmd2/*.c)
LIBA_OBJ := $(notdir $(LIBA_SOURCES:.c=.o))
LIBB_OBJ := $(notdir $(LIBB_SOURCES:.c=.o))
CMD1_OBJ := $(notdir $(CMD1_SOURCES:.c=.o))
CMD2_OBJ := $(notdir $(CMD2_SOURCES:.c=.o))
LIBB_SOBJ := libb.so
LIBA_SOBJ := liba.so

all: cmd1 cmd2

cmd1: liba libb
    $(CC) -o $@ $(CMD1_SOURCES) $(LIBA_SOBJ) $(LIBB_SOBJ)

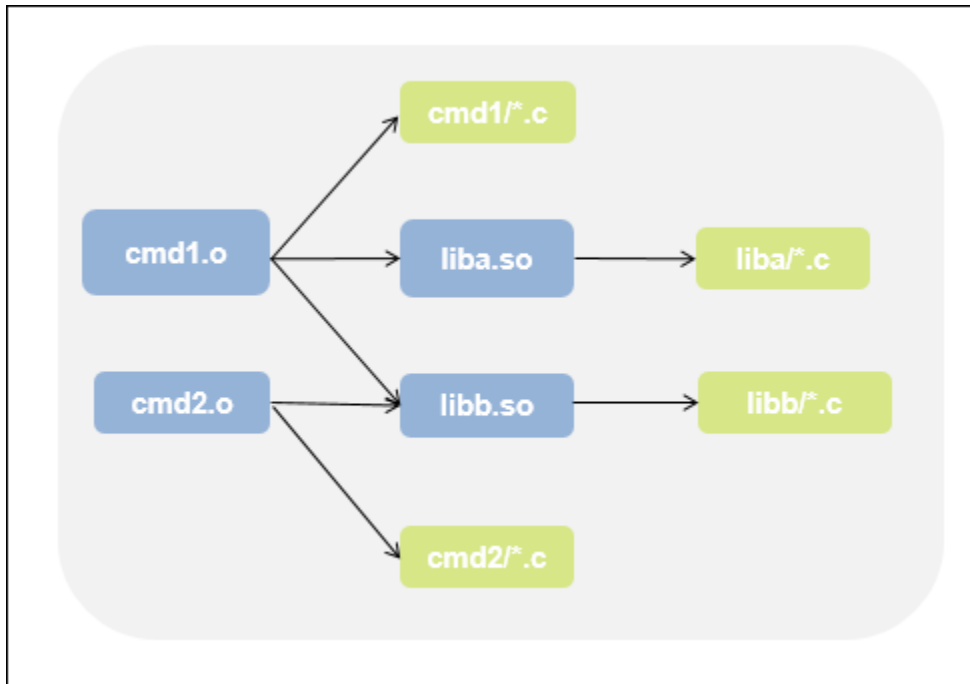
cmd2: libb
    $(CC) -c $(CMD2_SOURCES)
    $(LD) -o $@ $(CMD2_OBJ) $(LIBB_SOBJ)

liba: libb
    $(CC) -fPIC -c $(LIBA_SOURCES)
    $(CC) -shared -o $(LIBA_SOBJ) $(LIBA_OBJ)

libb:
    $(CC) -fPIC -c $(LIBB_SOURCES)
    $(CC) -shared -o $(LIBB_SOBJ) $(LIBB_OBJ)

.PHONY: clean
clean:
    rm *.o
```

The binaries created have the dependencies shown in this figure. For instance, creation of the object `cmd1.o` depends on all `.c` files in the folder `cmd1` and the shared objects `liba.so` and `libb.so`.



Build your source code by using the makefile. Use the `-B` flag to ensure full build.

```
make -B
```

Make sure that the build runs to completion.

Create One Polyspace Options File for Full Build

Trace the build command by using `polyspace-configure`. Use the option `-output-options-file` to create a Polyspace options file `psoptions` from the build command.

```
polyspace-configure -output-options-file psoptions make -B
```

Run Bug Finder or Code Prover by using the previously created options file: Save the analysis results in a `results` subfolder.

```
polyspace-bug-finder-server -options-file psoptions -results-dir results
```

You see this link error (warning in Bug Finder):

```
Procedure 'main' multiply defined.
```

The error occurs because the files `cmd1/cmd1_main.c` and `cmd2/cmd2_main.c` both have a `main` function. When you run your build command, the two files are used in separate targets in the makefile. However, `polyspace-configure` by default creates one options file for the full build. The Polyspace options file contains both source files resulting in conflicting definitions of the `main` function.

To verify the cause of the error, open the Polyspace options file `psoptions`. You see these lines that include the files with conflicting definitions of the `main` function.

```
-sources src/cmd1/cmd1_main.c  
-sources src/cmd2/cmd2_main.c
```

Create Options File for Specific Binary in Build Command

To avoid the link error, build the source code for a specific binary when tracing your build command by using `polyspace-configure`.

For instance, build your source code for the binary `cmd1.o`. Specify the makefile target `cmd1` for `make`, which creates this binary.

```
polyspace-configure -output-options-file psoptions make -B cmd1
```

Run Bug Finder or Code Prover by using the previously created options file.

```
polyspace-bug-finder-server -options-file psoptions -results-dir results
```

The link error does not occur and the analysis runs to completion. You can open the Polyspace options file `psoptions` and see that only the source files in the `cmd1` subfolder and the files involved in creating the shared objects are included with the `-sources`

option. The source files in the `cmd2` subfolder, which are not involved in creating the binary `cmd1.o`, are not included in the Polyspace options file.

Special Considerations for Libraries (Code Prover only)

If you trace the creation of a shared object from libraries, the source files extracted do not contain a `main` function. In the subsequent Code Prover analysis, you can see an error because of the missing `main`.

Use the Polyspace option `Verify module or library (-main-generator)` to generate a `main` function. Specify the option in the options file that was created or directly at the command line. See “Verify C Application Without main Function” (Polyspace Code Prover Server).

In C++, use these additional options for classes:

- `Class (-class-analyzer)`
- `Functions to call within the specified classes (-class-analyzer-calls)`

Create One Options File Per Binary Created in Build Command

To create an options file for a specific binary created in the build command, you must know the details of your build command. If you are not familiar with the internal details of the build command, you can create a separate Polyspace options file for *every* binary created in the build command. The approach works for binaries that are executables, shared (dynamic) libraries and static libraries.

This approach works only if you use these compilers:

- GNU C or GNU C++
- Microsoft Visual C++

Trace the build command by using `polyspace-configure`. To create a separate options file for each binary, use the option `-module` with `polyspace-configure`.

```
polyspace-configure -module -output-options-path optionsFilesFolder make -B
```


The command creates options files in the folder `optionsFilesFolder`. In the preceding example, the command creates four options files for the four binaries:

- `cmd1.psopts`
- `cmd2.psopts`
- `liba_so.psopts`
- `libb_so.psopts`

You can run Polyspace on the code implementation of a specific binary by using the corresponding options file. For instance, you can run Code Prover on the code implementation of the binary created from the makefile target `cmd1` by using this command:

```
polyspace-bug-finder-server -options-file cmd1.psopts -results-dir results
```

For this approach, you do not need to know the details of your build command. However, when you create a separate options file for each binary in this way, each options file contains source files directly involved in the binary and not through shared objects. For instance, the options file `cmd1.psopts` in this example specifies only the source files in the `cmd1` subfolder and not the source files involved in creating the shared objects `liba.so` and `libb.so`. The subsequent analysis by using this options file cannot access functions from the shared objects and uses function stubs instead. In the Code Prover analysis, if you see too many orange checks due to the stubbing, use the approach stated in the section “Create Options File for Specific Binary in Build Command” on page 2-11.

Special Considerations for Libraries (Code Prover only)

If you trace the creation of a shared object from libraries, the source files extracted do not contain a `main` function. In the subsequent Code Prover analysis, you can see an error because of the missing `main`.

Use the Polyspace option `Verify module or library (-main-generator)` to generate a `main` function. Specify the option in the options file that was created or directly at the command line. See “Verify C Application Without main Function” (Polyspace Code Prover Server).

In C++, use these additional options for classes:

- Class (-class-analyzer)
- Functions to call within the specified classes (-class-analyzer-calls)

See Also

polyspace-bug-finder-server | polyspace-configure

More About

- “Create Polyspace Analysis Configuration from Build Command” on page 2-2

Offload Polyspace Analysis to Remote Servers from Desktop

- “Send Polyspace Analysis from Desktop to Remote Servers” on page 3-2
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 3-7

Send Polyspace Analysis from Desktop to Remote Servers

In this section...
“Client-Server Workflow for Running Analysis” on page 3-2
“Prerequisites” on page 3-4
“Offload Analysis in Polyspace User Interface” on page 3-4

You can perform a Polyspace analysis locally on your desktop or offload the analysis to one or more dedicated servers. You offload a Polyspace analysis from a Polyspace desktop product such as Polyspace Bug Finder but the analysis runs on the server using a Polyspace server product such as Polyspace Bug Finder Server.

This topic shows how to send a Polyspace analysis from the user interface of the Polyspace desktop products.

- To offload an analysis with scripts, see “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 3-7.
- For a simple tutorial that walks through all the steps for offloading a Polyspace analysis, see “Send Bug Finder Analysis from Desktop to Locally Hosted Server”. In the tutorial, the same computer acts as the client and the server.

Client-Server Workflow for Running Analysis

After the initial setup, you can submit a Polyspace analysis from a client desktop to a server. The client-server workflow happens in three steps. All three steps can be performed on the same computer or three different computers.

- 1 Client node:** You specify Polyspace analysis options and start the analysis on the client desktop. The initial phase of analysis upto compilation runs on the desktop. After compilation, the analysis job is submitted to the server.

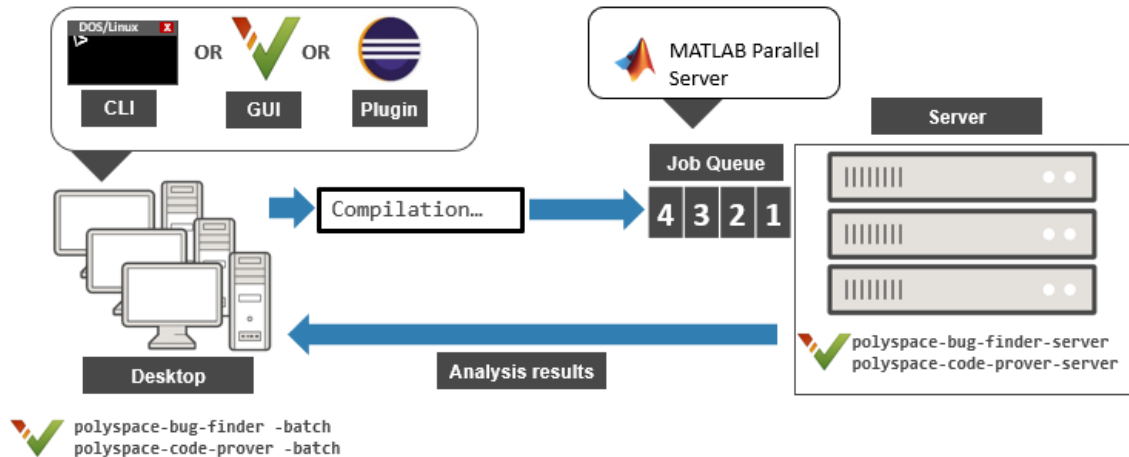
You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.

- 2 Head node:** The server consists of a head node and several worker nodes. The head node uses a job scheduler to manage submissions from multiple client desktops. The jobs are then distributed to the worker nodes as they become available.

You require the product MATLAB Parallel Server™ on the computer that acts as the head node.

- 3 Worker nodes:** When a worker becomes available, the job scheduler assigns the analysis to the worker. The Polyspace analysis runs on the worker and the results are downloaded back to the client desktop for review.

You require the product MATLAB Parallel Server on the computers that act as worker nodes. You also require the Polyspace server products, Polyspace Bug Finder Server and/or Polyspace Code Prover Server, to run the analysis.



Prerequisites

Before offloading an analysis from the user interface of the Polyspace desktop products, you must set up your project's source files, analysis options, and remote analysis settings. If you have not done so, for more information on:

- How to add source files, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Bug Finder).
- How to set up communication between client and server, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

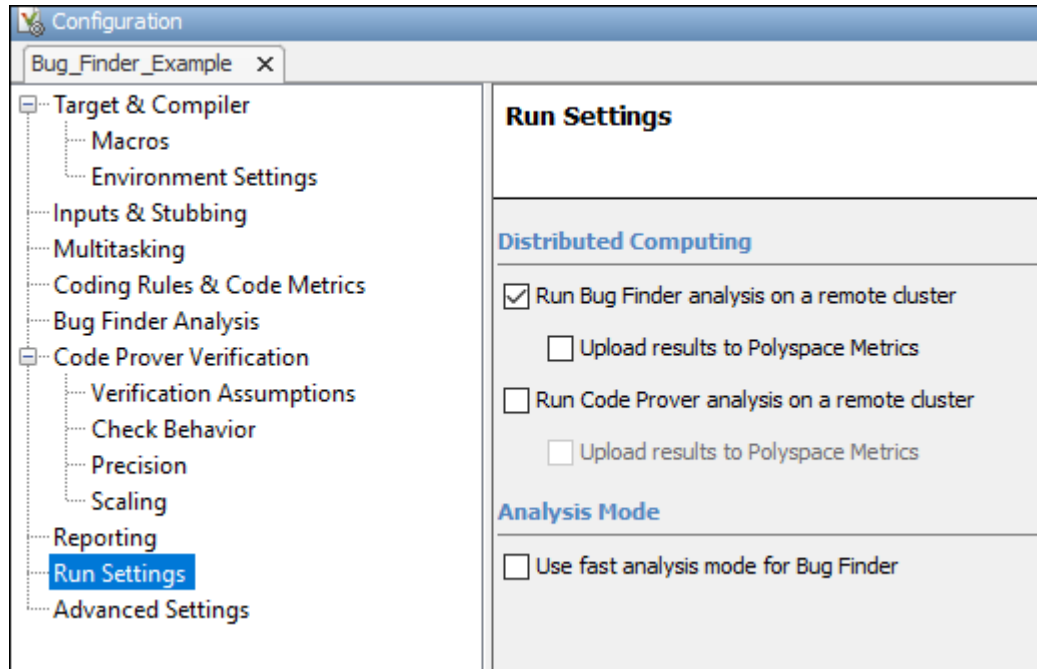
Once you have set up a Polyspace project and established communication between a desktop and a remote server, you are ready to offload a Polyspace analysis.

Offload Analysis in Polyspace User Interface

To start a remote analysis:

- 1 Select a project to analyze.
- 2 On the **Configuration** pane, select **Run Settings**.

Select **Run Bug Finder analysis on a remote cluster** and/or **Run Code Prover analysis on a remote cluster**.



- 3 If you want to store your results in the Polyspace Metrics repository, select **Upload results to Polyspace Metrics**.

Otherwise, clear this check box. After analysis, the results are downloaded to the desktop for review.

- 4 Start the analysis. For instance, to start a Bug Finder analysis, click the **Run Bug Finder** button.

The compilation part of the analysis takes place on the desktop product. After compilation, the analysis is offloaded to the server.

- 5 To monitor the analysis, select **Tools > Open Job Monitor**. In the Polyspace Job Monitor, follow your queued job to monitor progress.

Once the analysis is complete, the results are downloaded back to the user interface of the Polyspacedesktop products. You can open the results directly in the user interface. If you uploaded the results to Polyspace Metrics, you have to explicitly download them from the Polyspace Metrics interface.

If the analysis stops after compilation and you have to restart the analysis, to avoid restarting from the compilation phase, use the option `-submit-job-from-previous-compilation-results`.

Note If you choose to upload results to Polyspace Metrics, your results are not downloaded automatically after verification. Use the Polyspace Metrics web dashboard to view the results and download them to your desktop. For more information, see “View Projects in Polyspace Metrics” (Polyspace Bug Finder).

See Also

Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`)

More About

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts” on page 3-7

Send Polyspace Analysis from Desktop to Remote Servers Using Scripts

Instead of running a Polyspace analysis on your local desktop, you can send the analysis to a remote cluster. You can use a dedicated cluster for running Polyspace to free up memory on your local desktop.

This topic shows how to use Windows or Linux scripts to send the analysis to a remote cluster and download the results to your desktop after analysis.

- To offload an analysis from the Polyspace user interface, see “Send Polyspace Analysis from Desktop to Remote Servers” on page 3-2.
- For a simple tutorial that walks through all the steps for offloading a Polyspace analysis, see “Send Bug Finder Analysis from Desktop to Locally Hosted Server”. In the tutorial, the same computer acts as the client and the server.

Client-Server Workflow for Running Analysis

After the initial setup, you can submit a Polyspace analysis from a client desktop to a server. The client-server workflow happens in three steps. All three steps can be performed on the same computer or three different computers.

- 1 Client node:** You specify Polyspace analysis options and start the analysis on the client desktop. The initial phase of analysis upto compilation runs on the desktop. After compilation, the analysis job is submitted to the server.

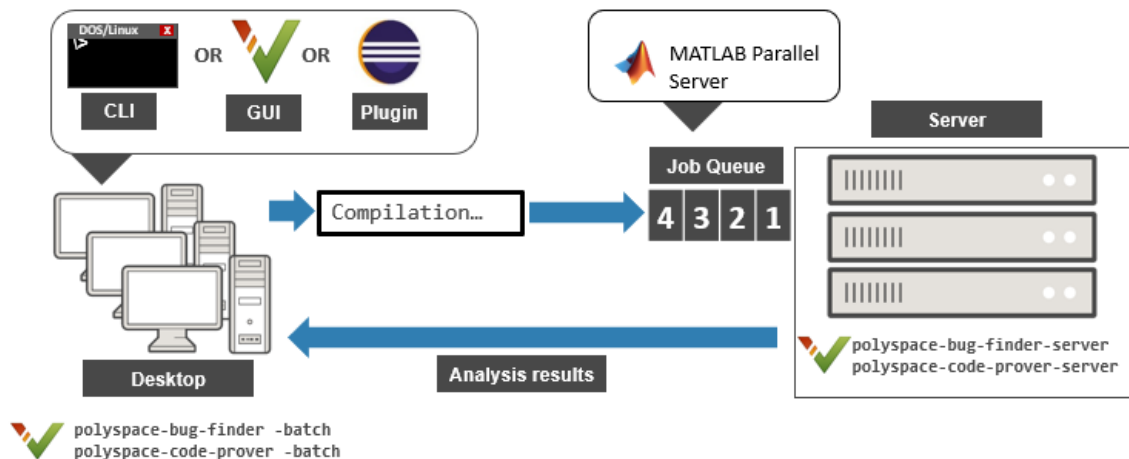
You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.

- 2 Head node:** The server consists of a head node and several worker nodes. The head node uses a job scheduler to manage submissions from multiple client desktops. The jobs are then distributed to the worker nodes as they become available.

You require the product MATLAB Parallel Server on the computer that acts as the head node.

- 3 Worker nodes:** When a worker becomes available, the job scheduler assigns the analysis to the worker. The Polyspace analysis runs on the worker and the results are downloaded back to the client desktop for review.

You require the product MATLAB Parallel Server on the computers that act as worker nodes. You also require the Polyspace server products, Polyspace Bug Finder Server and/or Polyspace Code Prover Server to run the analysis.



Prerequisites

Before you run a remote analysis by using scripts, you must set up communication between a desktop and a remote server. See “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Run Remote Analysis

To run a remote analysis, use this command:

```
polyspaceroot\polyspace\bin\polyspace-bug-finder  
-batch -scheduler NodeHost | MJSName@NodeHost [options]
```

where:

- *polyspaceroot* is the installation folder of Polyspace desktop products, for instance, C:\Program Files\Polyspace\R2019a.
- *NodeHost* is the name of the computer that hosts the head node of the MATLAB Parallel Server cluster.

MJSName is the name of the MATLAB Job Scheduler on the head node host.

If you set up communications with a cluster from the Polyspace user interface, you can determine *NodeHost* and *MJSName* from the user interface. Select **Metrics > Metrics and Remote Server Settings**. Open the MATLAB Parallel Server Admin Center. Under **MATLAB Job Scheduler**, see the **Name** and **Hostname** columns for *MJSName* and *NodeHost*.

If you use the `startjobmanager` command to start the MATLAB Job Scheduler, *MJSName* is the argument of the option `-name`. For details, see “Configure Advanced Options for MATLAB Job Scheduler Integration” (MATLAB Parallel Server).

- *options* are the analysis options. These options are the same as that of a local analysis. For instance, you can use these options:
 - `-sources-list-file`: Specify a text file with one source file name per line.
 - `-options-file`: Specify a text file with one option per line.
 - `-results-dir`: Specify a download folder for storing results after analysis.

For the full list of options, see “Analysis Options”. Alternatively, you can:

- Start an analysis in the user interface and stop after compilation. You can obtain the text files and scripts for running the analysis at the command line. See “Configure Polyspace Analysis Options in User Interface and Generate Scripts” on page 1-7.
- Enter `polyspace-bug-finder -h`. The list of available options with a brief description are displayed.
- Place your cursor over each option on the **Configuration** pane in the Polyspace user interface. Click the **More Help** button for information on the option syntax and when the option is required.

The analysis executes locally on your desktop up to the end of the compilation phase. After compilation, the software submits the analysis job to the cluster and provides a job ID. You can also read the ID from the file `ID.txt` in the results folder. To monitor your analysis, use the `polyspace-jobs-manager` command with the job ID.

If the analysis stops after compilation and you have to restart the analysis, to avoid rerunning the compilation phase, use the option `-submit-job-from-previous-compilation-results`.

Manage Remote Analysis

To manage multiple remote analyses, use the option `-batch`. For instance:

```
polyspaceroot\polyspace\bin\polyspace-jobs-manager action  
-scheduler schedulerName
```

See also Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`). Here:

- *polyspaceroot* is your MATLAB installation folder.
- *schedulerName* is one of the following:
 - Name of the computer that hosts the head node of your MATLAB Parallel Server cluster (*NodeHost*).
 - Name of the MATLAB Job Scheduler on the head node host (*MJSName@NodeHost*).
 - Name of a MATLAB cluster profile (*ClusterProfile*).

For more information about clusters, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox)

If you do not specify a job scheduler, `polyspace-job-manager` uses the scheduler specified in the Polyspace preferences. To see the scheduler name, select **Tools > Preferences**. On the **Server Configuration** tab, see the **Job scheduler host name**.

- *action* refers to the possible action commands to manage jobs on the scheduler:
 - `listjobs`:

Generate a list of Polyspace jobs on the scheduler. For each job, the software produces this information:

- `ID` — Verification or analysis identifier.
 - `AUTHOR` — Name of user that submitted job.
 - `APPLICATION` — Name of Polyspace product, for example, Polyspace Code Prover or Polyspace Bug Finder.
 - `LOCAL_RESULTS_DIR` — Results folder on local computer, specified through the **Tools > Preferences > Server Configuration** tab.
 - `WORKER` — Local computer from which job was submitted.
 - `STATUS` — Status of job, for example, running and completed.
 - `DATE` — Date on which job was submitted.
 - `LANG` — Language of submitted source code.
- `download -job ID -results-folder FolderPath:`

Download results of analysis with specified ID to folder specified by *FolderPath*.

When the analysis job is queued on the server, the command `polyspace-bug-finder` returns a job id. In addition, a file `ID.txt` in the results folder contains the job ID in this format:

```
job_id;server_name:project_name version_number
```

For instance, `92;localhost:Demo 1.0`.

If you do not use the `-results-folder` option, the software downloads the result to the folder that you specified when starting analysis, using the `-results-dir` option.

After downloading results, use the Polyspace user interface to view the results.

- `getlog -job ID:`
Open log for job with specified ID.
- `remove -job ID:`
Remove job with specified ID.
- `promote -job ID:`
Promote job with specified ID in the queue.
- `demote -job ID`
Demote job with specified ID in the queue.

Sample Scripts for Remote Analysis

In Windows, to avoid typing the commands each time, you can save the commands in a batch file. In Linux, you can relaunch the analysis by using a shell script. To create a batch file for running analysis:

- 1 Save your analysis options in a file `listoptions.txt`. See `-options-file`.
- 2 Create a file `launcher.bat` in a text editor like Notepad.

In the file, enter these commands:

```
echo off
set POLYSPACE_PATH=polyspaceroot\polyspace\bin
set RESULTS_PATH=C:\Results
set OPTIONS_FILE=C:\Options\listoptions.txt
"%POLYSPACE_PATH%\polyspace-bug-finder.exe" -batch -scheduler localhost
                                         -results-dir "%RESULTS_PATH%" -options-file "%OPTIONS_FILE%"
pause
```

`polyspaceroot` is the Polyspace installation folder. `localhost` is the name of the computer that hosts the head node of your MATLAB Parallel Server cluster.

- 3 Replace the definitions of these variables in the file:
 - `POLYSPACE_PATH`: Enter the actual location of the `.exe` file.
 - `RESULTS_PATH`: Enter the path to a folder. The files generated during compilation are saved in the folder.
 - `OPTIONS_FILE`: Enter the path to the file `listoptions.txt`.
- 4 Double-click `launcher.bat` to run the analysis.

Tip If you run a Polyspace analysis, a Windows `.bat` or Linux `.sh` file is generated. The file is in the `.settings` subfolder in your results folder. Instead of writing a script from scratch, you can relaunch the analysis using this file.

See Also

Run Bug Finder or Code Prover analysis on a remote cluster (`-batch`)

More About

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Send Polyspace Analysis from Desktop to Remote Servers” on page 3-2

Run Polyspace Analysis on Server with MATLAB Scripts

Integrate Polyspace Server Products with MATLAB and Simulink

You can install Polyspace Bug Finder Server and Polyspace Code Prover Server as standalone products and analyze C/C++ code. However, if you have an installation of MATLAB, you can run the Polyspace analysis with MATLAB scripts.

If you install Polyspace server products and MATLAB, you have to run the MATLAB installer twice and install Polyspace in a different root folder from the other products. For instance, in Windows:

- Your default MATLAB root folder is C:\Program Files\MATLAB\R2019a.
- Your default Polyspace root folder is C:\Program Files\Polyspace Server\R2019a for the Polyspace server products.

To run Polyspace from within MATLAB, Simulink or MATLAB Coder™, you have to perform a post-installation step to link your MATLAB and Polyspace installations.

Integrate Polyspace with MATLAB Installation from Same Release

If your Polyspace and MATLAB installations belong to the same release, you can use all MATLAB functions and classes available for running Polyspace.

To link your MATLAB and Polyspace installations:

- 1 Open MATLAB with administrator privileges.
- 2 Navigate to *polyspaceserverroot*\toolbox\polyspace\pscore\pscore\. Here, *polyspaceserverroot* is the Polyspace installation folder, for instance, C:\Program Files\Polyspace Server\R2019a.
- 3 At the MATLAB command prompt, enter:

```
polyspacesetup('install')
```

You see a prompt stating that the workspace will be cleared. Click **Yes** to continue the linking. The process might take a few minutes to complete.

- 4 Restart MATLAB. You can now use all functions and classes available for running Polyspace server products.

A MATLAB installation can be linked with only one Polyspace installation. To link to a new Polyspace installation, any previous links must be removed. To remove a link between a Polyspace and MATLAB installation, repeat the same steps as before with one difference: At the MATLAB command prompt, enter:

```
polyspacesetup('uninstall')
```

Integrate Polyspace with MATLAB Installation from Different Release

If you upgrade your Polyspace server product installation but not your MATLAB installation, you can link your MATLAB installation with the later release of the Polyspace server product.

Remove the existing link between your Polyspace and MATLAB installation as described in the previous section. Then, in your post-installation step, navigate to *polyspaceserverroot*\toolbox\polyspace\pscore\pscore\, where *polyspaceroot* is the installation folder for the later release of Polyspace Bug Finder Server and/or Polyspace Code Prover Server. At the MATLAB command prompt, enter:

```
polyspacesetup('install')
```

If you integrate MATLAB with a later release of Polyspace, you cannot use all functions and classes available to run the analysis. In particular, you cannot use the `polyspace.Project` class. Instead, use the `polyspaceCodeProverServer` function to run Code Prover and the `polyspaceBugFinderServer` function to run Bug Finder on handwritten code.

Check Integration Between MATLAB and Polyspace

To check if a MATLAB installation is already linked to a Polyspace installation, open MATLAB and enter a Polyspace-specific command, for instance:

```
obj = polyspace.Project
```

The command will display an error if the MATLAB installation is not linked to a Polyspace installation.

Run Polyspace Server Products with MATLAB Scripts

In a continuous integration process, you can execute MATLAB scripts that run a Polyspace analysis on new code submissions and compares the results against predefined criteria. Use these functions/classes:

- Create a `polyspace.Project` object to configure Polyspace analysis options, run an analysis and read results to MATLAB tables. You can use other MATLAB functions for comparing results against predefined criteria.

To only read existing results without running an analysis, use the `polyspace.BugFinderResults` or `polyspace.CodeProverResults` class with the path to a results folder.

- If you want a more granular selection of checkers for:
 - Coding rules, create a `polyspace.CodingRulesOptions` object.
 - Bug Finder defects, create a `polyspace.DefectsOptions` object.

To create a custom target for the analysis and explicitly specify sizes of data types, create a `polyspace.GenericTargetOptions` object.

You can also use the `polyspaceBugFinderServer` or `polyspaceCodeProverServer` function to run the analysis and then read results with the `polyspace.BugFinderResults` or `polyspace.CodeProverResults` class. If you use build commands to build your source code, you can create a Polyspace configuration from the build command using the `polyspaceConfigure` function.

Configure Target and Compiler Options

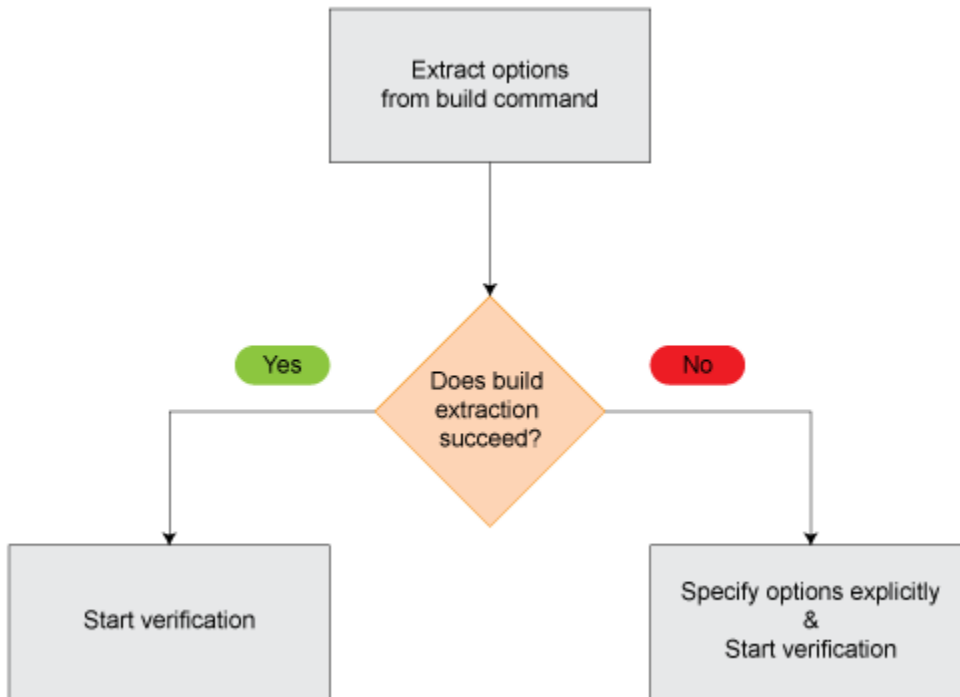
Specify Target Environment and Compiler Behavior

Before verification, specify your source code language (C or C++), target processor, and the compiler that you use for building your code. In certain cases, to emulate your compiler behavior, you might have to specify additional options.

Using your specification, the verification determines the sizes of fundamental types, considers certain macros as defined, and interprets compiler-specific extensions of the Standard. If the options do not correspond to your run-time environment, you can encounter:

- Compilation errors
- Verification results that might not apply to your target

If you use a build command such as `gmake` to build your code and the build command meets certain restrictions, you can extract the options from the build command. Otherwise, specify the options explicitly.



Extract Options from Build Command

If you use build automation scripts to build your source code, you can set up a Polyspace project from your scripts. The options associated with your compiler are specified in that project.

In the Polyspace desktop products, for information on how to trace your build command from the:

- Polyspace user interface, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Bug Finder).
- DOS or UNIX command line, see `polyspace-configure`.
- MATLAB command line, see `polyspaceConfigure`.

In the Polyspace server products, for information on how to trace your build command, see “Create Polyspace Analysis Configuration from Build Command” on page 2-2.

For Polyspace project creation, your build automation script (makefile) must meet certain requirements. See “Requirements for Project Creation from Build Systems” on page 5-21.

Specify Options Explicitly

If you cannot trace your build command and therefore manually create a project, you have to specify the options explicitly.

- In the user interface of the Polyspace desktop products, select a project configuration. On the **Configuration** pane, select **Target & Compiler**. Specify the options.
- At the DOS or UNIX command line, specify flags with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command.
- At the MATLAB command line, specify arguments with the `polyspaceBugFinder`, `polyspaceCodeProver`, `polyspaceBugFinderServer` or `polyspaceCodeProverServer` function.

Specify the options in this order.

- Required options:
 - **Source code language (-lang)**: If all files have the same extension `.c` or `.cpp`, the verification uses the extension to determine the source code language. Otherwise, explicitly specify the option.
 - **Compiler (-compiler)**: Select the compiler that you use for building your source code. If you cannot find your compiler, use an option that closely matches your compiler.
 - **Target processor type (-target)**: Specify the target processor on which you intend to execute your code. For some processors, you can change the default specifications. For instance, for the processor `hc08`, you can change the size of types `double` and `long double` from 32 to 64 bits.

If you cannot find your target processor, you can create your own target and specify the sizes of fundamental types, default signedness of `char`, and endianness of the target machine. See **Generic target options**.

- Language-specific options:
 - **C standard version (-c-version)**: The default C language standard depends on your compiler specification. If you do not specify a compiler explicitly, the

default analysis uses the C99 standard. Specify an earlier standard such as C90 or a later standard such as C11.

- **C++ standard version (-cpp-version):** The default C++ language standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C++03 standard. Specify later standards such as C++11 or C++14.
- **Compiler-specific options:**

Whether these options are available or not depends on your specification for **Compiler (-compiler)**. For instance, if you select a **visual** compiler, the option **Pack alignment value (-pack-alignment-value)** is available. Using the option, you emulate the compiler option **/Zp** that you use in Visual Studio.

For all compiler-specific options, see “Target and Compiler”.

- **Advanced options:**

Using these options, you can modify the verification results. For instance, if you use the option **Division round down (-div-round-down)**, the verification considers that quotients from division or modulus of negative numbers are rounded down. Use these options only if you use similar options when compiling your code.

For all advanced options, see “Target and Compiler”.

- **Compiler header files:**

If you specify the **diab**, **tasking** or **greenhills** compiler, you must specify the path to your compiler header files. See “Provide Standard Library Headers for Polyspace Analysis” on page 5-19.

If you still see compilation errors after running analysis, you might have to specify other options:

- **Define macros:** Sometimes, a compilation error occurs because the analysis considers a macro as undefined. Explicitly define these macros. See **Preprocessor definitions (-D)**.
- **Specify include files:** Sometimes, a compilation error occurs because your compiler defines standard library functions differently from Polyspace and you do not provide your compiler include files. Explicitly specify the path to your compiler include files. See “Provide Standard Library Headers for Polyspace Analysis” on page 5-19.

See Also

C standard version (-c-version) | C++ standard version (-cpp-version) | Compiler (-compiler) | Preprocessor definitions (-D) | Source code language (-lang) | Target processor type (-target)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 5-7
- “Provide Standard Library Headers for Polyspace Analysis” on page 5-19

C/C++ Language Standard Used in Polyspace Analysis

The Polyspace analysis adheres to a specific language standard for code compilation. The language standard, along with your compiler specification, defines the language elements that you can use in your code. For instance, if the Polyspace analysis uses the C99 standard, C11 features such as use of the thread support library from `threads.h` causes compilation errors.

Supported Language Standards

The Polyspace analysis supports these standards:

- **C:** C90, C99, C11

The default standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C99 standard. To change the language standard, use the option `C standard version (-c-version)`.

- **C++:** C++03, C++11, C++14

The default standard depends on your compiler specification. If you do not specify a compiler explicitly, the default analysis uses the C++03 standard. To change the language standard, use the option `C++ standard version (-cpp-version)`.

Default Language Standard

The default language standard depends on your specification for the option `Compiler (-compiler)`.

Compiler	C Standard	C++ Standard
generic	C99	C++03
gnu3.4, gnu4.6, gnu4.7, gnu4.8, gnu4.9	C99	C++03
gnu5.x	C11	C++03
gnu6.x	C11	C++14
gnu7.x	C11	C++14

Compiler	C Standard	C++ Standard
clang3.x	C99	C++03 The analysis accepts some C++11 extensions.
clang4.x	C99	C++03 The analysis accepts C++14 extensions.
clang5.x	C99	C++03 The analysis accepts C++14 extensions.
visual9.0, visual10.0, visual11.0, visual12.0	C99	C++03
visual14.0	C99	C++14
visual15.x	C99	C++14
iar	C99	C++03
armcc	C99	C++03
armclang	C11	C++03
keil	C99	C++03
diab	C99	C++03
tasking	C99	C++03
greenhills	C99	C++03
codewarrior	C99	C++03
ti	C99	C++03
iar-ew	C99	C++03
renesas	C99	C++03

See Also

C standard version (-c-version) | C++ standard version (-cpp-version) | Compiler (-compiler)

More About

- “C11 Language Elements Supported in Polyspace” on page 5-10
- “C++11 Language Elements Supported in Polyspace” on page 5-12
- “C++14 Language Elements Supported in Polyspace” on page 5-15

C11 Language Elements Supported in Polyspace

This table provides a partial list of C language elements that have been introduced since C11 and the corresponding Polyspace support. If your code contains non-supported constructions, Polyspace reports a compilation error.

C11 Language Element	Supported
<code>alignas</code> and <code>alignof</code> convenience macros	Yes
<code>aligned_alloc</code> function	Yes
<code>noreturn</code> convenience macros	Yes
Generic selection	Yes
Thread support library (<code>threads.h</code>)	Yes
Atomic operations library (<code>stdatomic.h</code>)	Yes
Atomic types with <code>_Atomic</code>	Yes. If you use the Clang compiler, see limitations book for limitations on atomic data types. See “Limitations of Polyspace Verification” (Polyspace Code Prover).
UTF-16 and UTF-32 character utilities	Yes
Bound-checking interfaces or alternative versions of standard library functions that check for buffer overflows (Annex K of C11) For instance, <code>strcpy_s</code> is an alternative to <code>strcpy</code> that checks for certain errors in the string copy.	No. Polyspace checks for certain run-time errors in use of standard library functions. The checking does not extend to these alternatives.
Anonymous structures and unions	Yes
Static assert declaration	Yes

C11 Language Element	Supported
Features related to error handling such as <code>errno_t</code> and <code>rsize_t</code> typedef-s	No. If you see compilation errors from use of these typedef-s, explicitly specify the path to your compiler headers. See “Provide Standard Library Headers for Polyspace Analysis” on page 5-19.
<code>quick_exit</code> and <code>at_quick_exit</code>	Yes. In Bug Finder, functions registered with <code>at_quick_exit</code> appear as uncalled.
<code>CMPLX</code> , <code>CMPLXF</code> and <code>CMPLXL</code> macros	Yes

See Also

C standard version (-c-version)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 5-7

C++11 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++11 and the corresponding Polyspace support. If your code contains non-supported constructions, Polyspace reports a compilation error.

C++11 Std Ref	Description	Supported
C++2011-DR226	Default template arguments for function templates	Yes
C++2011-DR339	Solving the SFINAE problem for expressions	Yes
C++2011-N1610	Initialization of class objects by rvalues	Yes
C++2011-N1653	C99 preprocessor	Yes
C++2011-N1720	Static assertions	Yes
C++2011-N1737	Multi-declarator auto	Yes
C++2011-N1757	Right angle brackets	Yes
C++2011-N1791	Extended friend declarations	No
C++2011-N1811	long long	Yes
C++2011-N1984	auto-typed variables	Yes
C++2011-N1986	Delegating constructors	Yes
C++2011-N1987	Extern templates	Yes
C++2011-N1988	Extended integral types	Yes
C++2011-N2118	Rvalue references	Yes
C++2011-N2170	Universal character name literals	Yes
C++2011-N2179	Concurrency: Propagating exceptions	No
C++2011-N2235	Generalized constant expressions	Yes
C++2011-N2239	Concurrency: Sequence points	No new syntax/keyword not affect Polyspace su
C++2011-N2242	Variadic templates	Yes
C++2011-N2249	New character types	Yes
C++2011-N2253	Extending sizeof	Yes
C++2011-N2258	Template aliases	Yes
C++2011-N2340	__func__ predefined identifier	Yes

C++11 Std Ref	Description	Supported
C++2011-N2341	Alignment support	Yes
C++2011-N2342	Standard Layout Types	Yes
C++2011-N2343	Declared type of an expression	Yes
C++2011-N2346	Defaulted and deleted functions	Yes
C++2011-N2347	Strongly typed enums	Yes
C++2011-N2427	Concurrency: Atomic operations	No
C++2011-N2429	Concurrency: Memory model	No new syntax/keyw not affect Polyspace su
C++2011-N2431	Null pointer constant	Yes
C++2011-N2437	Explicit conversion operators	Yes
C++2011-N2439	Rvalue references for *this	Yes
C++2011-N2440	Concurrency: Abandoning a process and at_quick_exit	Yes
C++2011-N2442	Unicode string literals	Yes
C++2011-N2442	Raw string literals	Yes
C++2011-N2535	Inline namespaces	Yes
C++2011-N2540	Inheriting constructors	Yes
C++2011-N2541	New function declarator syntax	Yes
C++2011-N2544	Unrestricted unions	Yes
C++2011-N2546	Removal of auto as a storage-class specifier	Yes
C++2011-N2547	Concurrency: Allow atomics use in signal handlers	No
C++2011-N2555	Extending variadic template template parameters	Yes
C++2011-N2657	Local and unnamed types as template arguments	Yes
C++2011-N2659	Concurrency: Thread-local storage	No
C++2011-N2660	Concurrency: Dynamic initialization and destruction with concurrency	Yes
C++2011-N2664	Concurrency: Data-dependency ordering: atomics and memory model	No
C++2011-N2672	Initializer lists	Yes
C++2011-N2748	Concurrency: Strong Compare and Exchange	No

C++11 Std Ref	Description	Supported
C++2011-N2752	Concurrency: Bidirectional Fences	No
C++2011-N2756	Nonstatic data member initializers	Yes
C++2011-N2761	Generalized attributes	Yes
C++2011-N2764	Forward declarations for enums	Yes
C++2011-N2765	User-defined literals	Yes
C++2011-N2927	New wording for C++0x lambdas	Yes
C++2011-N2928	Explicit virtual overrides	Yes
C++2011-N2930	Range-based for	Yes
C++2011-N3050	Allowing move constructors to throw [noexcept]	Yes
C++2011-N3053	Defining move special member functions	Yes
C++2011-N3276	decltype and call expressions	Yes

See Also

C++ standard version (-cpp-version)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 5-7

C++14 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++14 and the corresponding Polyspace support. If your code contains non-supported constructions, Polyspace reports a compilation error.

C++14 Std Ref	Description	Supported
C++2014-N3323	Implicit conversion from class type in certain contexts such as <code>delete</code> or <code>switch</code> statement.	This C++14 feature allows implicit conversion from class type in certain contexts. No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3462	More SFINAE-friendly <code>std::result_of</code>	Yes
C++2014-N3472	Binary literals, for instance, <code>0b100</code> .	Yes
C++2014-N3545	<code>operator()</code> in <code>integral_constant</code> template of <code>constexpr</code> type	Yes
C++2014-N3637	Relation between <code>std::async</code> and destructor of <code>std::future</code>	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3638	Automatic deduction of return type for functions where an explicit return type is not specified	Yes. In some cases, Code Prover can show compilation error
C++2014-N3642	Suffixes for user-defined literals indicating time (h, min, s, ms, us, ns) and strings (s)	Yes
C++2014-N3648	Initialization of captured members in lambda functions	Yes. In some cases, during initialization, Code Prover can call the corresponding constructors more number of times than necessary.

C++14 Std Ref	Description	Supported
C++2014-N3649	Generic (polymorphic) lambda expressions: <ul style="list-style-type: none"> Using <code>auto</code> type-specifier for parameter and return type Conversion of generic capture-less lambda expressions to pointer-to-function. 	Yes
C++2014-N3651	Variable templates	Yes
C++2014-N3652	Declarations, conditions and loops in <code>constexpr</code> functions.	Yes
C++2014-N3653	Initialization of aggregate classes with fewer initializers than members For instance, this initialization has fewer initializers than members. The member <code>c</code> is initialized with the value <code>0</code> and <code>d</code> is initialized with the value <code>s</code> . <pre> struct S { int a; const char* b; int c; int d = b[a];}; S ss = { 1, "asdf" }; </pre>	Yes
C++2014-N3654	<code>std::quoted</code>	Yes
C++2014-N3656	<code>std::make_unique</code>	Yes
C++2014-N3658	<code>std::integer_sequence</code>	Yes
C++2014-N3658	<code>std::shared_lock</code>	No. The use of <code>std::shared_lock</code> does not cause compilation error but the construct is not semantically supported.

C++14 Std Ref	Description	Supported
C++2014-N3664	Calling <code>new</code> and <code>delete</code> operators in batches.	This C++14 feature clarifies how successive calls to the <code>new</code> operator are implemented. No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3668	<code>std::exchange</code>	Partially supported.
C++2014-N3670	Using <code>std::get</code> with a data type to get one element in an <code>std::tuple</code> (provided there is only one element of the type in the tuple)	Yes
C++2014-N3671	Overloads for <code>std::equal</code> , <code>std::mismatch</code> and <code>std::is_permutation</code> function templates that accept two separate ranges	Yes
C++2014-N3733	Removal of <code>std::gets</code> from <code><cstdio></code>	Yes
C++2014-N3776	Wording change for destructor of <code>std::future</code>	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3779	<code>std::complex</code> literals representing pure imaginary numbers with suffix <code>i</code> , <code>if</code> or <code>il</code>	Yes
C++2014-N3781	Use of single quotation mark as digit separator, for instance, <code>1'000</code> .	Yes
C++2014-N3786	Prohibiting "out of thin air" results in C++14	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3910	Synchronizing behavior of signal handlers	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.

C++14 Std Ref	Description	Supported
C++2014-N3924	Discouraging use of rand()	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.
C++2014-N3927	Lock-free executions	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++14.

See Also

C++ standard version (-cpp-version)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” on page 5-7

Provide Standard Library Headers for Polyspace Analysis

Before Polyspace analyzes the code for bugs and run-time errors, it compiles your code. Even if the code compiles with your compiler, you can see compilation errors with Polyspace. If the error comes from a standard library function, it usually indicates that Polyspace is not using your compiler headers. To work around the errors, provide the path to your compiler headers.

This topic shows how to locate the standard library headers from your compiler. The code examples cause a compilation error that shows the location of the headers.

- To locate the folder containing your C compiler system headers, compile this C code by using your compilation toolchain:

```
float fopen(float f);
#include <stdio.h>
```

The code does not compile because the `fopen` declaration conflicts with the declaration inside `stdio.h`. The compilation error shows the location of your compiler implementation of `stdio.h`. Your C standard library headers are all likely to be in that folder.

- To locate the folder containing your C++ compiler system headers, compile this C++ code by using your compilation toolchain:

```
namespace std {
    float cin;
}
#include <iostream>
```

The code does not compile because the `cin` declaration conflicts with the declaration inside `iostream.h`. The compilation error shows the location of your compiler implementation of `iostream.h`. Your C++ standard library headers are all likely to be in that folder.

After you locate the path to your compiler's header files, specify the path for the Polyspace analysis. For C++ code, specify the paths to both your C and C++ headers.

- In the user interface (Polyspace desktop products), add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Bug Finder).

- At the command line, use the flag `-I` with the `polyspace-bug-finder`, `polyspace-code-prover`, `polyspace-bug-finder-server` or `polyspace-code-prover-server` command..

For more information, see `-I`.

See Also

More About

- “Errors from Conflicts with Polyspace Header Files” on page 11-49

Requirements for Project Creation from Build Systems

For automatic project creation from build systems, your build commands or makefiles must meet certain requirements.

Compiler Requirements

- Your compiler must be called locally.

If you use a compiler cache such as `ccache` or a distributed build system such as `distmake`, the software cannot trace your build. You must deactivate them.

- Your compiler must perform a clean build.

If your compiler performs only an incremental build, use appropriate options to build all your source files. For example, if you use `gmake`, append the `-B` or `-W` *makefileName* option to force a clean build. For the list of options allowed with the GNU® `make`, see `make` options.

- Your compiler configuration must be available to Polyspace. The compilers currently supported include the following:
 - arm Keil
 - Clang
 - Wind River® Diab
 - GNU C/C++
 - IAR Embedded Workbench
 - Green Hills®
 - NXP CodeWarrior®
 - Renesas®
 - Altium® Tasking
 - Texas Instruments™
 - tcc - Tiny C Compiler
 - Microsoft Visual C++®

If your compiler configuration is not available to Polyspace:

- Write a compiler configuration file for your compiler in a specific format. For more information, see “Compiler Not Supported for Project Creation from Build Systems” on page 11-8.
- Contact MathWorks Technical Support. For more information, see “Contact Technical Support” on page 11-5.
- With the TASKING compiler, if you use an alternative sfr file with extension `.asfr`, Polyspace might not be able to locate your file. If you encounter an error, explicitly `#include` your `.asfr` file in the preprocessed code using the option `Include (-include)`.

Typically, you use the statement `#include __SFRFILE__(__CPU__)` along with the compiler option `--alternative-sfr-file` to specify an alternative sfr file. The path to the file is typically `Tasking_C166_INSTALL_DIR\include\sfr\regCPU_NAME.asfr`. For instance, if your TASKING compiler is installed in `C:\Program Files\Tasking\C166-VX_v4.0r1\` and you use the CPU-related flag `-Cxc2287m_104f` or `--cpu=xc2287m_104f`, the path is `C:\Program Files\Tasking\C166-VX_v4.0r1\include\sfr\regxc2287m.asfr`.

Build Command Requirements

- Your build command must run to completion without any user interaction.
- In Linux, only UNIX shell (sh) commands must be used. If your build uses advanced commands such as commands supported only by bash, tcsh or zsh, Polyspace cannot trace your build.

In Windows, only DOS commands must be used. If your build uses advanced commands such as commands supported only by PowerShell or Cygwin™, Polyspace cannot trace your build. To see if Polyspace supports your build command, run the command from `cmd.exe` in Windows. For more information, see “Check if Polyspace Supports Build Scripts” on page 11-19.

- If you use statically linked libraries, Polyspace cannot trace your build. In Linux, you can install the full Linux Standard Base (LSB) package to allow dynamic linking. For example, on Debian® systems, install LSB with the command `apt-get install lsb`.
- Your build command must not use aliases.

The `alias` command is used in Linux to create an alternate name for commands. If your build command uses those alternate names, Polyspace cannot recognize them.

- Your build process must not use the `LD_PRELOAD` mechanism.

- Your build command must be executable completely on the current machine and must not require privileges of another user.

If your build uses `sudo` to change user privileges or `ssh` to remotely log in to another machine, Polyspace cannot trace your build.

- If your build command uses redirection with the `>` or `|` character, the redirection occurs after Polyspace traces the command. Therefore, Polyspace does not handle the redirection.

For example, if your command occurs as

```
command1 | command2
```

And you enter

```
polyspace-configure command1 | command2
```

When tracing the build, Polyspace traces the first command only.

- If the System Integrity Protection (SIP) feature is active on the operating system macOS El Capitan (10.11) or a later macOS version, Polyspace cannot trace your build command. Before tracing your build command, disable the SIP feature. You can reenale this feature after tracing the build command.
- If your computer hibernates during the build process, Polyspace might not be able to trace your build.

Note Your environment variables are preserved when Polyspace traces your build command.

See Also

`polyspace-configure`

Related Examples

- “Create Polyspace Analysis Configuration from Build Command” on page 2-2

Supported Keil or IAR Language Extensions

Polyspace analysis can interpret a subset of common C/C++ language constructs and extended keywords by default. For compiler-specific keywords, you must specify your choice of compiler. If you specify `keil` or `iar` for `Compiler` (`-compiler`), the Polyspace verification allows language extensions specific to the Keil or IAR compilers.

Special Function Register Data Type

Embedded control applications frequently read and write port data, set timer registers, and read input captures. To deal with these requirements without using assembly language, some microprocessor compilers define special data types such as `sfr` and `sbit`. Typical declarations are:

```
sfr A0 = 0x80;
sfr A1 = 0x81;
sfr ADCUP = 0xDE;
sbit EI = 0x80;
```

The declarations reside in header files such as `regxx.h` for the basic 80Cxxx micro processor. The declarations customize the compiler to the target processor.

You access a register or a port by using the `sfr` and `sbit` data as follows. However, these data types are not part of the C99 Standard.

```
int status,P0;

void main (void) {
    ADCUP = 0x08; /* Write data to register */
    A1 = 0xFF; /* Write data to Port */
    status = P0; /* Read data from Port */
    EI = 1; /* Set a bit (enable all interrupts) */
}
```

To analyze this type of code, use these options:

- `Compiler` (`-compiler`): Specify `keil` or `iar`.
- `Sfr type support` (`-sfr-types`): Specify the data type and size in bits.

The analysis then supports the Keil or IAR language extensions even if some structures, keywords, and syntax are not part of the C99 standard.

Keywords Removed During Preprocessing

Once you specify the Keil or IAR compiler, the analysis recognizes compiler-specific keywords in your code. If a keyword is not relevant for the analysis, it is removed from the source code during preprocessing.

If you disable the keyword and use it as an identifier instead, you can encounter a compilation error when you compile your code with Polyspace. See “Errors Related to Keil or IAR Compiler” on page 11-41.

These keywords are removed during preprocessing:

- Keil: `bdata`, `far`, `idata`, `huge`, `sdata`
- IAR: `saddr`, `reentrant`, `reentrant_idata`, `non_banked`, `plm`, `bdata`, `idata`, `pdata`, `code`, `xdata`, `xhuge`, `interrupt`, `__interrupt`, `__intrinsic`

The `data` keyword is not removed.

Remove or Replace Keywords Before Compilation

The Polyspace compiler strictly follows the ANSI® C99 Standard (ISO/IEC 9899:1999). If your compiler allows deviation from the Standard, the Polyspace compilation using default options cannot emulate your compiler. For instance, your compiler can allow certain non-ANSI keyword, which Polyspace does not recognize by default.

To emulate your compiler closely, you specify the “Target and Compiler” options. If you still get compilation errors from unrecognized keywords, you can remove or replace them only for the purposes of verification. The option `Preprocessor definitions (-D)` allows you to make simple substitutions. For complex substitutions, for instance to remove a group of space-separated keywords such as a function attribute, use the option `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

Remove Unrecognized Keywords

You can remove unsupported keywords from your code for the purposes of analysis. For instance, follow these steps to remove the `far` and `0x` keyword from your code (`0x` precedes an absolute address).

- 1 Save the following template as `C:\Polyspace\myTpl.pl`.

Content of myTpl.pl

```
#!/usr/bin/perl

#####
# Post Processing template script
#
#####
# Usage from GUI:
#
# 1) Linux: /usr/bin/perl PostProcessingTemplate.pl
# 2) Windows: polyspaceroot\sys\perl\win32\bin\perl.exe <pathtoscript>\
# PostProcessingTemplate.pl
#
#####

$version = 0.1;

$INFILE = STDIN;
```

```

$OUTFILE = STDOUT;

while (<$INFILE>)
{
    # Remove far keyword
    s/far//;

    # Remove "@ 0xFE1" address constructs
    s/\@s0x[A-F0-9]*//g;

    # Remove "@0xFE1" address constructs
    s/\@0x[A-F0-9]*//g;

    # Remove "@ ((unsigned)&LATD*8)+2" type constructs
    s/\@s\(\(unsigned\) \&[A-Z0-9]+\*8\) \+ \d//g;

    # DON'T DELETE LINE BELOW: Print the current processed line
    print $OUTFILE $_;
}

```

For reference, see a summary of Perl regular expressions.


Perl Regular Expressions

```

#####
# Metacharacter What it matches
#####
# Single Characters
# . Any character except newline
# [a-z0-9] Any single character in the set
# [^a-z0-9] Any character not in set
# \d A digit same as
# \D A non digit same as [^0-9]
# \w An Alphanumeric (word) character
# \W Non Alphanumeric (non-word) character
#
# Whitespace Characters
# \s Whitespace character
# \S Non-whitespace character
# \n newline
# \r return
# \t tab
# \f formfeed
# \b backspace

```

```
#
# Anchored Characters
# \B word boundary when no inside []
# \b non-word boundary
# ^ Matches to beginning of line
# $ Matches to end of line
#
# Repeated Characters
# x? 0 or 1 occurrence of x
# x* 0 or more x's
# x+ 1 or more x's
# x{m,n} Matches at least m x's and no more than n x's
# abc All of abc respectively
# to|be|great One of "to", "be" or "great"
#
# Remembered Characters
# (string) Used for back referencing see below
# \1 or $1 First set of parentheses
# \2 or $2 First second of parentheses
# \3 or $3 First third of parentheses
#####
# Back referencing
#
# e.g. swap first two words around on a line
# red cat -> cat red
# s/(\w+) (\w+)/$2 $1/;
#
#####
```

- 2 On the **Configuration** pane, select **Environment Settings**.
- 3 To the right of **Command/script to apply to preprocessed files**, click .
- 4 Use the Open File dialog box to navigate to C:\Polyspace.
- 5 In the **File name** field, enter myTpl.pl.
- 6 Click **Open**. You see C:\Polyspace\myTpl.pl in the **Command/script to apply to preprocessed files** field.

Remove Unrecognized Function Attributes

You can remove unsupported function attributes from your code for the purposes of analysis.

If you run verification on this code specifying a generic compiler, you can see compilation errors from the `noreturn` attribute. The code compiles using a GNU compiler.

```
void fatal () __attribute__ ((noreturn));

void fatal (/* ... */)
{
    /* ... */ /* Print error message. */ /* ... */
    exit (1);
}
```

If the software does not recognize an attribute and the attribute does not affect the code analysis, you can remove it from your code for the purposes of verification. For instance, you can use this Perl script to remove the `noreturn` attribute.

```
while ($line = <STDIN>)
{
    # __attribute__ ((noreturn))

    # Remove far keyword
    $line =~ s/__attribute__\ \(\(noreturn\)\)//g;

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

Specify the script using the option `Command/script` to apply to preprocessed files (`-post-preprocessing-command`).

See Also

Polyspace Analysis Options

`Command/script` to apply to preprocessed files (`-post-preprocessing-command`) | Preprocessor definitions (`-D`)

Related Examples

- “Troubleshoot Compilation Errors”

Gather Compilation Options Efficiently

Polyspace verification can sometimes stop in the compilation or linking phase due to the following reasons:

- The Polyspace compiler strictly follows a C or C++ Standard (depending on your choice of compiler). See “C/C++ Language Standard Used in Polyspace Analysis” on page 5-7. If your compiler allows deviation from the Standard, the Polyspace compilation using default options cannot emulate your compiler.
- Your compiler declares standard library functions with argument or return types different from the standard types. Unless you also provide the function definition, for efficient verification, Polyspace uses its own definitions of standard library functions, which have the usual prototype. The mismatch in types causes a linking error.

You can easily work around the compilation and standard library function errors. To work around the errors, you typically specify certain analysis options. In some cases, you might have to add a few lines to your code. For instance:

- To emulate your compiler behavior more closely, you specify the “Target and Compiler” options. If you still face compilation errors, you might have to remove or replace certain unrecognized keywords using the option `Preprocessor definitions (-D)`. However, the option allows only simple substitution of a string with another string. For more complex replacements, you might have to add `#define` statements to your code.
- To avoid errors from stubbing standard library functions, you might have to `#define` certain Polyspace-specific macros so that Polyspace does not use its own definition of standard library functions.

Instead of adding these modifications to your original code, create a single `polyspace.h` file that contains all modifications. Use the option `Include (-include)` to force inclusion of the `polyspace.h` file in all source files under verification.

Benefits of this approach include:

- The error detection is much faster since it will be detected during compilation rather than in the link or subsequent phases.
- There will be no need to modify original source files.
- The file is automatically included as the very first file in the original `.c` files.
- The file is reusable for other projects developed under the same environment.

Example 5.1. Example

This is an example of a file that can be used with the option `Include (-include)`.

```
// The file may include (say) a standard include file implicitly
// included by the cross compiler

#include <stdlib.h>
#include "another_file.h"

// Workarounds for compilation errors
#define far
#define at(x)

// Workarounds for errors due to redefining standard library functions
#define POLYSPACE_NO_STANDARD_STUBS // use this flag to prevent the
//automatic stubbing of std functions
#define __polyspace_no_sscanf
#define __polyspace_no_fgetc
void sscanf(int, char, char, char, char, char);
void fgetc(void);
```

See Also

More About

- “Troubleshoot Compilation Errors”

Configure Inputs and Stubbing Options

Specify External Constraints

This example shows how to specify constraints (also known as data range specifications or DRS) on variables in your code. Polyspace uses the code that you provide to make assumptions about items such as variable ranges and allowed buffer size for pointers. Sometimes the assumptions are broader than what you expect because:

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

Because of these broad assumptions:

- Code Prover can consider more execution paths than those paths that occur at run time. If an operation fails along one of the execution paths, Polyspace places an orange check on the operation. If that execution path comes from an assumption that is too broad, the orange check might indicate a false positive.
- Bug Finder can sometimes produce false positives.

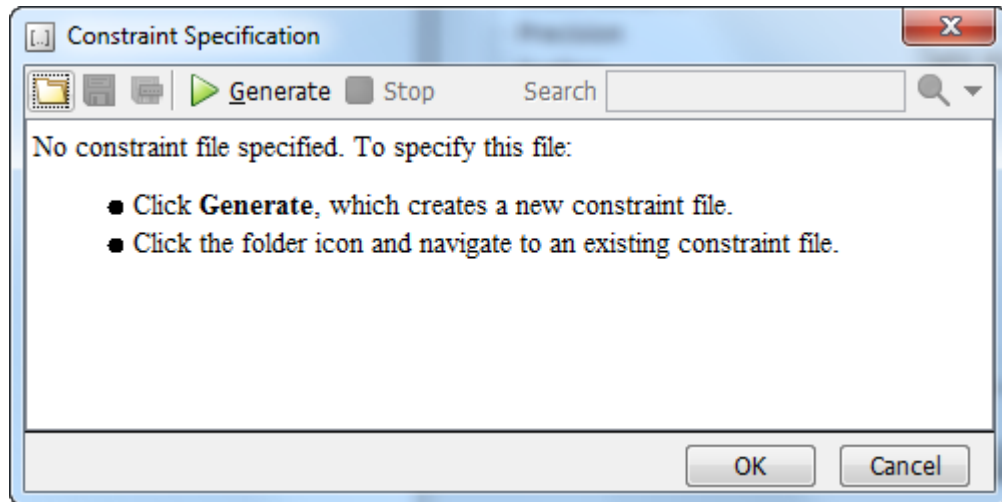
To reduce the number of such false positives, you can specify additional constraints on global variables, function inputs, and return values and modifiable arguments of stubbed functions. You save the constraints as an XML file to use them for subsequent analyses. If your source code changes, you can update the previous constraints. You do not have to create a new constraint template.

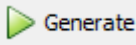
Note In Bug Finder, you can only constrain global variables. You cannot constrain function inputs or return values of stubbed functions.

Create Constraint Template

User Interface (Desktop Products Only)

- 1 Open the project configuration. On the **Configuration** pane, select **Inputs & Stubbing**.
- 2 To the right of **Constraint setup**, click the **Edit** button to open the **Constraint Specification** window.



- 3 In the Constraint Specification dialog box, create a blank constraint template. The template contains a list of all variables on which you can provide constraints. To create a new template, click . The software compiles your project and creates a template. The new template is stored in a file *Module_number_Project_name_drs_template.xml* in your project folder.
- 4 Specify your constraints and save the template as an XML file. For more information, see “External Constraints for Polyspace Analysis” on page 6-8.
- 5 Click **OK**.

You see the full path to the template XML file in the **Constraint setup** field. If you run an analysis, Polyspace uses this template for extracting variable constraints.

Command Line

Use the option `Constraint setup (-data-range-specifications)` to specify the constraints XML file.

To specify constraints in the XML file:

- 1 First, create a blank XML template. The template lists all global variables, function inputs and modifiable arguments and return values of stubbed functions without specifying any constraints on them.

To create a blank template, run an analysis only upto the compilation phase. In Bug Finder, disable checking of defects. Use the option `Find defects (-checkers)`. In Code Prover, check for source compliance only. Use the argument `compile` for the option `Verification level (-to)`. After the analysis, a blank template XML `drs-template.xml` is created in the results folder.

For C++ projects, to create a blank constraints template, you have to use the argument `cpp-normalize` for the option `Verification level (-to)`.

- 2 Edit the XML file to specify your constraints.


For examples, see:

- “Constrain Global Variable Range” (Polyspace Code Prover Server)
- “Constrain Function Inputs” (Polyspace Code Prover Server)

Create Constraint Template from Code Prover Analysis Results

You can constrain variable ranges based on their expected range in real-world applications. For instance, if a variable represents vehicle speed, you can set a maximum possible value. You can also constrain variable ranges only if they cause too many orange checks from overapproximation.

A Code Prover analysis shows all global variables, function inputs and stubbed functions that lead to orange checks from possible overapproximation. You can constrain only these variables for a more precise analysis.

- 1 Open Code Prover results in the Polyspace user interface or Polyspace Access web interface.
- 2 Open the **Orange Sources** pane. Do one of the following:
 - Select an orange check. If the software can trace an orange check to a root cause, a  icon appears on the **Result Details** pane. Click this icon to open the **Orange Sources** pane.
 - In the Polyspace user interface, select **Window > Show/Hide View > Orange Sources**. In the Polyspace Access web interface, select **Layout > Show/Hide View > Orange Sources**.


You see the full list of variables (function inputs or return values of stubbed functions) that can cause orange checks. Constrain the ranges of these variables.

Update Existing Template

With new code submissions, you might have to specify additional constraints. You can update an existing template to add global variables, function inputs and stubbed functions that come from the new code submissions.

Additionally, if you remove some variables or functions from your code, constraints on them are not applicable any more. Instead of regenerating a constraint template and respecifying the constraints, you can update an existing template and remove the variables that are not present in your code.

User Interface (Desktop Products Only)

- 1 On the **Configuration** pane, select **Inputs & Stubbing**.
- 2 Open the existing template in one of the following ways:
 - In the **Constraint setup** field, enter the path to the template XML file. Click **Edit**.
 - Click **Edit**. In the Constraint Specification dialog box, click the  icon to navigate to your template file.
- 3 Click **Update**.
 - a Variables that are no longer present in your source code appear under the **Non Applicable** node. To remove an entry under the **Non Applicable** node or the node itself, right-click and select **Remove This Node**.
 - b Specify your new constraints for any of the other variables.

Command Line

In a continuous integration workflow, you can use the constraints XML file from the previous run. If new code submissions require additional constraints:

- 1 Specify constraints on variables from new code submissions in a constraints XML file. See [Create Constraint Template: Command Line](#) on page 6-3.
- 2 Merge the constraints XML file with the new constraints and the constraints XML file from the previous run.

Specify Constraints in Code

Specifying constraints outside your code allows for more precise analysis. However, you must use the code within the specified constraints because the constraints are *outside* your code. Otherwise, the results might not apply. For example, if you use function inputs outside your specified range, a run-time error can occur on an operation even though checks on the operation are green.

To specify constraints *inside* your code, you can use:

- Appropriate error handling tests in your code.

Polyspace checks to determine if the errors can actually occur. If they do not occur, the test blocks appear as **Unreachable code**.

- The `assert` macro. For example, to constrain a variable `var` in the range `[0,10]`, you can use `assert(var >= 0 && var <=10);`.

Polyspace checks your `assert` statements to see if the condition can be false. Following the `assert` statement, Polyspace considers that the `assert` condition is true. Using `assert` statements, you can constrain your variables for the remaining code in the same scope. For examples, see [Assertion](#).

See Also

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

Related Examples

- “External Constraints for Polyspace Analysis” on page 6-8
- “Constrain Global Variable Range” (Polyspace Code Prover Server)

- “Constrain Function Inputs” (Polyspace Code Prover Server)
- “XML File Format for Constraints” on page 6-23

External Constraints for Polyspace Analysis

For a more precise analysis with Polyspace, you can specify external constraints on:

- Global Variables.
- User-defined Functions.

Constraints on user-defined functions do not apply to a Bug Finder analysis.

- Stubbed Functions.

Constraints on stubbed functions do not apply to a Bug Finder analysis.

For more information, see “Specify External Constraints” on page 6-2. For a partial list of limitations, see “Constraint Specification Limitations” on page 6-15.

In the user interface of the Polyspace desktop products, you can specify the constraints through a **Constraint Specification** window. The constraints are saved in an XML file that you can reuse for other projects.

Name	File	Attributes	Data ...	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated
Global Variables								
User Defined Functions								
isEndOfList()	file.c	unused		MAIN GENERATOR				
isEndOfList.arg1	file.c		int32		INIT	-1..0		
isEndOfList.arg2	file.c		int32 *		MAIN GENERATOR		Not NULL	MAIN GENERATOR
isEndOfList.* arg	file.c		int32		MAIN GENERATOR			
isEndOfList.return	file.c		int32					
Stubbed Functions								
Non Applicable								

This table explains the various columns in the **Constraint Specification** window. If you directly edit the constraint XML file to specify a constraint (for instance, in the Polyspace Server products), this table also shows the correspondence between columns in the user interface and entries in the XML file. The XML entry highlighted in bold appears in the corresponding column of the **Constraint Specification** window.

Column	Settings
Name	<p>Displays the list of variables and functions in your Project for which you can specify data ranges.</p> <p>This Column displays three expandable menu items:</p> <ul style="list-style-type: none"> • Globals - Displays global variables in the project. • User defined functions - Displays user-defined functions in the project. Expand a function name to see its inputs. • Stubbed functions - Displays a list of stub functions in the project. Expand a function name to see the inputs and return values. <p>XML File Entry:</p> <pre><function name = "funcName" ...> <scalar name = "arg1" ...> <pointer name = "arg2" ...></pre>
File	<p>Displays the name of the source file containing the variable or function.</p> <p>XML File Entry:</p> <pre><file name = "C:\Project1\Sources\file.c" ...></pre>
Attributes	<p>Displays information about the variable or function.</p> <p>For example, static variables display <code>static</code>. Uncalled functions display <code>unused</code>.</p> <p>XML File Entry:</p> <pre><function name="funcName" attributes="unused" ...></pre>
Data Type	<p>Displays the variable type.</p> <p>XML File Entry:</p> <pre><scalar name="arg1" complete_type="int32" ...></pre>

Column	Settings
<p>Main Generator Called</p>	<p>Applicable only for user-defined functions.</p> <p>Specifies whether the main generator calls the function:</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Main generator may call this function, depending on the value of the <code>-functions-called-in-loop</code> (C) or <code>-main-generator-calls</code> (C++) parameter. • NO - Main generator will not call this function. • YES - Main generator will call this function. <p>XML File Entry:</p> <pre><function name="funcName" main_generator_called="MAIN_GENERATOR" ...></pre>

Column	Settings
Init Mode	<p>Specifies how the software assigns a range to the variable:</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Variable range is assigned depending on the settings of the main generator options <code>-main-generator-writes-variables</code> and <code>-no-def-init-glob</code>. • IGNORE - Variable is not assigned to any range, even if a range is specified. • INIT - Variable is assigned to the specified range only at initialization, and keeps the range until first write. • PERMANENT - Variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the <code>globalassert</code> mode if you need a warning. <p>User-defined functions support only INIT mode.</p> <p>Stub functions support only PERMANENT mode.</p> <p>For C verifications, global pointers support MAIN GENERATOR, IGNORE, or INIT mode.</p> <ul style="list-style-type: none"> • MAIN GENERATOR - Pointer follows the options of the main generator. • IGNORE - Pointer is not initialized • INIT - Specify if the pointer is NULL, and how the pointed object is allocated (Initialize Pointer and Init Allocated options). <p>XML File Entry:</p> <pre><scalar name="arg1" init_mode="INIT" ...></pre>

Column	Settings
<p>Init Range</p>	<p>Specifies the minimum and maximum values for the variable.</p> <p>You can use the keywords <code>min</code> and <code>max</code> to denote the minimum and maximum values of the variable type. For example, for the type <code>long</code>, <code>min</code> and <code>max</code> correspond to -2^{31} and $2^{31}-1$ respectively.</p> <p>You can also use hexadecimal values. For example: <code>0x12..0x100</code></p> <p>For enum variables, you cannot specify ranges directly using the enumerator constants. Instead use the values represented by the constants.</p> <p>For enum variables, you can also use the keywords <code>enum_min</code> and <code>enum_max</code> to denote the minimum and maximum values that the variable can take. For example, for an enum variable of the type defined below, <code>enum_min</code> is 0 and <code>enum_max</code> is 5:</p> <pre>enum week{ sunday, monday=0, tuesday, wednesday, thursday, friday, saturday};</pre>
	<p>XML File Entry:</p> <pre><scalar name="arg1" init_range="-1..0"...></pre>
<p>Initialize Pointer</p>	<p>Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p>Specifies whether the pointer should be NULL:</p> <ul style="list-style-type: none"> • May-be NULL - The pointer could potentially be a NULL pointer (or not). • Not Null - The pointer is never initialized as a null pointer. • Null - The pointer is initialized as NULL. <p>Note Not applicable for C++ projects. See "Constraint Specification Limitations" on page 6-15.</p>
	<p>XML File Entry:</p> <pre><pointer name="arg1" initialize_pointer="Not NULL"...></pre>

Column	Settings
Init Allocated	<p data-bbox="486 302 1262 361">Applicable only to pointers. Enabled only when you specify Init Mode:INIT.</p> <p data-bbox="486 392 1035 421">Specifies how the pointed object is allocated:</p> <ul data-bbox="486 447 1332 670" style="list-style-type: none"><li data-bbox="486 447 1332 506">• MAIN GENERATOR - The pointed object is allocated by the main generator.<li data-bbox="486 522 969 552">• None - Pointed object is not written.<li data-bbox="486 567 1332 626">• SINGLE - Write the pointed object or the first element of an array. (This setting is useful for stubbed function parameters.)<li data-bbox="486 642 1184 670">• MULTI - All objects (or array elements) are initialized. <hr/> <p data-bbox="486 730 1332 789">Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 6-15.</p> <hr/> <p data-bbox="486 805 698 835">XML File Entry:</p> <pre data-bbox="486 861 1267 887"><pointer name="arg1" init_pointed="MAIN_GENERATOR"...></pre>

Column	Settings
# Allocated Objects	<p>Applicable only to pointers.</p> <p>Specifies how many objects are pointed to by the pointer (the pointed object is considered as an array).</p> <p>The Init Allocated parameter specifies how many allocated objects are actually initialized. For instance, consider this code:</p> <pre>void func(int *ptr) { assert(ptr[0]==1); assert(ptr[1]==1); }</pre> <p>If you specify these constraints:</p> <ul style="list-style-type: none"> • <code>ptr</code> has Init Allocated set to MULTI and # Allocated Objects set to 2, • <code>*ptr</code> has Init Range set to 1..1, <p>both assertions are green. However, if you specify these constraints:</p> <ul style="list-style-type: none"> • <code>ptr</code> has Init Allocated set to SINGLE • <code>*ptr</code> has Init Range set to 1..1, <p>the second assertion is orange. Only the first object that <code>ptr</code> points to initialized to 1. Objects beyond the first can be potentially full range.</p> <hr/> <p>Note Not applicable for C++ projects. See “Constraint Specification Limitations” on page 6-15.</p> <hr/> <p>XML File Entry:</p> <pre><pointer name="arg1" number_allocated="10"...></pre>
Global Assert	<p>Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.</p> <hr/> <p>XML File Entry:</p> <pre><scalar name="glob" global_assert="YES"...></pre>

Column	Settings
Global Assert Range	Specifies the minimum and maximum values for the range you want to check.
	XML File Entry: <pre><scalar name="glob" assert_range="0..200" ...></pre>
Comment	Remarks that you enter, for example, justification for your DRS values.
	XML File Entry: <pre><scalar name="glob" comment="Speed Range" ...></pre>

Constraint Specification Limitations

You cannot specify these constraints:

- *C++ Pointers cannot be constrained:*

In C++, you cannot constrain pointer arguments of functions. Functions that have pointer arguments only do not appear in the constraint specification interface.

Because of polymorphism, a C++ pointer can point to objects of multiple classes in a class hierarchy and can require invoking different constructors. The pre-analysis for constraint specification cannot determine which object type to constrain or which constructor to call.

- *Constraints cannot be relations:*

You cannot specify a constraint that relates the return value of a function to its inputs. You can only specify a constant range for the constraints.

- *Multiple ranges not possible:*

You cannot specify multiple ranges for a constraint. For instance, you cannot specify that a function argument has either the value -1 or a value in the range [1,100]. Instead, specify the range [-1,100] or perform two separate analyses, once with the value -1 and once with the range [1,100].

See Also

More About


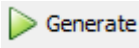
- “Specify External Constraints” on page 6-2

Constrain Global Variable Range

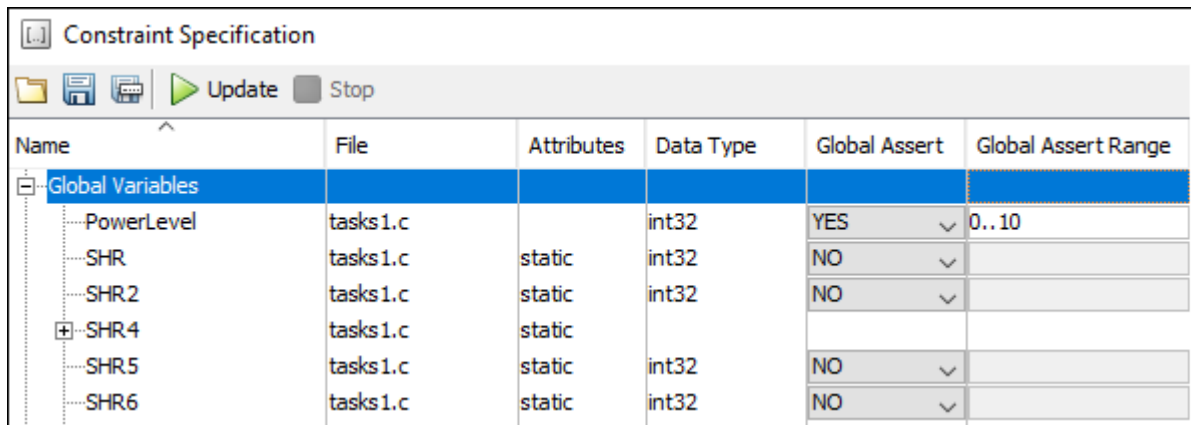
You can impose constraints (also known as data range specifications or DRS) on the range of a global variable and check with Code Prover whether write operations on the variable violate the constraint. For the general workflow, see “Specify External Constraints” on page 6-2.

User Interface (Desktop Products Only)

To constrain a global variable range and also check for violation of the constraint:

- 1 In your project configuration, select **Inputs & Stubbing**. Click the  button next to the **Constraint setup** field.
- 2 In the Constraint Specification window, click .

Under the **Global Variables** node, you see a list of global variables.




The screenshot shows the 'Constraint Specification' window with a table of global variables. The table has columns for Name, File, Attributes, Data Type, Global Assert, and Global Assert Range. The 'Global Variables' node is expanded, showing a list of variables including PowerLevel, SHR, SHR2, SHR4, SHR5, and SHR6.

Name	File	Attributes	Data Type	Global Assert	Global Assert Range
Global Variables					
PowerLevel	tasks1.c		int32	YES	0..10
SHR	tasks1.c	static	int32	NO	
SHR2	tasks1.c	static	int32	NO	
SHR4	tasks1.c	static			
SHR5	tasks1.c	static	int32	NO	
SHR6	tasks1.c	static	int32	NO	

- 3 For the global variable that you want to constrain:

- From the drop-down list in the **Global Assert** column, select YES.
- In the **Global Assert Range** column, enter the range in the format *min* . *max*. *min* is the minimum value and *max* the maximum value for the global variable.

4 To save your specifications, click the  button.

In **Save a Constraint File** window, save your entries as an xml file.

5 Run a verification and open the results.

For every write operation on the global variable, you see a green, orange, or red **Correctness condition** check. If the check is:

- Green, the variable is within the range that you specified.
- Orange, the variable can be outside the range that you specified.
- Red, the variable is outside the range that you specified.

When two or more tasks write to the same global variable, the **Correctness condition** check can appear orange on all write operations to the variable even when only one write operation takes the variable outside the **Global Assert** range.

Command Line

Use the option `Constraint setup (-data-range-specifications)` with an XML file specifying your constraint.

For instance, for an analysis with Polyspace Code Prover Server, specify the option as follows:

```
polyspace-code-prover-server -sources filename
                             -data-range-specifications "C:\Polyspace\drs_project1.xml"
```

Create a blank constraint XML template as described in “Specify External Constraints” on page 6-2. In the XML file, locate and constrain the global variables. XML tags for global variables appear directly within the `file` tag without an enclosing function tag. For instance, in this constraint XML, `PowerLevel` and `SHR` are global variables:

```
<file name="\\\\home\\Polyspace_Workspace\\Examples\\Code_Prover_Example\\sources\\task1.xml">
  <scalar name="PowerLevel" line="26" .. global_assert="YES" assert_range="0..10"/>
  <scalar name="SHR" line="30" ... global_assert="NO" assert_range="" />
</file>
```

```
<function name="Tserver" line="73" .../>
<function name="initregulate" line="47" .../>
<function name="orderregulate" line="35" ...>
  <scalar name="return" ... global_assert="unsupported" assert_range="unsupported"
</function>
<function name="proc1" line="101" .../>
</file>
```

To specify a constraint on a global variable and check during a Code Prover analysis if the constraint is violated:

- 1 Set the `global_assert` attribute of the variable's `scalar` tag to YES.
- 2 Set the `assert_range` attribute to a range in the form *min* . *max*, for instance, 0..10.

In the preceding example, the variable `PowerLevel` is constrained this way.

See Also

Polyspace Analysis Options

Constraint setup (-data-range-specifications)

Polyspace Results

Correctness condition

More About

- “Specify External Constraints” on page 6-2
- “External Constraints for Polyspace Analysis” on page 6-8
- “Constrain Function Inputs” (Polyspace Code Prover Server)

Constrain Function Inputs

For a more precise Code Prover analysis, you can specify constraints (also known as data range specifications or DRS) on function inputs. Code Prover checks your function definition for run-time errors with respect to the constrained inputs. For the general workflow, see “Specify External Constraints” on page 6-2.

For instance, for a function defined as follows, you can specify that the argument `val` has values in the range `[1..10]`. You can also specify that the argument `ptr` points to a 3-element array where each element is initialized:

```
int func(int val, int* ptr) {  
    :  
    :  
}
```

User Interface (Desktop Products Only)

To specify constraints on function inputs:

1 In your project configuration, select **Inputs & Stubbing**. Click the  button for **Constraint setup**.

2 In the Constraint Specification window, click .

Under the **User Defined Functions** node, you see a list of functions whose inputs can be constrained.

3 Expand the node for each function.

You see each function input on a separate row. The inputs have the syntax `function_name.arg1`, `function_name.arg2`, etc.

4 Specify your constraints on one or more of the function inputs. For more information, see “External Constraints for Polyspace Analysis” on page 6-8.

For example, in the preceding code:

- To constrain `val` to the range `[1..10]`, select **INIT** for **Init Mode** and enter `1..10` for **Init Range**.

- To specify that `ptr` points to a 3-element array where each element is initialized, select **MULTI** for **Init Allocated** and enter 3 for **# Allocated Objects**.

Name	File	Attributes	Data Type	Main Generator Called	Init Mode	Init Range	Initialize Pointer	Init Allocated	# Allocated Objects
func()	file.c	unused		MAIN GENERATOR	INIT	1..10			
func.arg1	file.c		int32		INIT				
func.arg2	file.c		int32 *		INIT		Not NULL	MULTI	3
func.*arg2	file.c		int32		MAIN GENE...				
func.return	file.c		int32						

- Run verification and open the results. On the **Source** pane, place your cursor on the function inputs.

The tooltips display the constraints. For example, in the preceding code, the tooltip displays that `val` has values in `1..10`.

Command Line

Use the option `Constraint setup (-data-range-specifications)` with an XML file specifying your constraint.

For instance, for an analysis with Polyspace Code Prover Server, specify the option as follows:

```
polyspace-code-prover-server -sources filename
                             -data-range-specifications "C:\Polyspace\drs_project1.xml"
```

Create a blank constraint XML template as described in “Specify External Constraints” on page 6-2. In the XML file, locate and constrain the function inputs. The function inputs appear as a `scalar` or `pointer` tag in a `function` tag. The inputs are named as `arg1`, `arg2` and so on. For instance, for the preceding code, the XML structure for the inputs of `func` appear as follows:

```
<function name="func" line="1" attributes="unused"
          main_generator_called="MAIN_GENERATOR" comment="">
```

```
<scalar name="arg1" line="1" base_type="int32"
  complete_type="int32" init_mode="INIT" init_range="1..10" />
<pointer name="arg2" line="1" complete_type="int32 *"
  init_mode="INIT" initialize_pointer="Not NULL" number_allocated="3"
  init_pointed="MULTI">
  <scalar line="1" base_type="int32" complete_type="int32"
    init_mode="MAIN_GENERATOR" init_range="" />
</pointer>
<scalar name="return" line="1" base_type="int32" complete_type="int32"
  init_mode="disabled" init_range="disabled"/>
</function>
```

To specify a constraint on a function input, set the attributes `init_mode` and `init_range` for scalar variables, and `init_pointed` and `number_allocated` for pointer variables.

- To constrain `val` to the range `[1..10]`, set the `init_mode` attribute of the tag with name `arg1` to `INIT` and `init_range` to `1..10`.
- To specify that `ptr` points to a 3-element array where each element is initialized, set the `init_mode` attribute of the tag with name `arg2` to `INIT`, `init_pointed` to `MULTI` and `number_allocated` to `3`.

See Also

Constraint setup (-data-range-specifications)

More About

- “Specify External Constraints” on page 6-2
- “External Constraints for Polyspace Analysis” on page 6-8
- “Constrain Global Variable Range” (Polyspace Code Prover Server)

XML File Format for Constraints

For a more precise Polyspace analysis, you can specify constraints on global variables, function inputs and stubbed functions. You can specify the constraints in the user interface of the Polyspace desktop products or at the command line as an XML file. For the general workflow, see “Specify External Constraints” on page 6-2.

This topic describes details of the constraint XML file schema. You typically require this information only if you create a constraint XML from scratch. If you run a verification once, the software automatically generates a template constraint file `drs-template.xml` in your results folder. Instead of creating a constraint XML file from scratch, it is easier to edit this template XML file to specify your constraints. For some examples, see:

- “Constrain Global Variable Range” (Polyspace Code Prover Server)
- “Constrain Function Inputs” (Polyspace Code Prover Server)

For another explanation of what the XML tags mean, see “External Constraints for Polyspace Analysis” on page 6-8.

You can also see the information in this topic and the underlying XML schema in *polyspaceroot*\polyspace\drs. Here, *polyspaceroot* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.

Syntax Description — XML Elements

The constraints file contains the following XML elements:

- `<global>` element — Declares the global scope, and is the root element of the XML file.
- `<file>` element — Declares a file scope. Must be enclosed in the `<global>` element. May enclose any variable or function declaration. Static variables must be enclosed in a file element to avoid conflicts.
- `<scalar>` element— Declares an integer or a floating point variable. May be enclosed in any recognized element, but cannot enclose any element. Sets init/permanent/global asserts on variables.
- `<pointer>` element — Declares a pointer variable. May enclose any other variable declarations (including itself), to define the pointed objects. Specifies what value is written into pointer (NULL or not), how many objects are allocated and how the pointed objects are initialized.

- `<array>` element — Declares an array variable. May enclose any other variable definition (including itself), to define the members of the array.
- `<struct>` element — Declares a structure variable or object (instance of class). May enclose any other variable definition (including itself), to define the fields of the structure.
- `<function>` element — Declares a function or class method scope. May enclose any variable definition, to define the arguments and the return value of the function. Arguments should be named *arg1*, *arg2*, ..., *argn* and the return value should be called *return*.

The following notes apply to specific fields in each XML element:

- **(*)** — Fields used only by the GUI. These fields are not mandatory for verification to accept the ranges. The field line contains the line number where the variable is declared in the source code, `complete_type` contains a string with the complete variable type, and `base_type` is used by the GUI to compute the min and max values. The field comment is used to add information about any node.
- **(**)** — The field name is mandatory for scope elements `<file>` and `<function>` (except for function pointers). For other elements, the name must be specified when declaring a root symbol or a `struct` field.
- **(***)** — If more than one attribute applies to the variable, the attributes must be separated by a space. Only the static attribute is mandatory, to avoid conflicts between static variables having the same name. An attribute can be defined multiple times without impact.
- **(****)** — This element is used only by the GUI, to determine which `init` modes are allowed for the current element (according to its type). The value works as a mask, where the following values are added to specify which modes are allowed:
 - **1**: The mode “NO” is allowed.
 - **2**: The mode “INIT” is allowed.
 - **4**: The mode “PERMANENT” is allowed.
 - **8**: The mode “MAIN_GENERATOR” is allowed.

For example, the value “**10**” means that modes “INIT” and “MAIN_GENERATOR” are allowed. To see how this value is computed, refer to “Valid Modes and Default Values” on page 6-28.

- **(*****)** — A sub-element of a pointer (i.e. a pointed object) will be taken into account only if `init_pointed` is equal to `SINGLE`, `MULTI`, `SINGLE_CERTAIN_WRITE` or `MULTI_CERTAIN_WRITE`.

- (*****) — SINGLE_CERTAIN_WRITE or MULTI_CERTAIN_WRITE are available for parameters and return values of stubbed functions only if they are pointers. If the parameter or return value is a structure and the structure has a pointer field, they are also available for the pointer field.

<file> Element

Field	Syntax
name	<i>filepath_or_filename</i>
comment	<i>string</i>

<scalar> Element

Field	Syntax
name (**)	<i>name</i>
line (*)	<i>line</i>
base_type (*)	intx uintx floatx
Attributes (***)	volatile extern static const
complete_type (*)	<i>type</i>
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	<i>single value (****)</i>
init_range	<i>range</i> disabled unsupported

Field	Syntax
global_assert	YES NO disabled unsupported
assert_range	<i>range</i> disabled unsupported
comment(*)	<i>string</i>

<pointer> Element

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
Attributes (***)	volatile extern static const
complete_type (*)	<i>type</i>
init_mode	MAIN_GENERATOR IGNORE INIT PERMANENT disabled unsupported
init_modes_allowed (*)	<i>single value (****)</i>
initialize_pointer	May be: NULL Not NULL NULL
number_allocated	<i>single value</i> disabled unsupported

Field	Syntax
init_pointed (*****)	MAIN_GENERATOR NONE SINGLE MULTI SINGLE_CERTAIN_WRITE MULTI_CERTAIN_WRITE disabled
comment	<i>string</i>

<array> and <struct> Elements

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>
complete_type (*)	<i>type</i>
attributes (***)	volatile extern static const
comment	<i>string</i>

<function> Element

Field	Syntax
Name (**)	<i>name</i>
line (*)	<i>line</i>

Field	Syntax
main_generator_called	MAIN_GENERATOR YES NO disabled
attributes (***)	static extern unused
comment	<i>string</i>

Valid Modes and Default Values

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
Global variables	Base type	Unqualified/ static/ const scalar	MAIN_GENERATOR IGNORE INIT PERMANENT	YES NO			Main generator dependant
		Volatile scalar	PERMANENT	disabled			PERMANENT min..max
		Extern scalar	INIT PERMANENT	YES NO			INIT min..max
	Struct	Struct field	Refer to field type				
	Array	Array element	Refer to element type				
Global variables	Pointer	Unqualified/ static/ const scalar	MAIN_GENERATOR IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	Main generator dependant
		Volatile pointer	un-supported		un-supported	un-supported	

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		Extern pointer	IGNORE INIT		May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Pointed volatile scalar	un-supported	un-supported			
		Pointed extern scalar	INIT	un-supported			INIT min..max
		Pointed other scalars	MAIN_ GENERATOR INIT	un-supported			MAIN_ GENERATOR dependant
		Pointed pointer	MAIN_ GENERATOR INIT/	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	MAIN_ GENERATOR dependant
		Pointed function	un-supported	un-supported			
Function parameters	Userdefined functions	Scalar parameters	MAIN_ GENERATOR INIT	un-supported			INIT min..max
		Pointer parameters	MAIN_ GENERATOR INIT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI	INIT May be NULL max MULTI
		Other parameters	Refer to parameter type				
	Stubbed function	Scalar parameter	disabled	un-supported			

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		Pointer parameters	disabled		disabled	NONE SINGLE MULTI SINGLE_CERTAIN_WRITE MULTI_CERTAIN_WRITE	MULTI
		Pointed parameters	PERMANENT	un-supported			PERMANENT min..max
		Pointed const parameters	disabled	un-supported			
Function return	Userdef function	Return	disabled	un-supported	disabled	disabled	
	Stubbed function	Scalar return	PERMANENT	un-supported			PERMANENT min..max

Scope	Type		Init modes	Gassert mode	Initialize pointer	Init allocated	Default
		Pointer return	PERMANENT	un-supported	May be NULL Not NULL NULL	NONE SINGLE MULTI SINGLE_CERTAIN_WRITE MULTI_CERTAIN_WRITE	PERMANENT May be NULL max MULTI

See Also

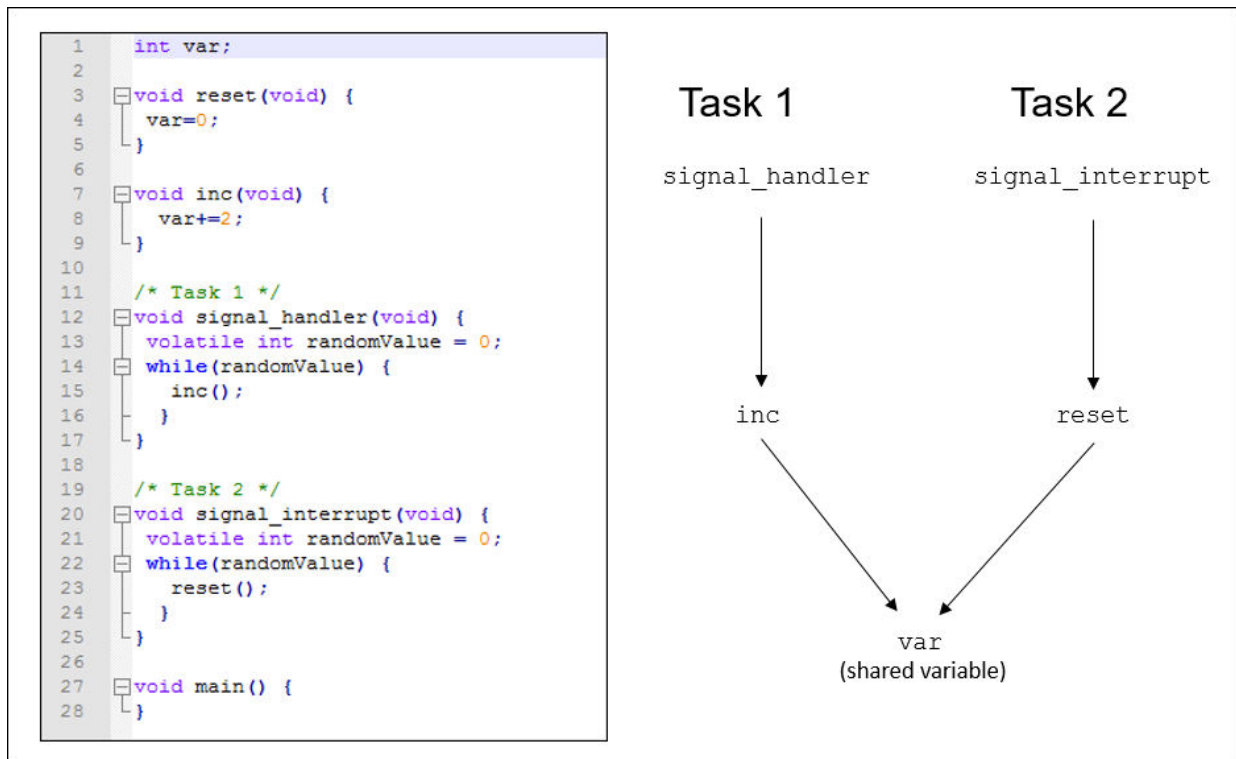
More About

- “Specify External Constraints” on page 6-2
- “Constrain Global Variable Range” (Polyspace Code Prover Server)
- “Constrain Function Inputs” (Polyspace Code Prover Server)

Configure Multitasking Analysis

Analyze Multitasking Programs in Polyspace

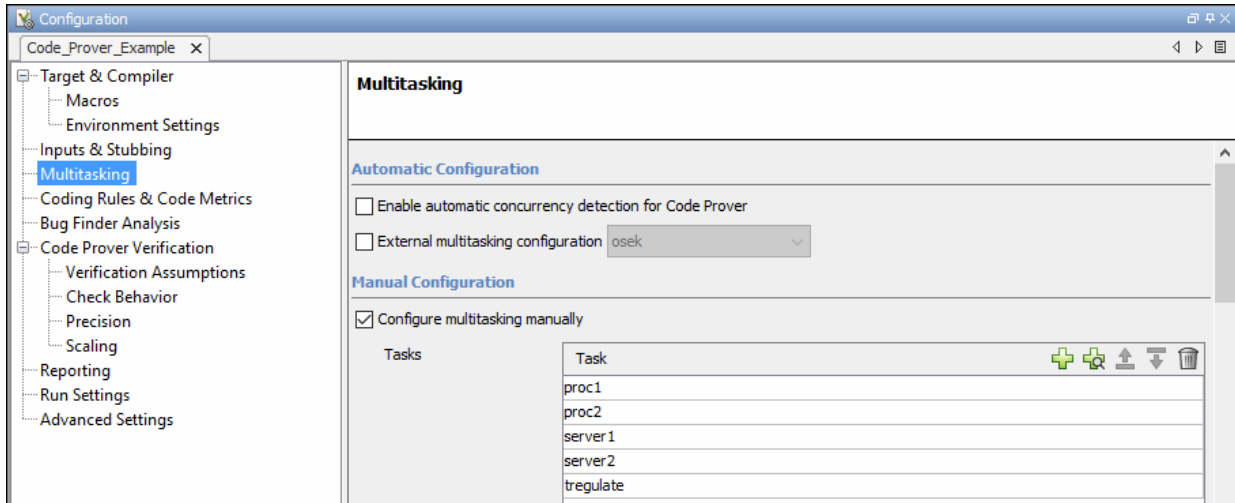
With Polyspace, you can analyze programs where multiple threads (tasks) run concurrently.



In addition to regular run-time checks, the analysis looks for issues specific to concurrent execution:

- Data races, deadlocks, consecutive or missing locks and unlocks (Bug Finder)
- Unprotected shared variables (Code Prover)

Configure Analysis



If your code uses multitasking primitives from certain families, for instance, `pthread_create` for thread creation:

- In Bug Finder, the analysis detects them and extracts your multitasking model from the code.
- In Code Prover, you must enable this automatic detection explicitly.

See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 7-6.

Alternatively, define your multitasking model through the analysis options. In the user interface, the options are on the **Multitasking** node in the **Configuration** pane. For more information, see “Configuring Polyspace Multitasking Analysis Manually” on page 7-19.

Review Analysis Results

Bug Finder

The screenshot displays the Polyspace Bug Finder interface. On the left, the 'Results List' pane shows a hierarchy of defects under 'Defect 249', including 'Concurrency 9' and 'Data race 2'. The 'Data race 2' defect is selected, showing two instances with 'Impact: High' and 'Certain operations'. On the right, the 'Result Details' pane shows the 'Result Review' section with 'Status' set to 'Unreviewed' and 'Severity' set to 'Unset'. A yellow highlight indicates a 'Data race (Impact: High)' with the description: 'Certain operations on variable 'bad_glob1' can interfere with each other and cause unpredictable values.' Below this, a table provides details for the conflicting accesses.

Access	Access Protections	Task	File	Scope	Line
Write	No protection	bug_datarace_task1()	concurrency.c	bug_datarace_task1()	57
Read	No protection	bug_datarace_task2()	concurrency.c	bug_datarace_task2()	62


The Bug Finder analysis shows concurrency defects such as data races and deadlocks. See “Concurrency Defects” (Polyspace Bug Finder Access).

Code Prover

The screenshot shows two panes from the Code Prover interface. The left pane, 'Results List', displays a tree view of variables. Under 'Shared', 'Potentially unprotected variable' is expanded, showing 'Variable: PowerLevel' in 'tasks1.' with a red checkmark icon. The right pane, 'Result Details', shows a 'Potentially unprotected variable' warning for 'tasks1.PowerLevel'. Below the warning is a table of conflicting operations:

Event	File	Scope	Line
Written value: -10000	main.c	main()	36
Written value: 0	tasks1.c	_init_globals()	26
Written value: [-2147483639 .. 2 ³¹ -1]	tasks2.c	Increase_PowerLevel()	19
Read value: [-2147483640 .. 2 ³¹ -1]	tasks1.c	orderregulate()	40
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Increase_PowerLevel()	19
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Compute_Injection()	34
Read value: [-2147483640 .. 2 ³¹ -1]	tasks2.c	Get_PowerLevel()	41

The Code Prover analysis exhaustively checks if shared global variables are protected from concurrent access. See “Global Variables” (Polyspace Code Prover Access).

Review the results using the message on the **Result Details** pane. See a visual representation of conflicting operations using the  (graph) icon.

See Also

More About

- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 7-6
- “Configuring Polyspace Multitasking Analysis Manually” on page 7-19
- “Protections for Shared Variables in Multitasking Code” on page 7-24

Auto-Detection of Thread Creation and Critical Section in Polyspace

With Polyspace, you can analyze programs where multiple threads run concurrently. Polyspace can analyze your multitasking code for data races, deadlocks and other concurrency defects, if the analysis is aware of the concurrency model in your code. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. Bug Finder detects them by default. In Code Prover, you enable automatic detection using the option `Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)`.

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 7-2.

If your thread creation function is not detected automatically:

- You can also map the function to a thread-creation function that Polyspace can detect automatically. Use the option `-function-behavior-specifications`.
- Otherwise, you must manually model your multitasking threads by using configuration options. See “Configuring Polyspace Multitasking Analysis Manually” on page 7-19.

Multitasking Routines that Polyspace Can Detect

Polyspace can detect thread creation and critical sections if you use primitives from these groups. Polyspace recognizes calls to these routines as the creation of a new thread or as the beginning or end of a critical section.

POSIX

Thread creation: `pthread_create`

Critical section begins: `pthread_mutex_lock`

Critical section ends: `pthread_mutex_unlock`

VxWorks

Thread creation: `taskSpawn`

Critical section begins: `semTake`

Critical section ends: `semGive`

To activate automatic detection of concurrency primitives for VxWorks®, in the user interface of the Polyspace desktop products, use the VxWorks template. For more information on templates, see “Create Project Using Configuration Template” (Polyspace Bug Finder). At the command-line, use these options:

```
-D1=CPU=I80386  
-D2=__GNUC__=2  
-D3=__OS_VXWORKS
```

Concurrency detection is possible only if the multitasking functions are created from an entry point named `main`. If the entry point has a different name, such as `vxworks_entry_point`, do one of the following:

- Provide a `main` function.
- Preprocessor definitions (-D): In preprocessor definitions, set `vxworks_entry_point=main`.

Windows

Thread creation: `CreateThread`

Critical section begins: `EnterCriticalSection`

Critical section ends: `LeaveCriticalSection`

µC/OS II

Thread creation: `OSTaskCreate`

Critical section begins: `OSMutexPend`

Critical section ends: `OSMutexPost`

C++11

Thread creation: `std::thread::thread`

Critical section begins: `std::mutex::lock`

Critical section ends: `std::mutex::unlock`

For autodetection of C++11 threads, explicitly specify paths to your compiler header files or use `polyspace-configure`.

For instance, if you use `std::thread` for thread creation, explicitly specify the path to the folder containing `thread.h`.

See also “Limitations of Automatic Thread Detection” on page 7-13.

C11

Thread creation: `thr_create`

Critical section begins: `mtx_lock`

Critical section ends: `mtx_unlock`

Example of Automatic Thread Detection

The following multitasking code models five philosophers sharing five forks. The example uses POSIX® thread creation routines and illustrates a classic example of a deadlock. Run Bug Finder on this code to see the deadlock.

```
#include "pthread.h"
#include <stdio.h>
#include <unistd.h>

pthread_mutex_t forks[5];

void* philo1(void* args)
{
    while (1) {
        printf("Philosopher 1 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[0]);
        printf("Philosopher 1 takes left fork\n");
        pthread_mutex_lock(&forks[1]);
        printf("Philosopher 1 takes right fork\n");
        printf("Philosopher 1 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[1]);
        printf("Philosopher 1 puts down right fork\n");
        pthread_mutex_unlock(&forks[0]);
        printf("Philosopher 1 puts down left fork\n");
    }
    return NULL;
}

void* philo2(void* args)
{
    while (1) {
        printf("Philosopher 2 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[1]);
        printf("Philosopher 2 takes left fork\n");
        pthread_mutex_lock(&forks[2]);
        printf("Philosopher 2 takes right fork\n");
        printf("Philosopher 2 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[2]);
        printf("Philosopher 2 puts down right fork\n");
        pthread_mutex_unlock(&forks[1]);
        printf("Philosopher 2 puts down left fork\n");
    }
    return NULL;
}
```

```
void* philo3(void* args)
{
    while (1) {
        printf("Philosopher 3 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[2]);
        printf("Philosopher 3 takes left fork\n");
        pthread_mutex_lock(&forks[3]);
        printf("Philosopher 3 takes right fork\n");
        printf("Philosopher 3 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[3]);
        printf("Philosopher 3 puts down right fork\n");
        pthread_mutex_unlock(&forks[2]);
        printf("Philosopher 3 puts down left fork\n");
    }
    return NULL;
}

void* philo4(void* args)
{
    while (1) {
        printf("Philosopher 4 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[3]);
        printf("Philosopher 4 takes left fork\n");
        pthread_mutex_lock(&forks[4]);
        printf("Philosopher 4 takes right fork\n");
        printf("Philosopher 4 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[4]);
        printf("Philosopher 4 puts down right fork\n");
        pthread_mutex_unlock(&forks[3]);
        printf("Philosopher 4 puts down left fork\n");
    }
    return NULL;
}

void* philo5(void* args)
{
    while (1) {
        printf("Philosopher 5 is thinking\n");
        sleep(1);
        pthread_mutex_lock(&forks[4]);
```

```
        printf("Philosopher 5 takes left fork\n");
        pthread_mutex_lock(&forks[0]);
        printf("Philosopher 5 takes right fork\n");
        printf("Philosopher 5 is eating\n");
        sleep(1);
        pthread_mutex_unlock(&forks[0]);
        printf("Philosopher 5 puts down right fork\n");
        pthread_mutex_unlock(&forks[4]);
        printf("Philosopher 5 puts down left fork\n");
    }
    return NULL;
}

int main(void)
{
    pthread_t ph[5];
    pthread_create(&ph[0], NULL, philo1, NULL);
    pthread_create(&ph[1], NULL, philo2, NULL);
    pthread_create(&ph[2], NULL, philo3, NULL);
    pthread_create(&ph[3], NULL, philo4, NULL);
    pthread_create(&ph[4], NULL, philo5, NULL);

    pthread_join(ph[0], NULL);
    pthread_join(ph[1], NULL);
    pthread_join(ph[2], NULL);
    pthread_join(ph[3], NULL);
    pthread_join(ph[4], NULL);
    return 1;
}
```

Each philosopher needs two forks to eat, a right and a left fork. The functions `philo1`, `philo2`, `philo3`, `philo4`, and `philo5` represent the philosophers. Each function requires two `pthread_mutex_t` resources, representing the two forks required to eat. All five functions run at the same time in five concurrent threads.

However, a deadlock occurs in this example. When each philosopher picks up their first fork (each thread locks one `pthread_mutex_t` resource), all the forks are being used. So, the philosophers (threads) wait for their second fork (second `pthread_mutex_t` resource) to become available. However, all the forks (resources) are being held by the waiting philosophers (threads), causing a deadlock.

Naming Convention for Automatically Detected Threads

If you use a function such as `pthread_create()` to create new threads (tasks), each thread is associated with a unique identifier. For instance, in this example, two threads are created with identifiers `id1` and `id2`.

```
pthread_t* id1, id2;

void main()
{
    pthread_create(id1, NULL, start_routine, NULL);
    pthread_create(id2, NULL, start_routine, NULL);
}
```

If a data race occurs between the threads, the analysis can detect it. When displaying the results, the threads are indicated as `task_id`, where `id` is the identifier associated with the thread. In the preceding example, the threads are identified as `task_id1` and `task_id2`.

If a thread identifier is:

- Local to a function, the thread name shows the function.

For instance, the thread created below appears as `task_f:id`

```
void f(void)
{
    pthread_t* id;
    pthread_create(id, NULL, start_routine, NULL);
}
```

- A field of a structure, the thread name shows the structure.

For instance, the thread created below appears as `task_a#id`

```
struct {pthread_t* id; int x;} a;
pthread_create(a.id, NULL, start_routine, NULL);
```

- An array member, the thread name shows the array.

For instance, the thread created below appears as `task_tab[1]`.

```
pthread_t* tab[10];
pthread_create(tab[1], NULL, start_routine, NULL);
```


Limitations of Automatic Thread Detection

The multitasking model extracted by Polyspace does not include some features. Polyspace cannot model:

- Thread priorities and attributes — Ignored by Polyspace.
- Recursive semaphores.
- Unbounded thread identifiers, such as `extern pthread_t ids[]` — Warning.
- Calls to concurrency primitive through high-order calls — Warning.
- Aliases on thread identifiers — Polyspace over-approximates when the alias is used.
- Termination of threads — Polyspace ignores `pthread_join` and `thrd_join`. Polyspace replaces `pthread_exit` and `thrd_exit` by a standard `exit`.
- (Polyspace Bug Finder only) Creation of multiple threads through multiple calls to the same function with different pointer arguments.

Example

In this example, Polyspace considers that only one thread is created.

```
pthread_t id1, id2;
void start(pthread_t* id)
{
    pthread_create(id, NULL, start_routine, NULL);
}
void main()
{
    start(&id1);
    start(&id2);
}
```

- (Polyspace Code Prover only) Shared local variables — Only global variables are considered shared. If a local variable is accessed by multiple threads, the analysis does not take into account the shared nature of the variable.

Example

In this example, the analysis does not take into account that the local variable `x` can be accessed by both `task1` and `task2` (after the new thread is created).

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

void* task2(void* args)
{
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1()
{
    int x;
    x = 2;
    pthread_t id;
    (void)pthread_create(&id, NULL, task2, (void*) &x);
    /* x (local var) passed to task2 */
    x = 3 ;

    /* Unknown thread priority means x = 1 OR x = 3.*/
    /* However, the analysis considers x = 3 */
    /* Assertion below is green */
    assert(x == 3);
}

int main(void)
{
    task1();
    return 0;
}
```

- (Polyspace Code Prover only) Shared dynamic memory — Only global variables are considered shared. If a dynamically allocated memory region is accessed by multiple threads, the analysis does not take into account its shared nature.

Example

In this example, the analysis does not take into account that `lx` points to a shared memory region. The region can be accessed by both `task1` and `task2` (after the new thread is created). The Code Prover analysis also reports `lx` as a non-shared variable.

```

#include <pthread.h>
#include <stdlib.h>

static int* lx;

void* task2(void* args)
{
    int* x = (int*) args;
    *x = 1;
    return (void*)x;
}

void task1()
{
    pthread_t id;
    lx = (int*)malloc(sizeof(int));

    if (lx == NULL) exit(1);

    (void)pthread_create(&id, NULL, task2, (void*) lx);

    *lx = 3 ;

    /* Unknown thread priority means *lx = 1 OR *lx = 3.*/
    /* However, the analysis considers *lx = 3 */
    /* Assertion below is green */
    assert(*lx == 3);
}

int main(void)
{
    task1();
    return 0;
}

```

- Number of tasks created with CreateThread when threadId is set to NULL— When you create multiple threads that execute the same function, if the last argument of CreateThread is NULL, Polyspace only detects one instance of this function, or task.

Example

In this example, Polyspace detects only one instance of thread_function1(), but 10 instances of thread_function2().

```
#include <windows.h>

#define MAX_LOOP_THREADS 10

DWORD WINAPI thread_function1(LPVOID data) {}
DWORD WINAPI thread_function2(LPVOID data) {}

HANDLE hds1[MAX_LOOP_THREADS];
HANDLE hds2[MAX_LOOP_THREADS];
DWORD threadId[MAX_LOOP_THREADS];

int main(void)
{
    for (int i = 0; i < MAX_LOOP_THREADS; i++) {
        hds1[i] = CreateThread(NULL, 0, thread_function1, NULL, 0, NULL);
        hds2[i] = CreateThread(NULL, 0, thread_function2, NULL, 0, &threadId[i]);
    }

    return 0;
}
```

- (C++11 only) If you use lambda expressions as start functions during thread creation, Polyspace does not detect shared variables in the lambda expressions.

Example

In this example, Polyspace does not detect that the variable `y` used in the lambda expressions is shared between two threads. As a result, Bug Finder, for instance, does not show a **Data race** defect.

```
#include <thread>
int y;
int main() {
    std::thread t1([] {y++;});
    std::thread t2([] {y++;});
    t1.join();
    t2.join();
    return 0;
}
```

- (C++11 threads with Polyspace Code Prover only) String literals as thread function argument — Code Prover shows a red **Illegally dereferenced pointer** error if the

thread function has an `std::string&` parameter and you pass a string literal argument.

Example

In this example, the thread function `foo` has an `std::string&` parameter. When starting a thread, a string literal is passed as argument to this function, which undergoes an implicit conversion to `std::string` type. Code Prover loses track of the original string literal in this conversion. Therefore, a dashed red underline appears on `operator<<` in the body of `foo` and a red **Illegally dereferenced pointer** check in the body of `operator<<`.

```
#include <iostream>
#include <thread>

using namespace std;

void foo(const std::string& f) {
    std::cout << f;
}

void main() {
    std::thread t1(foo, "foo_arg");
}
```

To work around this issue, assign the string literal to a temporary variable and pass the variable as argument to the thread function.

```
#include <iostream>
#include <thread>

using namespace std;

void foo(const std::string& f) {
    std::cout << f;
}

void main() {
    std::string str = "foo_arg";
    std::thread t1(foo, str);
}
```

See Also

-function-behavior-specifications | Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)

More About

- “Analyze Multitasking Programs in Polyspace” on page 7-2
- “Configuring Polyspace Multitasking Analysis Manually” on page 7-19

Configuring Polyspace Multitasking Analysis Manually

With Polyspace, you can analyze programs where multiple threads run concurrently. In some situations, Polyspace can detect thread creation and critical sections in your code automatically. See “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 7-6.

If your code has functions that are intended for concurrent execution, but that cannot be detected automatically, you must specify them before analysis. If these functions operate on a common variable, you must also specify protection mechanisms for those operations.

For the multitasking code analysis workflow, see “Analyze Multitasking Programs in Polyspace” on page 7-2.

Specify Options for Multitasking Analysis

Use these options to specify cyclic tasks, interrupts and protections for shared variables. In the Polyspace user interface, the options are on the **Multitasking** node in the **Configuration** pane.

- **Entry points (-entry-points)**: Specify noncyclic entry point functions.

Do not specify `main`. Polyspace implicitly considers `main` as an entry point function.

- **Cyclic tasks (-cyclic-tasks)**: Specify functions that are scheduled at periodic intervals.
- **Interrupts (-interrupts)**: Specify functions that can run asynchronously.
- **Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)**: Specify functions that disable and reenables interrupts (Bug Finder only).
- **Critical section details (-critical-section-begin -critical-section-end)**: Specify functions that begin and end critical sections.
- **Temporally exclusive tasks (-temporal-exclusions-file)**: Specify groups of functions that are temporally exclusive.
- **-preemptable-interrupts**: Specify functions that have lower priority than interrupts, but higher priority than tasks (preemptable or non-preemptable).

Only the Bug Finder analysis considers priorities.

- **-non-preemptable-tasks:** Specify functions that have higher priority than tasks, but lower priority than interrupts (preemptable or non-preemptable).

Only the Bug Finder analysis considers priorities.

Adapt Code for Code Prover Multitasking Analysis

The multitasking analysis in Code Prover is more exhaustive about finding potentially unprotected shared variables and therefore follows a strict model.

Tasks and interrupts must be void-void functions.

Functions that you specify as tasks and interrupts must have the prototype:

```
void func(void);
```

Suppose you want to specify a function `func` that takes `int` arguments:

```
void func(int);
```

Define a wrapper void-void function that calls `func` with a volatile value. Specify this wrapper function as a task or interrupt.

```
void func_wrapper() {  
    volatile int arg;  
    func(arg);  
}
```

The main function must end.

Code Prover assumes that the `main` function ends before all tasks and interrupts begin. If the `main` function contains an infinite loop or run-time error, the tasks and interrupts are not analyzed. If you see that there are no checks in your tasks and interrupts, look for a token underlined in dashed red to identify the issue in the `main` function. See “Reasons for Unchecked Code” (Polyspace Code Prover).

Suppose you want to specify the `main` function as a cyclic task.

```
void performTask1Cycle(void);  
void performTask2Cycle(void);
```



```
void main() {
    while(1) {
        performTask1Cycle();
    }
}

void task2() {
    while(1) {
        performTask2Cycle();
    }
}
```

Replace the definition of main with:

```
#ifdef POLYSPACE
void main() {
}
void task1() {
    while(1) {
        performTask1Cycle();
    }
}

#else
void main() {
    while(1) {
        performTask1Cycle();
    }
}
#endif
```

The replacement defines an empty main and places the content of main into another function task1 if a macro POLYSPACE is defined. Define the macro POLYSPACE using the option Preprocessor definitions (-D) and specify task1 for the option Tasks (-entry-points).

This assumption does not apply to automatically detected threads. For instance, a main function can create threads using pthread_create.

All tasks and interrupts can interrupt each other.

The Bug Finder analysis considers priorities of tasks. A function that you specify as a task cannot interrupt a function that you specify as an interrupt because an interrupt has higher priority.

The Code Prover analysis considers that all tasks and interrupts can interrupt each other.

All tasks and interrupts can run any number of times in any sequence.

The Code Prover analysis considers that all tasks and interrupts can run any number of times in any sequence.

Suppose in this example, you specify `reset` and `inc` as cyclic tasks. The analysis shows an overflow on the operation `var+=2`.

```
void reset(void) {
    var=0;
}

void inc(void) {
    var+=2;
}
```

Suppose you want to model a scheduling of tasks such that `reset` executes after `inc` has executed five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of `reset` and `inc`.

```
void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        inc();
        inc();
        inc();
        inc();
        inc();
        reset();
    }
}
```

Suppose you want to model a scheduling of tasks such that `reset` executes after `inc` has executed zero to five times. Write a wrapper function that implements this sequence. Specify this new function as a cyclic task instead of `reset` and `inc`.

```
void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        if(randomValue)
            inc();
        if(randomValue)
            inc();
        if(randomValue)
            inc();
        if(randomValue)
            inc();
        if(randomValue)
            inc();
        reset();
    }
}
```

See Also

More About

- “Analyze Multitasking Programs in Polyspace” on page 7-2
- “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 7-6

Protections for Shared Variables in Multitasking Code

If your code is intended for multitasking, tasks in your code can access a common shared variable. To prevent data races, you can protect read and write operations on the variable. This topic shows the various protection mechanisms that Polyspace can recognize.

Detect Unprotected Access

Access	Access Protections	Task	File	Scope	Line
Write	No protection	bug_datarace_task1()	concurrency.c	bug_datarace_task1()	57
Read	No protection	bug_datarace_task2()	concurrency.c	bug_datarace_task2()	62

You can detect an unprotected access using either Bug Finder or Code Prover. Code Prover is more exhaustive and proves if a shared variable is protected from concurrent access.

- Bug Finder detects an unprotected access using the result **Data race**. See Data race.
- Code Prover detects an unprotected access using the result **Shared unprotected global variable**. See Potentially unprotected variable.

Suppose you analyze this code, specifying `signal_handler_1` and `signal_handler_2` as cyclic tasks. Use the analysis option `Cyclic tasks (-cyclic-tasks)`.

```
#include <limits.h>
int shared_var;
```

```
void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void signal_handler_1(void) {
    reset();
    inc();
    inc();
}

void signal_handler_2(void) {
    shared_var = INT_MAX;
}

void main() {
}
```

Bug Finder shows a data race on `shared_var`. Code Prover shows that `shared_var` is a potentially unprotected shared variable. Code Prover also shows that the operation `shared_var += 2` can overflow. The overflow occurs if the call to `inc` in `signal_handler_1` immediately follows the operation `shared_var = INT_MAX` in `signal_handler_2`.

Protect Using Critical Sections

One possible solution is to protect operations on shared variables using critical sections.

In the preceding example, modify your code so that operations on `shared_var` are in the same critical section. Use the functions `take_semaphore` and `give_semaphore` to begin and end the critical sections. To specify these functions that begin and end critical sections, use the analysis options `Critical section details (-critical-section-begin -critical-section-end)`.

```
#include <limits.h>
int shared_var;

void inc() {
```

```
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

/* Declare lock and unlock functions */
void take_semaphore(void);
void give_semaphore(void);

void signal_handler_1() {
    /* Begin critical section */
    take_semaphore();
    reset();
    inc();
    inc();
    /* End critical section */
    give_semaphore();
}

void signal_handler_2() {
    /* Begin critical section */
    take_semaphore();
    shared_var = INT_MAX;
    /* End critical section */
    give_semaphore();
}

void main() {
}
```

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

You can also use primitives such as the POSIX functions `pthread_mutex_lock` and `pthread_mutex_unlock` to begin and end critical sections. For a list of primitives that Polyspace can detect automatically, see “Auto-Detection of Thread Creation and Critical Section in Polyspace” on page 7-6.

Protect Using Temporally Exclusive Tasks

Another possible solution is to specify a group of tasks as temporally exclusive. Temporally exclusive tasks cannot interrupt each other.

In the preceding example, specify that `signal_handler_1` and `signal_handler_2` are temporally exclusive. Use the option Temporally exclusive tasks (`-temporal-exclusions-file`).

You do not see the data race in Bug Finder. Code Prover proves that the shared variable is protected. You also do not see the overflow because the call to `reset()` in `signal_handler_1` always precedes calls to `inc()`.

Protect Using Priorities

Another possible solution is to specify that one task has higher priority over another.

In the preceding example, specify that `signal_handler_1` is an interrupt. Retain `signal_handler_2` as a cyclic task. Use the options Cyclic tasks (`-cyclic-tasks`) and Interrupts (`-interrupts`).

Bug Finder does not show the data race defect anymore. The reason is this:

- The operation `shared_var = INT_MAX` in `signal_handler_2` is atomic. Therefore, the operations in `signal_handler_1` cannot interrupt it.
- The operations in `signal_handler_1` cannot be interrupted by the operation in `signal_handler_2` because `signal_handler_1` has higher priority.

You can specify up to four different priorities with these options (with highest priority listed first):

- Interrupts (`-interrupts`)
- `-preemptable-interrupts`
- `-non-preemptable-tasks`
- Cyclic tasks (`-cyclic-tasks`)

A task with higher priority is atomic with respect to a task with lower priority. Note that the checker Data race including atomic operations ignores the difference in

priorities and continues to show the data race. See also “Define Preemptable Interrupts and Nonpreemptable Tasks” on page 7-33.

Code Prover does not consider priorities of tasks. Therefore, Code Prover still shows `shared_var` as a potentially unprotected global variable.

Protect By Disabling Interrupts

In a Bug Finder analysis, you can protect a group of operations by disabling all interrupts. Use the option `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

After you call a routine to disable interrupts, all subsequent operations are atomic until you call another routine to reenable interrupts. The operations are atomic with respect to operations in all other tasks.

See Also

More About

- “Analyze Multitasking Programs in Polyspace” on page 7-2
- “Define Atomic Operations in Multitasking Code” on page 7-29

Define Atomic Operations in Multitasking Code

In code with multiple threads, you can use Polyspace Bug Finder to detect data races or Polyspace Code Prover to list potentially unprotected shared variables.

To determine if a variable shared between multiple threads is protected against concurrent access, Polyspace checks if the operations on the variable are atomic.

Nonatomic Operations

If an operation is nonatomic, Polyspace considers that the operation involves multiple steps. These steps do not need to occur together and can be interrupted by operations in other threads.

For instance, consider these two operations in two different threads:

- Thread 1: `var++;`

This operation is nonatomic because it takes place in three steps: reading `var`, incrementing `var`, and writing back `var`.

- Thread 2: `var = 0;`

This operation is atomic if the size of `var` is less than the word size on the target. See details below for how Polyspace determines the word size.

If the two operations are not protected (by using, for instance, critical sections), the operation in the second thread can interrupt the operation in the first thread. If the interruption happens after `var` is incremented in the first thread but before the incremented value is written back, you can see unexpected results.

What Polyspace Considers as Nonatomic

Code Prover considers all operations as nonatomic unless you protect them, for instance, by using critical sections. See “Define Specific Operations as Atomic” on page 7-30.

Bug Finder considers an operation as nonatomic if it can translate into more than one machine instruction. For instance:

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

```
long long var1, var2;  
var1=var2;
```

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

Polyspace uses the **Pointer** size for your **Target processor type** as the threshold to compute atomicity. For instance, if you use `i386` as your **Target processor type**, the **Pointer** size is 32 bits and **Long long** and **Double** sizes are both 64 bits. Therefore, Polyspace considers copying one `long long` or `double` variable to another as nonatomic.

See also `Target processor type (-target)`.

- The operation can involve writing the return value of a function call to a shared variable. For example, the operation `x=func()` involves calling `func` and writing the return value of `func` to `x`.

To detect data races where at least one of the two interrupting operations is nonatomic, enable the Bug Finder checker `Data race`. To remove this constraint on the checker, enable `Data race including atomic operations`.

Define Specific Operations as Atomic

You might want to define a group of operations as atomic. This group of operations cannot be interrupted by operations in another thread or task.

Use one of these techniques:

- **Critical sections**

Protect a group of operations with critical sections.

A critical section begins and ends with calls to specific functions. You can use a predefined set of primitives to begin or end critical sections, or use your own functions.

A group of operations in a critical section are atomic with respect to another group of operations that are in the same critical section (that is, having the same beginning and ending function).

Specify critical sections using the option `Critical section details (-critical-section-begin -critical-section-end)`.

- **Temporally exclusive tasks**

Protect a group of operations by specifying certain tasks as temporally exclusive.

If a group of tasks are temporally exclusive, all operations in one task are atomic with respect to operations in the other tasks.

Specify temporal exclusion using the option `Temporally exclusive tasks (-temporal-exclusions-file)`.

- **Task priorities** (Bug Finder only)

Protect a group of operations by specifying that certain tasks have higher priorities. For instance, interrupts have higher priorities over cyclic tasks.

You can specify up to four different priorities with these options (with highest priority listed first):

- `Interrupts (-interrupts)`
- `-preemptable-interrupts`
- `-non-preemptable-tasks`
- `Cyclic tasks (-cyclic-tasks)`

All operations in a task with higher priority are atomic with respect to operations in tasks with lower priorities. See also “Define Preemptable Interrupts and Nonpreemptable Tasks” on page 7-33.

- **Routine disabling interrupts** (Bug Finder only)

Protect a group of operations by disabling all interrupts. Use the option `Disabling all interrupts (-routine-disable-interrupts -routine-enable-interrupts)`.

After you call a routine to disable interrupts, all subsequent operations are atomic until you call another routine to reenable interrupts. The operations are atomic with respect to operations in all other tasks.

For a tutorial, see “Protections for Shared Variables in Multitasking Code” on page 7-24.

See Also

Critical section details (-critical-section-begin -critical-section-end) | Cyclic tasks (-cyclic-tasks) | Interrupts (-interrupts) | Temporally exclusive tasks (-temporal-exclusions-file)

More About

- “Analyze Multitasking Programs in Polyspace” on page 7-2
- “Protections for Shared Variables in Multitasking Code” on page 7-24

Define Preemptable Interrupts and Nonpreemptable Tasks

Bug Finder detects data races between concurrent tasks. Using Bug Finder analysis options, you can fix data race detection by specifying that certain tasks have higher priorities over others. A task with higher priority is atomic with respect to tasks with lower priority and cannot be interrupted by those tasks.

Emulating Task Priorities

You can specify up to four different priorities with these options (with highest priority listed first):

- Interrupts (nonpreemptable): Use option `Interrupts (-interrupts)`.
- Interrupts (preemptable): Use options `Interrupts (-interrupts)` and `-preemptable-interrupts`.
- Cyclic tasks (nonpreemptable): Use options `Cyclic tasks (-cyclic-tasks)` and `-non-preemptable-tasks`.

You can also define preemptable noncyclic tasks with the option `Entry points (-entry-points)` and `-non-preemptable-tasks`.

- Cyclic tasks (preemptable): Use option `Cyclic tasks (-cyclic-tasks)`.

You can also define noncyclic tasks with the option `Entry points (-entry-points)`.

For instance, interrupts have the highest priority and cannot be preempted by other tasks. To define a class of interrupts that can be preempted, lower their priority by making them preemptable.

Examples of Task Priorities

Consider this example with three tasks. A variable `var` is shared between the two tasks `task1` and `task2` without any protection such as a critical section. Depending on the priorities of `task1` and `task2`, Bug Finder shows a data race. The third task is not

relevant for the example (and is added only to include a critical section, otherwise data race detection is disabled).

```
int var;

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

void task1(void) {
    var++;
}

void task2(void) {
    var=0;
}

void task3(void){
    begin_critical_section();
    /* Some atomic operation */
    end_critical_section();
}
```

Adjust the priorities of `task1` and `task2` and see whether a data race is detected. For instance:

- 1 Configure these multitasking options:
 - `Interrupts (-interrupts)`: Specify `task1` and `task2` as interrupts.
 - `Cyclic tasks (-cyclic-tasks)`: Specify `task3` as a cyclic task.
 - `Critical section details (-critical-section-begin -critical-section-end)`: Specify `begin_critical_section` as a function beginning a critical section and `end_critical_section` as a function ending a critical section.
- 2 Run Bug Finder.

You do not see a data race. Since `task1` and `task2` are nonpreemptable interrupts, the shared variable cannot be accessed concurrently.
- 3 Change `task1` to a preemptable interrupt by using the option `-preemptable-interrupts`.
- 4 Run Bug Finder again. You now see a data race on the shared variable `var`.

Further Explorations

Modify this example in the following ways and see the effect of the modification:

- Change the priorities of `task1` and `task2`.

For instance, you can leave `task1` as a nonpreemptable interrupt but change `task2` to a preemptable interrupt by using the option `-preemptable-interrupts`.

The data race disappears. The reason is:

- `task1` has higher priority and cannot be interrupted by `task2`.
- The operation in `task2` is atomic and cannot be interrupted by `task1`.
- Enable the checker `Data race including atomic operations` (not enabled by default). Use the option `Find defects (-checkers)`.

You see the data race again. The checker considers all operations as potentially nonatomic and the operation in `task2` can now be interrupted by the higher priority operation in `task1`.

Try other modifications to the analysis options and see the result of the checkers.

See Also

Polyspace Analysis Options

`-non-preemptable-tasks` | `-preemptable-interrupts` | `Cyclic tasks (-cyclic-tasks)` | `Interrupts (-interrupts)`

Polyspace Results

`Data race` | `Data race including atomic operations`

More About

- “Analyze Multitasking Programs in Polyspace” on page 7-2
- “Protections for Shared Variables in Multitasking Code” on page 7-24
- “Define Atomic Operations in Multitasking Code” on page 7-29

Define Critical Sections with Functions That Take Arguments

When verifying multitasking code, Polyspace considers that a critical section lies between calls to a lock and unlock function.

```
lock();  
/* Critical section code */  
unlock();
```

A group of operations in a critical section are atomic with respect to another group of operations that are in the same critical section (that is, having the same lock and unlock function). See also “Define Atomic Operations in Multitasking Code” on page 7-29.

Polyspace Assumption on Functions Defining Critical Sections

Polyspace ignores arguments to functions that begin and end critical sections.

For instance, Polyspace treats the two code sections below as the same critical section if you specify `my_task_1` and `my_task_2` as entry points, `my_lock` as the lock function and `my_unlock` as the unlock function.

```
int shared_var;  
  
void my_lock(int);  
void my_unlock(int);  
  
void my_task_1() {  
    my_lock(1);  
    /* Critical section code */  
    shared_var=0;  
    my_unlock(1);  
}  
  
void my_task_2() {  
    my_lock(2);  
    /* Critical section code */  
    shared_var++;  
    my_unlock(2);  
}
```


As a result, the analysis considers that these two sections are protected from interrupting each other even though they might not be protected. For instance, Bug Finder does not detect the data race on `shared_var`.

Often, the function arguments can be determined only at run time. Since Polyspace models the critical sections prior to the static analysis and run-time error checking phase, the analysis cannot determine if the function arguments are different and ignores the arguments.

Adapt Polyspace Analysis to Lock and Unlock Functions with Arguments

When the arguments to the functions defining critical sections are compile-time constants, you can adapt the analysis to work around the Polyspace assumption.

For instance, you can use Polyspace analysis options so that the code in the preceding example appears to Polyspace as shown here.

```
int shared_var;

void my_lock_1(void);
void my_lock_2(void);
void my_unlock_1(void);
void my_unlock_2(void);

void my_task_1() {
    my_lock_1();
    /* Critical section code */
    shared_var=0;
    my_unlock_1();
}

void my_task_2() {
    my_lock_2();
    /* Critical section code */
    shared_var++;
    my_unlock_2();
}
```

If you then specify `my_lock_1` and `my_lock_2` as the lock functions and `my_unlock_1` and `my_unlock_2` as the unlock functions, the analysis recognizes the two sections of code as part of different critical sections. For instance, Bug Finder detects a data race on `shared_var`.

To adapt the analysis for lock and unlock functions that take compile-time constants as arguments:

- 1 In a header file `common_polyspace_include.h`, convert the function arguments into extensions of the function name with `#define`-s. In addition, provide a declaration for the new functions.

For instance, for the preceding example, use these `#define`-s and declarations:

```
#define my_lock(X) my_lock_##X()
#define my_unlock(X) my_unlock_##X()

void my_lock_1(void);
void my_lock_2(void);
void my_unlock_1(void);
void my_unlock_2(void);
```

- 2 Specify the file name `common_polyspace_include.h` as argument for the option `Include (-include)`.

The analysis considers this header file as `#include`-d in all source files that are analyzed.

- 3 Specify the new function names as functions beginning and ending critical sections. Use the options `Critical section details (-critical-section-begin -critical-section-end)`.

See Also

`Critical section details (-critical-section-begin -critical-section-end)`

More About

- “Protections for Shared Variables in Multitasking Code” on page 7-24

Configure Coding Rules Checking and Code Metrics Computation

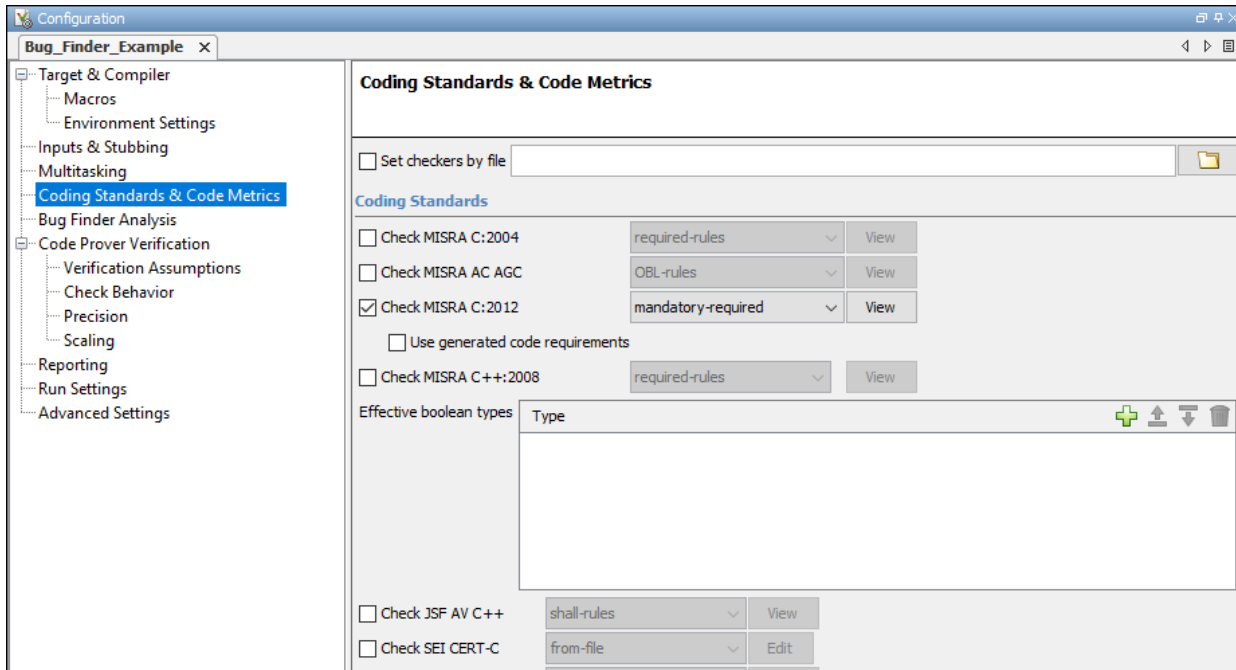
Check for Coding Standard Violations

With Polyspace, you can check your C/C++ code for violations of coding rules such as MISRA C:2012 rules. Adhering to coding rules can reduce the number of defects and improve the quality of your code.

Polyspace can detect the violations of these rules:

- MISRA C: 2004
- MISRA C: 2012
- MISRA C++
- JSF AV C++
- AUTOSAR C++14 (*Bug Finder only*)
- CERT C (*Bug Finder only*)
- CERT C++ (*Bug Finder only*)
- ISO®/IEC TS 17961 (*Bug Finder only*)

Configure Coding Rules Checking



Specify Standard and Predefined Checker Subsets

Specify the coding rules through Polyspace analysis options. When you run Bug Finder or Code Prover, the analysis looks for coding rule violations in addition to other checks. You can disable the other checks and look for coding rule violations only.

In the Polyspace user interface (desktop products), the options are on the **Configuration** pane under the **Coding Standards & Code Metrics** node.

For C code, use one of these options:

- Check MISRA C:2004 (-misra2)

For generated code, enable the option specific to generated code.

- Check MISRA C:2012 (-misra3)

For generated code, enable the option specific to generated code.

- Check SEI CERT-C (-cert-c)
- Check ISO/IEC TS 17961 (-iso-17961)

For C++ code, use one of these options:

- Check MISRA C++ rules (-misra-cpp)
- Check JSF C++ rules (-jsf-coding-rules)
- Check AUTOSAR C++ 14 (-autosar-cpp14)
- Check SEI CERT-C++ (-cert-cpp)

You can specify a predefined subset of rules, for instance, mandatory for MISRA C: 2012. These subsets are typically defined by the standard.

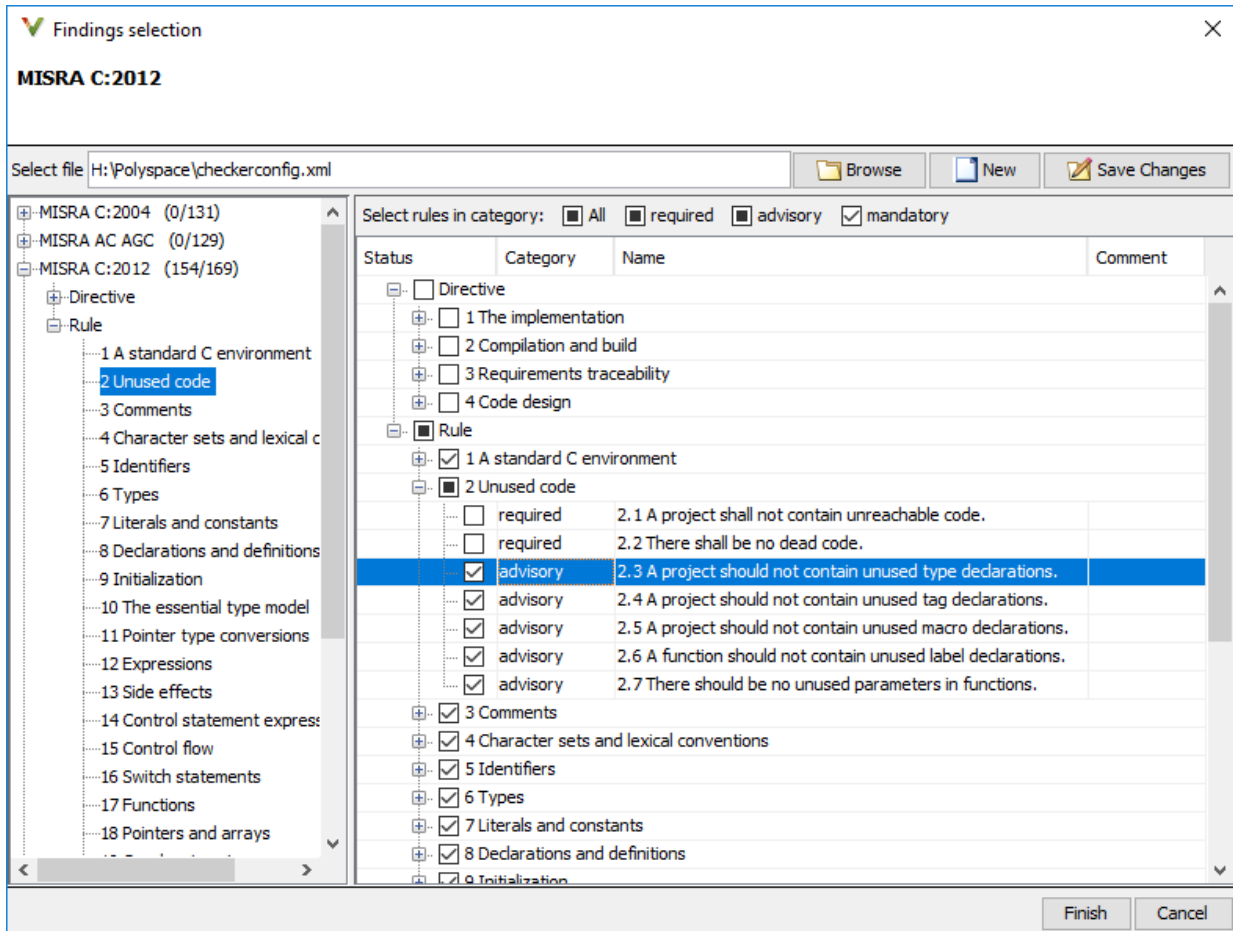
You can also define naming conventions for identifiers using regular expressions. See “Create Custom Coding Rules” on page 8-56.

Customize Checker Subsets

Instead of the predefined subsets, you can specify your own subset of rules from a coding standard.

User Interface (Desktop Products Only)

- 1 Select the coding standard. From the drop-down list for the subset of rules, select from-file. Click **Edit**.
- 2 In the **Findings selection** window, the coding standard is highlighted on the left pane. On the right pane, select the rules that you want to include in your analysis.



When you save the rule selections, the configuration is saved in an XML file that you can reuse for multiple analyses. The same file contains rules selected for all coding standards. You can reuse this file across multiple projects to enforce common coding standards in a team or organization. To reuse this file in another project in the Polyspace user interface:

- Choose a coding standard in the project configuration. From the drop-down list for the subset of rules, select `from-file`.

- Click **Edit** and browse to the file location. Alternatively, enter the file name as argument for the option Set checkers by file (-checkers-selection-file).

Command Line

With the Polyspace desktop products, you can create a coding standard XML file in the user interface and then use this file for command-line analysis.

With the Polyspace server products, you have to create a coding standard XML from scratch. Use the file `StandardsConfiguration.xml` in `polyspaceserverroot\polyspace\examples\cxx\Bug_Finder_Example\sources` as a template and turn on rules using entries in the XML file. Here, `polyspaceserverroot` is the root installation folder for the Polyspace Server products, for instance, `C:\Program Files\Polyspace Server\R2019a`.

For instance, to turn on MISRA C: 2012 rule 8.1, use this entry:

```
<standard name="MISRA C:2012">
  ...
  <section name="8 Declarations and definitions">
    ...
    <check id="8.1" state="on">
      </check>
    ...
  </section>
  ...
</standard>
```

For full list of rule id-s and section names, see:

- "AUTOSAR C++14 Rules" (Polyspace Bug Finder Access)
- "CERT C Rules and Recommendations" (Polyspace Bug Finder Access)
- "CERT C++ Rules" (Polyspace Bug Finder Access)
- "ISO/IEC TS 17961 Rules" (Polyspace Bug Finder Access)
- "Custom Coding Rules" (Polyspace Bug Finder Access)
- "JSF C++ Rules" (Polyspace Bug Finder Access)
- "MISRA C:2004 and MISRA AC AGC Rules" (Polyspace Bug Finder Access)
- "MISRA C:2012 Directives and Rules" (Polyspace Bug Finder Access)

- “MISRA C++:2008 Rules” (Polyspace Bug Finder Access)

Check for Coding Standards Only

To check for coding standards only:

- In Bug Finder, disable checking of defects. Use the option `Find defects (-checkers)`.
- In Code Prover, check for source compliance only. Use the option `Verification level (-to)`.

These rules are checked in the later stages of a Code Prover analysis: MISRA C:2004 rules 9.1, 13.7, and 21.1, and MISRA C:2012 rules 2.2, 9.1, 14.3, and 18.1. If you stop Code Prover at source compliance checking, the analysis might not find all violations of these rules. You can also see a difference in results based on your choice for the option `Verification level (-to)`. For example, it is possible that Code Prover suspects in the first pass that a variable may be uninitialized but proves in the second pass that the variable is initialized. In that case, you see a violation of MISRA C:2012 Rule 9.1 in the first pass but not in the second pass.

Review Coding Rule Violations

The screenshot displays the 'Result Details' pane at the top, which includes a 'Variable trace' checkbox and a 'Result Review' section with 'Status' set to 'To fix' and 'Severity' set to 'Medium'. Below this, a yellow banner indicates a violation of 'MISRA C:2012 5.1 (Required)', stating that external identifiers must be distinct and that the function 'demo_corrected_sighandlerasynsafestRICT' conflicts with the identifier 'demo_corrected_sighandlerasynsafE' (found at line 1171 of programming.c).

A table below the banner lists the violation details:

Event	File	Scope	Line
1	Violation site	programming.c	programming.c 1171
2	MISRA C:2012 5.1	programming.c	File Scope 1230


The bottom pane shows the 'Source' code for 'programming.c'. The function 'demo_corrected_sighandlerasynsafestRICT' is highlighted in blue, corresponding to the violation at line 1230. The code includes several conditional checks for signal errors (SIGTERM, SIGINT) and a call to 'raise(SIGINT)'.

```

1225
1226 void corrected_sighandlerasynsafestRICT(int signum) {
1227     int s0 = signum; /* Fix: avoid raise() */
1228 }
1229
1230 int demo_corrected_sighandlerasynsafestRICT(void) {
1231     if (signal(SIGTERM, demo_term_handler) == SIG_ERR) {
1232         /* Handle error */
1233     }
1234     if (signal(SIGINT, corrected_sighandlerasynsafestRICT) == SIG_ERR) {
1235         /* Handle error */
1236     }
1237     /* Program code */
1238     if (raise(SIGINT) != 0) {
1239         /* Handle error */
1240     }
1241     /* More code */
1242     return 0;
1243 }

```

After analysis, you see the coding standard violations on the **Results List** pane. Select a violation to see further details on the **Result Details** pane and the source code on the **Source** pane.

Violations of coding standards are indicated in the source code with the  icon.

For further steps, see “Review Results in Polyspace Bug Finder Access” (Polyspace Bug Finder Access).

Generate Reports

You can generate reports using templates that are explicitly defined for coding standards. Use the `CodingStandards` template. This template:

- Reports only coding standard violations in your analysis results, and omits other types of results such as defects, run-time errors or code metrics.
- Creates a separate chapter in the report for each coding standard. the chapter provides an overview of all violations of the standard and then lists each violation.

To specify a report template, use the option `Bug Finder and Code Prover report (-report-template)`.

See Also

More About

- “Interpret Polyspace Bug Finder Access Results” (Polyspace Bug Finder Access)

Avoid Violations of MISRA C 2012 Rules 8.x

MISRA C:2012 rules 8.1-8.14 enforce good coding practices surrounding declarations and definitions. If you follow these practices, you are less likely to have conflicting declarations or to unintentionally modify variables.

If you do not follow these practices *during coding*, your code might require major changes later to be MISRA C-compliant. You might have too many MISRA C violations. Sometimes, in fixing a violation, you might violate another rule. Instead, keep these rules in mind when coding. Use the MISRA C:2012 checker to spot any issues that you might have missed.

- **Explicitly specify all data types in declarations.**

Avoid implicit data types like this declaration of k:

```
extern void foo (char c, const k);
```

Instead use:

```
extern void foo (char c, const int k);
```

That way, you do not violate MISRA C:2012 Rule 8.1.

- **When declaring functions, provide names and data types for all parameters.**

Avoid declarations without parameter names like these declarations:

```
extern int func(int);  
extern int func2();
```

Instead use:

```
extern int func(int arg);  
extern int func2(void);
```

That way, you do not violate MISRA C:2012 Rule 8.2.

- **If you want to use an object or function in multiple files, declare the object or function once in only one header file.**

To use an object in multiple source files, declare it as `extern` in a header file. Include the header file in all the source files where you need the object. In one of those source files, define the object. For instance:

```
/* header.h */
extern int var;

/* file1.c */
#include "header.h"
/* Some usage of var */

/* file2.c */
#include "header.h"
int var=1;
```

To use a function in multiple source files, declare it in a header file. Include the header file in all the source files where you need the function. In one of those source files, define the function.

That way, you do not violate MISRA C:2012 Rule 8.3, MISRA C:2012 Rule 8.4, MISRA C:2012 Rule 8.5, or MISRA C:2012 Rule 8.6.

- **If you want to use an object or function in one file only, declare and define the object or function with the static specifier.**

Make sure that you use the `static` specifier in all declarations and the definition. For instance, this function `func` is meant to be used only in the current file:

```
static int func(void);
static int func(void){
}
```

That way, you do not violate MISRA C:2012 Rule 8.7 and MISRA C:2012 Rule 8.8.

- **If you want to use an object in one function only, declare the object in the function body.**

Avoid declaring the object outside the function.

For instance, if you use `var` in `func` only, do declare it outside the body of `func`:

```
int var;
void func(void) {
    var=1;
}
```

Instead use:

```
void func(void) {
    int var;
    var=1;
}
```

That way, you do not violate MISRA C:2012 Rule 8.7 and MISRA C:2012 Rule 8.9.

- **If you want to inline a function, declare and define the function with the static specifier.**

Every time you add `inline` to a function definition, add `static` too:

```
static inline double func(int val);
static inline double func(int val) {
}
```

That way, you do not violate MISRA C:2012 Rule 8.10.

- **When declaring arrays, explicitly specify their size.**

Avoid implicit size specifications like this:

```
extern int32_t array[];
```

Instead use:

```
#define MAXSIZE 10
extern int32_t array[MAXSIZE];
```

That way, you do not violate MISRA C:2012 Rule 8.11.

- **When declaring enumerations, try to avoid mixing implicit and explicit specifications.**

Avoid mixing implicit and explicit specifications. You can specify the first enumeration constant explicitly, but after that, use either implicit or explicit specifications. For instance, avoid this type of mix:

```
enum color {red = 2, blue, green = 3, yellow};
```

Instead use:

```
enum color {red = 2, blue, green, yellow};
```

That way, you do not violate MISRA C:2012 Rule 8.12.

- **When declaring pointers, point to a const-qualified type unless you want to use the pointer to modify an object.**

Point to a const-qualified type by default unless you intend to use the pointer for modifying the pointed object. For instance, in this example, `ptr` is not used to modify the pointed object:

```
char last_char(const char * const ptr){  
}
```

That way, you do not violate MISRA C:2012 Rule 8.13.

Software Quality Objective Subsets (C:2004)

In this section...

“Rules in SQO-Subset1” on page 8-14

“Rules in SQO-Subset2” on page 8-15

Rules in SQO-Subset1

In Polyspace Code Prover, the following set of coding rules will typically reduce the number of unproven results.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control.
14.4	The <i>goto</i> statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.

Rule number	Description
16.2	Functions shall not call themselves, either directly or indirectly.
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.
20.4	Dynamic heap memory allocation shall not be used.

Note Polyspace software does not check MISRA rule **18.3**.

Rules in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The SQO-subset2 option checks the rules in SQO-subset1 and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function

Rule number	Description
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression
10.5	Bitwise operations shall not be performed on signed integer types
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
12.1	Limited dependence should be placed on C's operator precedence rules in expressions
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits
12.5	The operands of a logical && or shall be primary-expressions
12.6	Operands of logical operators (&&, and !) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, or !)
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
12.10	The comma operator shall not be used

Rule number	Description
12.12	The underlying bit representations of floating-point values shall not be used.
13.1	Assignment operators shall not be used in expressions that yield Boolean values
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control.
13.6	Numeric variables being used within a " <i>for</i> " loop for iteration counting should not be modified in the body of the loop
14.4	The <i>goto</i> statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a <i>switch</i> , <i>while</i> , <i>do while</i> or <i>for</i> statement shall be a compound statement
14.10	All <i>if else if</i> constructs should contain a final <i>else</i> clause
15.3	The final clause of a <i>switch</i> statement shall be the <i>default</i> clause
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty

Rule number	Description
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.
20.4	Dynamic heap memory allocation shall not be used.

Note Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \  
return -1 else return 0; }
```

See Also

Check MISRA C:2004 (-misra2)

More About

- “Check for Coding Standard Violations” on page 8-2

Software Quality Objective Subsets (AC AGC)

In this section...
“Rules in SQO-Subset1” on page 8-20
“Rules in SQO-Subset2” on page 8-21

Rules in SQO-Subset1

In Polyspace Code Prover, the following set of coding rules will typically reduce the number of unproven results.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

Rules in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The SQO-subset2 option checks the rules in SQO-subset1 and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits

Rule number	Description
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
12.10	The comma operator shall not be used
12.12	The underlying bit representations of floating-point values shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.

Note Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \  
return -1 else return 0; }
```

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

See Also

Check MISRA AC AGC (-misra-ac-agc)

More About

- “Check for Coding Standard Violations” on page 8-2

Software Quality Objective Subsets (C:2012)

In this section...

“Guidelines in SQO-Subset1” on page 8-24

“Guidelines in SQO-Subset2” on page 8-25

These subsets of MISRA C:2012 guidelines can have a direct or indirect impact on the precision of your Polyspace results. When you set up coding rules checking, you can select these subsets.

Guidelines in SQO-Subset1

The following set of MISRA C:2012 coding guidelines typically reduces the number of unproven results in Polyspace Code Prover.

Rule	Description
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage
8.11	When an array with external linkage is declared, its size should be explicitly specified
8.13	A pointer should point to a const-qualified type whenever possible
11.1	Conversions shall not be performed between a pointer to a function and any other type
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type
11.4	A conversion should not be performed between a pointer to object and an integer type
11.5	A conversion should not be performed from pointer to void into pointer to object
11.6	A cast shall not be performed between pointer to void and an arithmetic type
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type
14.1	A loop counter shall not have essentially floating type

Rule	Description
14.2	A for loop shall be well-formed
15.1	The goto statement should not be used
15.2	The goto statement shall jump to a label declared later in the same function
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement
15.5	A function should have a single point of exit at the end
17.1	The features of <starg.h> shall not be used
17.2	Functions shall not call themselves, either directly or indirectly
18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type
18.5	Declarations should contain no more than two levels of pointer nesting
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist
19.2	The union keyword should not be used
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used

Guidelines in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The SQO-subset2 option checks the rules in SQO-subset1 and some additional rules.

Rule	Description
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage
8.11	When an array with external linkage is declared, its size should be explicitly specified

Rule	Description
8.13	A pointer should point to a const-qualified type whenever possible
11.1	Conversions shall not be performed between a pointer to a function and any other type
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type
11.4	A conversion should not be performed between a pointer to object and an integer type
11.5	A conversion should not be performed from pointer to void into pointer to object
11.6	A cast shall not be performed between pointer to void and an arithmetic type
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer
12.1	The precedence of operators within expressions should be made explicit
12.3	The comma operator should not be used
13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders
13.4	The result of an assignment operator should not be used
14.1	A loop counter shall not have essentially floating type
14.2	A for loop shall be well-formed
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type
15.1	The goto statement should not be used
15.2	The goto statement shall jump to a label declared later in the same function
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement
15.5	A function should have a single point of exit at the end

Rule	Description
15.6	The body of an iteration- statement or a selection- statement shall be a compound- statement
15.7	All if ... else if constructs shall be terminated with an else statement
16.4	Every switch statement shall have a default label
16.5	A default label shall appear as either the first or the last switch label of a switch statement
17.1	The features of <starg.h> shall not be used
17.2	Functions shall not call themselves, either directly or indirectly
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type
18.5	Declarations should contain no more than two levels of pointer nesting
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist
19.2	The union keyword should not be used
20.4	A macro shall not be defined with the same name as a keyword
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses
20.9	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation
20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used

See Also

Check MISRA C:2012 (-misra3)

More About

- “Check for Coding Standard Violations” on page 8-2

Software Quality Objective Subsets (C++)

In this section...

“SQO Subset 1 - Direct Impact on Selectivity” on page 8-29

“SQO Subset 2 - Indirect Impact on Selectivity” on page 8-31

SQO Subset 1 - Direct Impact on Selectivity

The following set of MISRA C++ coding rules will typically improve the number of unproven results in Polyspace Code Prover.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	The One Definition Rule shall not be violated.
3-9-3	The underlying bit representations of floating-point values shall not be used.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.

MISRA C++ Rule	Description
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.

MISRA C++ Rule	Description
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
18-4-1	Dynamic heap memory allocation shall not be used.

SQO Subset 2 - Indirect Impact on Selectivity

Good design practices generally lead to less code complexity, which can improve the number of unproven results in Polyspace Code Prover. The following set of coding rules may help to address design issues in your code. The SQO-subset2 option checks the rules in SQO-subset1 and SQO-subset2.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.
3-9-3	The underlying bit representations of floating-point values shall not be used.

MISRA C++ Rule	Description
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, , !, the equality operators == and !=, the unary & operator, and the conditional operator.
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.
5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.
5-0-13	The condition of an if-statement and the condition of an iteration- statement shall have type bool
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-1	Each operand of a logical && or shall be a postfix - expression.
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.

MISRA C++ Rule	Description
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
5-2-11	The comma operator, && operator and the operator shall not be overloaded.
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3	The unary & operator shall not be overloaded.
5-18-1	The comma operator shall not be used.
6-2-1	Assignment operators shall not be used in sub-expressions.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
6-4-2	All if ... else if constructs shall be terminated with an else clause.
6-4-6	The final clause of a switch statement shall be the default-clause.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

MISRA C++ Rule	Description
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
8-4-3	All exit paths from a function with non- void return type shall have an explicit return statement with an expression.
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.
8-5-2	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
11-0-1	Member data in non- POD class types shall be private.
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.

MISRA C++ Rule	Description
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.
18-4-1	Dynamic heap memory allocation shall not be used.

See Also

Check MISRA C++:2008 (-misra-cpp)

More About

- “Check for Coding Standard Violations” on page 8-2

Coding Rule Subsets Checked Early in Analysis

In the initial compilation phase of the analysis, Polyspace checks those coding rules that do not require the run-time error detection part of the analysis. If you want only those rules checked, you can perform a much quicker analysis.

The software provides two predefined subsets of rules that it checks earlier in the analysis. The subsets are available with the options Check MISRA C:2004 (-misra2), Check MISRA AC AGC (-misra-ac-agc), and Check MISRA C:2012 (-misra3).

Argument	Purpose
single-unit-rules	Check rules that apply only to single translation units. If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the compilation phase.
system-decidable-rules	Check rules in the single-unit-rules subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the linking phase.

See also “Check for Coding Standard Violations” on page 8-2.

MISRA C: 2004 and MISRA AC AGC Rules

The software checks the following rules early in the analysis. The rules that are checked at a system level and appear only in the system-decidable-rules subset are indicated by an asterisk.

Environment

Rule	Description
1.1*	All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

Language Extensions

Rule	Description
2.1	Assembly language shall be encapsulated and isolated.
2.2	Source code shall only use /* */ style comments.
2.3	The character sequence /* shall not be used within a comment.

Documentation

Rule	Description
3.4	All uses of the #pragma directive shall be documented and explained.

Character Sets

Rule	Description
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.
4.2	Trigraphs shall not be used.

Identifiers

Rule	Description
5.1*	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
5.3*	A typedef name shall be a unique identifier.
5.4*	A tag name shall be a unique identifier.
5.5*	No object or function identifier with a static storage duration should be reused.
5.6*	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.
5.7*	No identifier name should be reused.

Types

Rule	Description
6.1	The plain char type shall be used only for the storage and use of character values.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.
6.3	typedefs that indicate size and signedness should be used in place of the basic types.
6.4	Bit fields shall only be defined to be of type unsigned int or signed int.
6.5	Bit fields of type signed int shall be at least 2 bits long.

Constants

Rule	Description
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and Definitions

Rule	Description
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
8.4*	If objects or functions are declared more than once their types shall be compatible.
8.5	There shall be no definitions of objects or functions in a header file.
8.6	Functions shall always be declared at file scope.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8*	An external object or function shall be declared in one file and only one file.
8.9*	An identifier with external linkage shall have exactly one external definition.
8.10*	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
8.11	The <code>static</code> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

Rule	Description
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.
9.3	In an enumerator list, the <code>=</code> construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic Type Conversion

Rule	Description
10.1	<p>The value of an expression of integer type shall not be implicitly converted to a different underlying type if:</p> <ul style="list-style-type: none"> • It is not a conversion to a wider integer type of the same signedness, or • The expression is complex, or • The expression is not constant and is a function argument, or • The expression is not constant and is a return expression
10.2	<p>The value of an expression of floating type shall not be implicitly converted to a different type if</p> <ul style="list-style-type: none"> • It is not a conversion to a wider floating type, or • The expression is complex, or • The expression is a function argument, or • The expression is a return expression
10.3	<p>The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.</p>
10.4	<p>The value of a complex expression of float type may only be cast to narrower floating type.</p>
10.5	<p>If the bitwise operator <code>~</code> and <code><<</code> are applied to an operand of underlying type <code>unsigned char</code> or <code>unsigned short</code>, the result shall be immediately cast to the underlying type of the operand</p>
10.6	<p>The "U" suffix shall be applied to all constants of unsigned types.</p>

Pointer Type Conversion

Rule	Description
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.
11.5	A cast shall not be performed that removes any <code>const</code> or <code>volatile</code> qualification from the type addressed by a pointer

Expressions

Rule	Description
12.1	Limited dependence should be placed on C's operator precedence rules in expressions.
12.3	The <code>sizeof</code> operator should not be used on expressions that contain side effects.
12.5	The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions.
12.6	Operands of logical operators (<code>&&</code> , <code> </code> and <code>!</code>) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (<code>&&</code> , <code> </code> or <code>!</code>).
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed.
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
12.10	The comma operator shall not be used.
12.11	Evaluation of constant unsigned expression should not lead to wraparound.
12.12	The underlying bit representations of floating-point values shall not be used.
12.13	The increment (<code>++</code>) and decrement (<code>--</code>) operators should not be mixed with other operators in an expression

Control Statement Expressions

Rule	Description
13.1	Assignment operators shall not be used in expressions that yield Boolean values.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <code>for</code> statement shall not contain any objects of floating type.
13.5	The three expressions of a <code>for</code> statement shall be concerned only with loop control.
13.6	Numeric variables being used within a <code>for</code> loop for iteration counting should not be modified in the body of the loop.

Control Flow

Rule	Description
14.3	All non-null statements shall either <ul style="list-style-type: none"> • have at least one side effect however executed, or • cause control flow to change.
14.4	The <code>goto</code> statement shall not be used.
14.5	The <code>continue</code> statement shall not be used.
14.6	For any iteration statement, there shall be at most one <code>break</code> statement used for loop termination.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a <code>switch</code> , <code>while</code> , <code>do while</code> or <code>for</code> statement shall be a compound statement.
14.9	An <code>if</code> (expression) construct shall be followed by a compound statement. The <code>else</code> keyword shall be followed by either a compound statement, or another <code>if</code> statement.
14.10	All <code>if else if</code> constructs should contain a final <code>else</code> clause.

Switch Statements

Rule	Description
15.0	Unreachable code is detected between <code>switch</code> statement and first <code>case</code> .
15.1	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement
15.2	An unconditional <code>break</code> statement shall terminate every non-empty <code>switch</code> clause.
15.3	The final clause of a <code>switch</code> statement shall be the <code>default</code> clause.
15.4	A <code>switch</code> expression should not represent a value that is effectively Boolean.
15.5	Every <code>switch</code> statement shall have at least one <code>case</code> clause.

Functions

Rule	Description
16.1	Functions shall not be defined with variable numbers of arguments.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.
16.4*	The identifiers used in the declaration and definition of a function shall be identical.
16.5	Functions with no parameters shall be declared with parameter type <code>void</code> .
16.6	The number of arguments passed to a function shall match the number of parameters.
16.8	All exit paths from a function with non- <code>void</code> return type shall have an explicit return statement with an expression.
16.9	A function identifier shall only be used with either a preceding <code>&</code> , or with a parenthesized parameter list, which may be empty.

Pointers and Arrays

Rule	Description
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	A type should not contain more than 2 levels of pointer indirection.

Structures and Unions

Rule	Description
18.1	All structure or union types shall be complete at the end of a translation unit.
18.4	Unions shall not be used.

Preprocessing Directives

Rule	Description
19.1	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments.
19.2	Nonstandard characters should not occur in header file names in <code>#include</code> directives.
19.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or "filename" sequence.
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
19.5	Macros shall not be <code>#defined</code> and <code>#undefd</code> within a block.
19.6	<code>#undef</code> shall not be used.
19.7	A function should be used in preference to a function like-macro.
19.8	A function-like macro shall not be invoked without all of its arguments.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
19.10	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of <code>#</code> or <code>##</code> .
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in <code>#ifdef</code> and <code>#ifndef</code> preprocessor directives and the <code>defined()</code> operator.
19.12	There shall be at most one occurrence of the <code>#</code> or <code>##</code> preprocessor operators in a single macro definition.
19.13	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
19.14	The <code>defined</code> preprocessor operator shall only be used in one of the two standard forms.
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

Rule	Description
19.17	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related.

Standard Libraries

Rule	Description
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
20.2	The names of standard library macros, objects and functions shall not be reused.
20.4	Dynamic heap memory allocation shall not be used.
20.5	The error indicator <code>errno</code> shall not be used.
20.6	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.
20.7	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.
20.8	The signal handling facilities of <code><signal.h></code> shall not be used.
20.9	The input/output library <code><stdio.h></code> shall not be used in production code.
20.10	The library functions <code>atof</code> , <code>atoi</code> and <code>atoll</code> from library <code><stdlib.h></code> shall not be used.
20.11	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <code><stdlib.h></code> shall not be used.
20.12	The time handling functions of library <code><time.h></code> shall not be used.

The rules that are checked at a system level and appear only in the `system-decidable-rules` subset are indicated by an asterisk.

MISRA C: 2012 Rules

The software checks the following rules early in the analysis. The rules that are checked at a system level and appear only in the `system-decidable-rules` subset are indicated by an asterisk.

Standard C Environment

Rule	Description
1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.
1.2	Language extensions should not be used.

Unused Code

Rule	Description
2.3*	A project should not contain unused type declarations.
2.4*	A project should not contain unused tag declarations.
2.5*	A project should not contain unused macro declarations.
2.6	A function should not contain unused label declarations.
2.7	There should be no unused parameters in functions.

Comments

Rule	Description
3.1	The character sequences <code>/*</code> and <code>//</code> shall not be used within a comment.
3.2	Line-splicing shall not be used in <code>//</code> comments.

Character Sets and Lexical Conventions

Rule	Description
4.1	Octal and hexadecimal escape sequences shall be terminated.
4.2	Trigraphs should not be used.

Identifiers

Rule	Description
5.1*	External identifiers shall be distinct.
5.2	Identifiers declared in the same scope and name space shall be distinct.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
5.4	Macro identifiers shall be distinct.
5.5	Identifiers shall be distinct from macro names.
5.6*	A typedef name shall be a unique identifier.
5.7*	A tag name shall be a unique identifier.
5.8*	Identifiers that define objects or functions with external linkage shall be unique.
5.9*	Identifiers that define objects or functions with internal linkage should be unique.

Types

Rule	Description
6.1	Bit-fields shall only be declared with an appropriate type.
6.2	Single-bit named bit fields shall not be of a signed type.

Literals and Constants

Rule	Description
7.1	Octal constants shall not be used.
7.2	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
7.3	The lowercase character "l" shall not be used in a literal suffix.
7.4	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Declarations and Definitions

Rule	Description
8.1	Types shall be explicitly specified.
8.2	Function types shall be in prototype form with named parameters.
8.3*	All declarations of an object or function shall use the same names and type qualifiers.
8.4	A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5*	An external object or function shall be declared once in one and only one file.
8.6*	An identifier with external linkage shall have exactly one external definition.
8.7*	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
8.8	The <code>static</code> storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.9*	An object should be defined at block scope if its identifier only appears in a single function.
8.10	An inline function shall be declared with the <code>static</code> storage class.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.14	The <code>restrict</code> type qualifier shall not be used.

Initialization

Rule	Description
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.
9.4	An element of an object shall not be initialized more than once.
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The Essential Type Model

Rule	Description
10.1	Operands shall not be of an inappropriate essential type.
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
10.5	The value of an expression should not be cast to an inappropriate essential type.
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer Type Conversion

Rule	Description
11.1	Conversions shall not be performed between a pointer to a function and any other type.
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
11.9	The macro NULL shall be the only permitted form of integer null pointer constant.

Expressions

Rule	Description
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used.
12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side Effects

Rule	Description
13.3	A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.
13.4	The result of an assignment operator should not be used.
13.6	The operand of the sizeof operator shall not contain any expression which has potential side effects.

Control Statement Expressions

Rule	Description
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control Flow

Rule	Description
15.1	The goto statement should not be used.
15.2	The goto statement shall jump to a label declared later in the same function.
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.
15.4	There should be no more than one break or goto statement used to terminate any iteration statement.
15.5	A function should have a single point of exit at the end
15.6	The body of an iteration-statement or a selection-statement shall be a compound statement.
15.7	All if ... else if constructs shall be terminated with an else statement.

Switch Statements

Rule	Description
16.1	All <code>switch</code> statements shall be well-formed.
16.2	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement.
16.3	An unconditional <code>break</code> statement shall terminate every <code>switch</code> -clause.
16.4	Every <code>switch</code> statement shall have a <code>default</code> label.
16.5	A <code>default</code> label shall appear as either the first or the last <code>switch</code> label of a <code>switch</code> statement.
16.6	Every <code>switch</code> statement shall have at least two <code>switch</code> -clauses.
16.7	A <code>switch</code> -expression shall not have essentially Boolean type.

Functions

Rule	Description
17.1	The features of <code><starg.h></code> shall not be used.
17.3	A function shall not be declared implicitly.
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
17.6	The declaration of an array parameter shall not contain the <code>static</code> keyword between the <code>[]</code> .
17.7	The value returned by a function having non-void return type shall be used.

Pointers and Arrays

Rule	Description
18.4	The <code>+</code> , <code>-</code> , <code>+=</code> and <code>-=</code> operators should not be applied to an expression of pointer type.
18.5	Declarations should contain no more than two levels of pointer nesting.
18.7	Flexible array members shall not be declared.
18.8	Variable-length array types shall not be used.

Overlapping Storage

Rule	Description
19.2	The union keyword should not be used.

Preprocessing Directives

Rule	Description
20.1	<code>#include</code> directives should only be preceded by preprocessor directives or comments.
20.2	The <code>'</code> , <code>"</code> , or <code>\</code> characters and the <code>/*</code> or <code>//</code> character sequences shall not occur in a header file name.
20.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or <code>"filename"</code> sequence.
20.4	A macro shall not be defined with the same name as a keyword.
20.5	<code>#undef</code> should not be used.
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument.
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.
20.8	The controlling expression of a <code>#if</code> or <code>#elif</code> preprocessing directive shall evaluate to 0 or 1.
20.9	All identifiers used in the controlling expression of <code>#if</code> or <code>#elif</code> preprocessing directives shall be <code>#define</code> 'd before evaluation.
20.10	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
20.11	A macro parameter immediately following a <code>#</code> operator shall not immediately be followed by a <code>##</code> operator.
20.12	A macro parameter used as an operand to the <code>#</code> or <code>##</code> operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
20.13	A line whose first token is <code>#</code> shall be a valid preprocessing directive.
20.14	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> , <code>#ifdef</code> or <code>#ifndef</code> directive to which they are related.

Standard Libraries

Rule	Description
21.1	<code>#define</code> and <code>#undef</code> shall not be used on a reserved identifier or reserved macro name.
21.2	A reserved identifier or macro name shall not be declared.
21.3	The memory allocation and deallocation functions of <code><stdlib.h></code> shall not be used.
21.4	The standard header file <code><setjmp.h></code> shall not be used.
21.5	The standard header file <code><signal.h></code> shall not be used.
21.6	The Standard Library input/output functions shall not be used.
21.7	The <code>atof</code> , <code>atoi</code> , <code>atol</code> , and <code>atoll</code> functions of <code><stdlib.h></code> shall not be used.
21.8	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> of <code><stdlib.h></code> shall not be used.
21.9	The library functions <code>bsearch</code> and <code>qsort</code> of <code><stdlib.h></code> shall not be used.
21.10	The Standard Library time and date functions shall not be used.
21.11	The standard header file <code><tgmath.h></code> shall not be used.
21.12	The exception handling features of <code><fenv.h></code> should not be used.

The rules that are checked at a system level and appear only in the system-decidable-rules subset are indicated by an asterisk.

See Also

Check MISRA AC AGC (-misra-ac-agc) | Check MISRA C:2004 (-misra2) |
 Check MISRA C:2012 (-misra3)

More About

- “Check for Coding Standard Violations” on page 8-2

Create Custom Coding Rules

This example shows how to create a custom coding rules file. You can use this file to check names or text patterns in your source code against custom rules that you specify. For each rule, you specify a pattern in the form of a regular expression. The software compares the pattern against identifiers in the source code and determines whether the custom rule is violated.

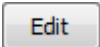
The tutorial uses the following code stored in a file `printInitialValue.c`:

```
#include <stdio.h>

typedef struct {
    int a;
    int b;
} collection;

void main()
{
    collection myCollection= {0,0};
    printf("Initial values in the collection are %d and %d.",
        myCollection.a,myCollection.b);
}
```

User Interface (Desktop Products Only)

- 1 Create a Polyspace project. Add `printInitialValue.c` to the project.
- 2 On the **Configuration** pane, select **Coding Standards & Code Metrics**. Select the **Check custom rules** box.
- 3 Click .

The **Findings selection** window opens, displaying in the left pane all the coding standards Polyspace supports, and with the **Custom** node highlighted.

- 4 Specify the rules to check for in the right pane.

Expand the **4 Structs** node. For the option **4.3 All struct fields must follow the specified pattern**:

Column Title	Action
Status	Select <input checked="" type="checkbox"/> .
Convention	Enter All struct fields must begin with s_ and have capital letters or digits
Pattern	Enter s_[A-Z0-9_]+
Comment	Leave blank. This column is for comments that appear in the coding rules file alone.

- 5 Save the file and run the analysis. On the **Results List** pane, you see two violations of rule 4.3. Select the first violation.
 - a On the **Source** pane, the line `int a;` is marked.
 - b On the **Result Details** pane, you see the error message that you had entered, All struct fields must begin with s_ and have capital letters
- 6 Right-click the **Source** pane and select **Open Editor**. The file `printInitialValue.c` opens in the **Code Editor** pane or an external text editor depending on your **Preferences**.
- 7 In the file, replace all instances of `a` with `s_A` and `b` with `s_B`. Rerun the analysis.

The custom rule violations no longer appear on the **Results List** pane.

Command Line

Create a coding standard configuration file. You can begin from the template file `StandardsConfiguration.xml` provided in `polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources` or `polyspaceroot\polyspace\examples\cxx\Code_Prover_Example\sources`. Here, *polyspaceroot* is the installation folder of the Polyspace desktop or server products. For instance, for the desktop products, the installation folder can be `C:\Program Files\Polyspace\R2019a`.

To enable the custom rule 4.3, enter the following in the configuration file:

```
<standard name="CUSTOM RULES">
  ...
  <section name="4 Structures">
```

```
...
<check id="4.3" state="on">
  <convention>All struct fields must begin with s_
    and have capital letters or digits</convention>
  <pattern>s_[A-Z0-9_]+</pattern>
  <comment># Issue when structure field name does not begin with c_</comment>
</check>
...
</section>
...
</standard>
```

Provide this file as argument for the option `Check custom rules (-custom-rules)` during analysis. For instance, if the text file is named `codingStandard.xml`, for an analysis with Polyspace Code Prover Server, enter:

```
polyspace-code-prover-server -sources file -custom-rules codingStandard.xml
```

See Also

Check custom rules (`-custom-rules`)

Compute Code Complexity Metrics

This example shows how to review the code complexity metrics that Polyspace computes. For information on the individual metrics, see “Code Metrics” (Polyspace Bug Finder Access).

Polyspace does not compute code complexity metrics by default. To compute them during analysis, use the option `Calculate code metrics (-code-metrics)`.

After analysis, the software displays project, file and function metrics on the **Results List** pane. You can compare the computed metric values against predefined limits. If a metric value exceeds limits, you can redesign your code to lower the metric value. For instance, if the number of called functions is high and several of those functions are always called together, you can write one function that fuses the bodies of those functions. Call that one function instead of the group of functions that are called together.

Impose Limits on Metrics (Desktop Products Only)

In the user interface of the Polyspace desktop products, open some results with metrics computations. Then impose limits on the metric values and update results on the **Results List** pane to show only metric values that exceed the limits.

- 1 Select **Tools > Preferences**.
- 2 On the **Review Scope** tab, do one of the following:
 - To use a predefined limit, select **Include Quality Objectives Scopes**.

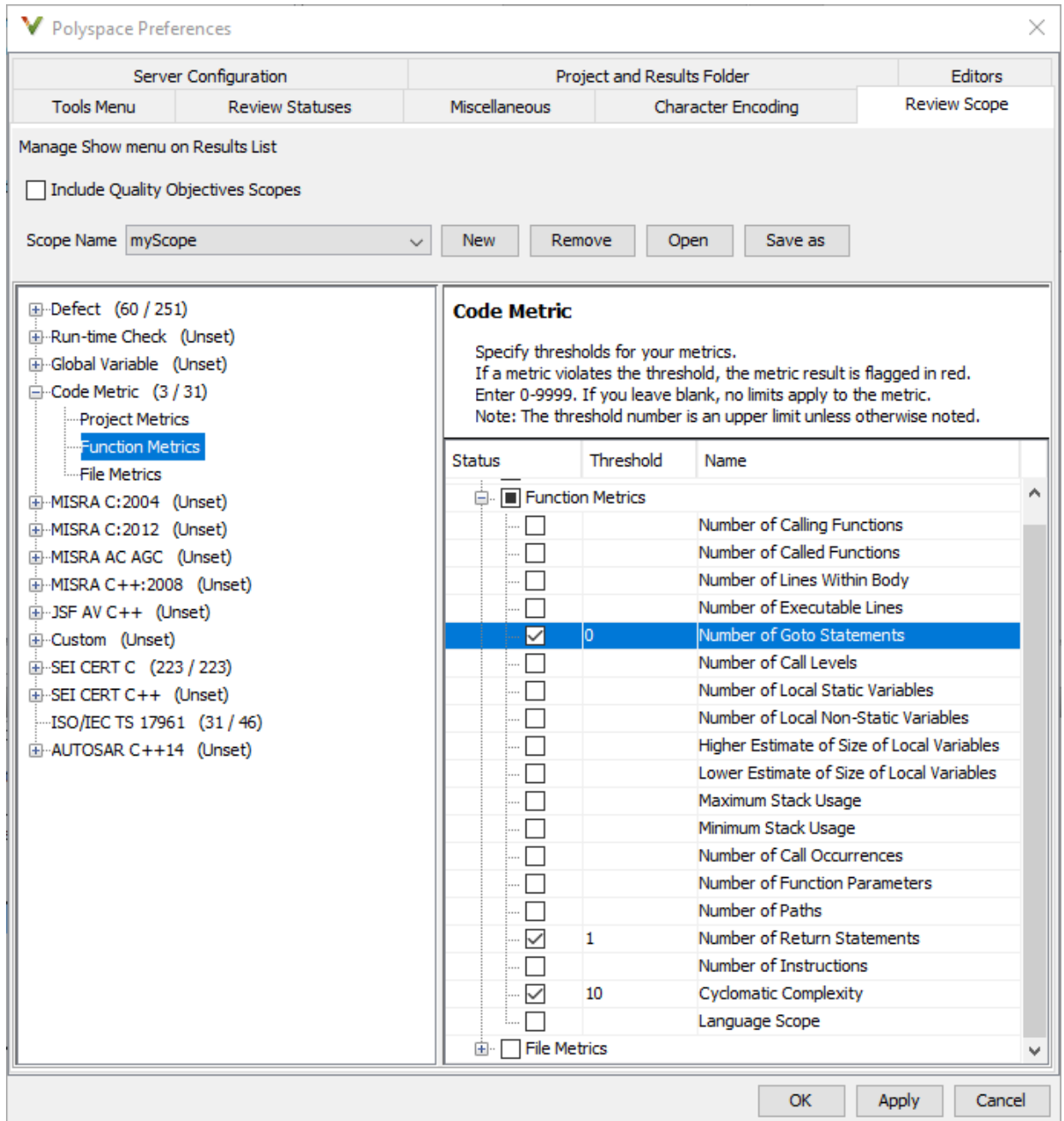
The **Scope Name** list shows the additional option **HIS**. The option **HIS** displays the **HIS** code metrics on page 8-62 only. Select the option to see the limit values.

- To define your own limits, select **New**. Save your limits file.

On the left pane, select **Code Metric**. On the right, select a metric and specify a limit value for the metric. Other than **Comment Density**, limit values are upper limits.

To select all metrics in a category such as **Function Metrics**, select the box next to the category name. For more information on the metrics categories, see “Code Metrics” (Polyspace Bug Finder Access). If only a some metrics in a category are selected, the check box next to the category name displays a symbol.

8 Configure Coding Rules Checking and Code Metrics Computation



3 Select **Apply** or **OK**.

The drop-down list in the left of the **Results List** pane toolbar displays additional options.

- If you use predefined limits, the option **HIS** appears. This option displays code metrics only.
 - If you define your own limits, the option corresponding to your limits file name appears.
- 4** Select the option corresponding to the limits that you want. Only metric values that violate your limits appear on the **Results List** pane.
- 5** Review each violation and decide how to rework your code to avoid the violation.

Note To enforce coding standards across your organization, share your limits file that you saved in XML format.

People in your organization can use the **Open** button on the **Review Scope** tab and navigate to the location of the XML file.

Impose Limits on Metrics (Server and Access products)

In the Polyspace Access web interface, limits on code complexity metrics are predefined. In the **Dashboard** perspective, if you select **Code Metric**, a **Code Metrics** window shows the metric values and limits.

To find the limits used, see “HIS Code Complexity Metrics” on page 8-62.

See Also

Calculate code metrics (-code-metrics)

More About

- “Code Metrics” (Polyspace Bug Finder Access)
- “HIS Code Complexity Metrics” on page 8-62

HIS Code Complexity Metrics

The following list shows the Hersteller Initiative Software (HIS) standard metrics that Polyspace evaluates. These metrics and the recommended limits for their values are part of a standard defined by a major group of Original Equipment Manufacturers or OEMs. For more information on how to focus your review to this subset of code metrics, see “Compute Code Complexity Metrics” on page 8-59.

Project

Polyspace evaluates the following HIS metrics at the project level.

Metric	Recommended Upper Limit
Number of direct recursions	0
Number of recursions	0

File

Polyspace evaluates the HIS metric, comment density, at the file level. The recommended lower limit is 20.

Function

Polyspace evaluates the following HIS metrics at the function level.

Metric	Recommended Upper Limit
Cyclomatic complexity	10
Language scope	4
Number of call levels	4
Number of calling functions	5
Number of called functions	7
Number of function parameters	5
Number of goto statements	0
Number of instructions	50

Metric	Recommended Upper Limit
Number of paths	80
Number of return statements	1

See Also

More About

- “Compute Code Complexity Metrics” on page 8-59
- “Code Metrics” (Polyspace Bug Finder Access)

Configure Bug Finder Checkers

- “Choose Specific Bug Finder Defect Checkers” on page 9-2
- “Short Names of Bug Finder Defect Checkers” on page 9-4
- “Bug Finder Defect Groups” on page 9-21
- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 9-27
- “Bug Finder Results Found in Fast Analysis Mode” on page 9-33
- “CWE Coding Standard and Polyspace Results” on page 9-66
- “Mapping Between CWE-658 or 659 and Polyspace Results” on page 9-100

Choose Specific Bug Finder Defect Checkers

You can check your C/C++ code using the predefined subsets of defect checkers in Bug Finder. However, you can also customize which defects to check for during the analysis.

You can use a spreadsheet to keep track of the defect checkers that you enable and add notes explaining why you do not enable the other checkers. A spreadsheet of checkers is provided in *polyspaceroot*\polyspace\resources. Here, *polyspaceroot* is the Polyspace installation folder, such as C:\Program Files\Polyspace\R2019a.

User Interface (Desktop Products Only)

- 1 On the **Configuration** pane, select **Bug Finder Analysis**.
- 2 From the **Find defects** menu, select a set of defects. The options are:
 - **default** for the default list of defects. This list contains defects that are applicable to most coding projects.

See “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 9-27.

- **all** for all defects.
- **CWE** for defects related to CWE coding standard.

For more information, see “CWE Coding Standard and Polyspace Results” on page 9-66.

- **custom** to add defects to the default list or remove defects from it.

To standardize the bug finding across your organization, you can save your list of defect checkers as a configuration template and share with others. See “Create Project Using Configuration Template” (Polyspace Bug Finder).

Command Line

Use the option `Find defects (-checkers)`. Specify a comma-separated list of checkers as arguments. For instance, to run a Bug Finder analysis on a server with only the data race checkers enabled, enter:

`polyspace-bug-finder-server -sources filename -checkers DATA_RACE,DATA_RACE_STD_LIB`

Use short names for the Bug Finder checkers instead of their full names. See “Short Names of Bug Finder Defect Checkers” on page 9-4.

See Also

Find defects (-checkers)

More About

- “Bug Finder Defect Groups” on page 9-21
- “Short Names of Bug Finder Defect Checkers” on page 9-4

Short Names of Bug Finder Defect Checkers

To justify defects through code annotations, use the command-line names, or short names, listed in the following table.

You can also enable the detection of a specific defect by using its short name as argument of the `-checkers` option. Instead of listing individual defects, you can also specify groups of defects by the group name, for instance, `numerical`, `data_flow`, and so on. See `Find defects (-checkers)`.

Defect	Command-line Name
*this not returned in copy assignment operator	RETURN_NOT_REF_TO_THIS
Abnormal termination of exit handler	EXIT_ABNORMAL_HANDLER
Absorption of float operand	FLOAT_ABSORPTION
Accessing object with temporary lifetime	TEMP_OBJECT_ACCESS
Alignment changed after memory reallocation	ALIGNMENT_CHANGE
Alternating input and output from a stream without flush or positioning call	IO_INTERLEAVING
Ambiguous declaration syntax	MOST_VEXING_PARSE
Arithmetic operation with NULL pointer	NULL_PTR_ARITH
Array access out of bounds	OUT_BOUND_ARRAY
Array access with tainted index	TAINTED_ARRAY_INDEX
Assertion	ASSERT

Defect	Command-line Name
Atomic load and store sequence not atomic	ATOMIC_VAR_SEQUENCE_NOT_ATOMIC
Atomic variable accessed twice in an expression	ATOMIC_VAR_ACCESS_TWICE
Bad file access mode or status	BAD_FILE_ACCESS_MODE_STATUS
Bad order of dropping privileges	BAD_PRIVILEGE_DROP_ORDER
Base class assignment operator not called	MISSING_BASE_ASSIGN_OP_CALL
Base class destructor not virtual	DTOR_NOT_VIRTUAL
Bitwise and arithmetic operation on the same data	BITWISE_ARITH_MIX
Bitwise operation on negative value	BITWISE_NEG
Blocking operation while holding lock	BLOCKING_WHILE_LOCKED
Buffer overflow from incorrect string format specifier	STR_FORMAT_BUFFER_OVERFLOW
C++ reference to const-qualified type with subsequent modification	WRITE_REFERENCE_TO_CONST_TYPE
C++ reference type qualified with const or volatile	CV_QUALIFIED_REFERENCE_TYPE
Call through non-prototyped function pointer	UNPROTOTYPED_FUNC_CALL
Call to memset with unintended value	MEMSET_INVALID_VALUE

Defect	Command-line Name
Character value absorbed into EOF	CHAR_EOF_CONFUSED
Closing a previously closed resource	DOUBLE_RESOURCE_CLOSE
Code deactivated by constant false condition	DEACTIVATED_CODE
Command executed from externally controlled path	TAINTED_PATH_CMD
Constant block cipher initialization vector	CRYPTO_CIPHER_CONSTANT_IV
Constant cipher key	CRYPTO_CIPHER_CONSTANT_KEY
Context initialized incorrectly for cryptographic operation	CRYPTO_PKEY_INCORRECT_INIT
Context initialized incorrectly for digest operation	CRYPTO_MD_BAD_FUNCTION
Conversion or deletion of incomplete class pointer	INCOMPLETE_CLASS_PTR
Copy constructor not called in initialization list	MISSING_COPY_CTOR_CALL
Copy of overlapping memory	OVERLAPPING_COPY
Copy operation modifying source operand	COPY_MODIFYING_SOURCE
Data race	DATA_RACE
Data race including atomic operations	DATA_RACE_ALL

Defect	Command-line Name
Data race through standard library function call	DATA_RACE_STD_LIB
Dead code	DEAD_CODE
Deadlock	DEADLOCK
Deallocation of previously deallocated pointer	DOUBLE_DEALLOCATION
Declaration mismatch	DECL_MISMATCH
Delete of void pointer	DELETE_OF_VOID_PTR
Destination buffer overflow in string manipulation	STRLIB_BUFFER_OVERFLOW
Destination buffer underflow in string manipulation	STRLIB_BUFFER_UNDERFLOW
Destruction of locked mutex	DESTROY_LOCKED
Deterministic random output from constant seed	RAND_SEED_CONSTANT
Double lock	DOUBLE_LOCK
Double unlock	DOUBLE_UNLOCK
Environment pointer invalidated by previous operation	INVALID_ENV_POINTER
Errno not checked	ERRNO_NOT_CHECKED
Errno not reset	MISSING_ERRNO_RESET
Exception caught by value	EXCP_CAUGHT_BY_VALUE
Exception handler hidden by previous handler	EXCP_HANDLER_HIDDEN

Defect	Command-line Name
Execution of a binary from a relative path can be controlled by an external actor	RELATIVE_PATH_CMD
Execution of externally controlled command	TAINTED_EXTERNAL_CMD
File access between time of check and use (TOCTOU)	TOCTOU
File descriptor exposure to child process	FILE_EXPOSURE_TO_CHILD
File manipulation after chroot without chdir	CHROOT_MISUSE
Float conversion overflow	FLOAT_CONV_OVFL
Float division by zero	FLOAT_ZERO_DIV
Floating point comparison with equality operators	BAD_FLOAT_OP
Float overflow	FLOAT_OVFL
Format string specifiers and arguments mismatch	STRING_FORMAT
Function called from signal handler not asynchronous-safe	SIG_HANDLER_ASYNC_UNSAFE
Function called from signal handler not asynchronous-safe (strict)	SIG_HANDLER_ASYNC_UNSAFE_STRICT
Function pointer assigned with absolute address	FUNC_PTR_ABSOLUTE_ADDR

Defect	Command-line Name
Function that can spuriously fail not wrapped in loop	SPURIOUS_FAILURE_NOT_WRAPPED_IN_LOOP
Function that can spuriously wake up not wrapped in loop	SPURIOUS_WAKEUP_NOT_WRAPPED_IN_LOOP
Hard-coded buffer size	HARD_CODED_BUFFER_SIZE
Hard-coded loop boundary	HARD_CODED_LOOP_BOUNDARY
Hard-coded object size used to manipulate memory	HARD_CODED_MEM_SIZE
Host change using externally controlled elements	TAINTED_HOSTID
Improper array initialization	IMPROPER_ARRAY_INIT
Inappropriate I/O operation on device files	INAPPROPRIATE_IO_ON_DEVICE
Incompatible padding for RSA algorithm operation	CRYPTO_RSA_BAD_PADDING
Incompatible types prevent overriding	VIRTUAL_FUNC_HIDING
Inconsistent cipher operations	CRYPTO_CIPHER_BAD_FUNCTION
Incorrect data type passed to va_arg	VA_ARG_INCORRECT_TYPE
Incorrect key for cryptographic algorithm	CRYPTO_PKEY_INCORRECT_KEY
Incorrect order of network connection operations	BAD_NETWORK_CONNECT_ORDER

Defect	Command-line Name
Incorrect pointer scaling	BAD_PTR_SCALING
Incorrect type data passed to va_start	VA_START_INCORRECT_TYPE
Incorrect use of offsetof in C++	OFFSETOF_MISUSE
Incorrect use of va_start	VA_START_MISUSE
Incorrect syntax of flexible array member size	FLEXIBLE_ARRAY_MEMBER_INCORRECT_SIZE
Information leak via structure padding	PADDING_INFO_LEAK
Inline constraint not respected	INLINE_CONSTRAINT_NOT_RESPECTED
Integer constant overflow	INT_CONSTANT_OVFL
Integer conversion overflow	INT_CONV_OVFL
Integer division by zero	INT_ZERO_DIV
Integer overflow	INT_OVFL
Integer precision exceeded	INT_PRECISION_EXCEEDED
Invalid assumptions about memory organization	INVALID_MEMORY_ASSUMPTION
Invalid deletion of pointer	BAD_DELETE
Invalid file position	INVALID_FILE_POS
Invalid free of pointer	BAD_FREE
Invalid use of = (assignment) operator	BAD_EQUAL_USE

Defect	Command-line Name
Invalid use of == (equality) operator	BAD_EQUAL_EQUAL_USE
Invalid use of standard library floating point routine	FLOAT_STD_LIB
Invalid use of standard library integer routine	INT_STD_LIB
Invalid use of standard library memory routine	MEM_STD_LIB
Invalid use of standard library routine	OTHER_STD_LIB
Invalid use of standard library string routine	STR_STD_LIB
Invalid va_list argument	INVALID_VA_LIST_ARG
Large pass-by-value argument	PASS_BY_VALUE
Library loaded from externally controlled path	TAINTED_PATH_LIB
Line with more than one statement	MORE_THAN_ONE_STATEMENT
Load of library from a relative path can be controlled by an external actor	RELATIVE_PATH_LIB
Loop bounded with tainted value	TAINTED_LOOP_BOUNDARY
Member not initialized in constructor	NON_INIT_MEMBER
Memory allocation with tainted size	TAINTED_MEMORY_ALLOC_SIZE
Memory comparison of float-point values	MEMCMP_FLOAT

Defect	Command-line Name
Memory comparison of padding data	MEMCMP_PADDING_DATA
Memory comparison of strings	MEMCMP_STRINGS
Memory leak	MEM_LEAK
Mismatch between data length and size	DATA_LENGTH_MISMATCH
Mismatched alloc/dealloc functions on Windows	WIN_MISMATCH_DEALLOC
Missing blinding for RSA algorithm	CRYPTO_RSA_NO_BLINDING
Missing block cipher initialization vector	CRYPTO_CIPHER_NO_IV
Missing break of switch case	MISSING_SWITCH_BREAK
Missing byte reordering when transferring data	MISSING_BYTESWAP
Missing case for switch condition	MISSING_SWITCH_CASE
Missing cipher algorithm	CRYPTO_CIPHER_NO_ALGORITHM
Missing cipher data to process	CRYPTO_CIPHER_NO_DATA
Missing cipher final step	CRYPTO_CIPHER_NO_FINAL
Missing cipher key	CRYPTO_CIPHER_NO_KEY
Missing data for encryption, decryption or signing operation	CRYPTO_PKEY_NO_DATA
Missing explicit keyword	MISSING_EXPLICIT_KEYWORD
Missing lock	BAD_UNLOCK
Missing null in string array	MISSING_NULL_CHAR

Defect	Command-line Name
Missing overload of allocation or deallocation function	MISSING_OVERLOAD_NEW_DELETE_PAIR
Missing padding for RSA algorithm	CRYPTO_RSA_NO_PADDING
Missing parameters for key generation	CRYPTO_PKEY_NO_PARAMS
Missing peer key	CRYPTO_PKEY_NO_PEER
Missing private key	CRYPTO_PKEY_NO_PRIVATE_KEY
Missing public key	CRYPTO_PKEY_NO_PUBLIC_KEY
Missing reset of a freed pointer	MISSING_FREED_PTR_RESET
Missing return statement	MISSING_RETURN
Missing unlock	BAD_LOCK
Missing virtual inheritance	MISSING_VIRTUAL_INHERITANCE
Misuse of a FILE object	FILE_OBJECT_MISUSE
Misuse of errno	ERRNO_MISUSE
Misuse of errno in a signal handler	SIG_HANDLER_ERRNO_MISUSE
Misuse of narrow or wide character string	NARROW_WIDE_STR_MISUSE
Misuse of readlink()	READLINK_MISUSE
Misuse of return value from nonreentrant standard function	NON_REENTRANT_STD_RETURN
Misuse of sign-extended character value	CHARACTER_MISUSE
Misuse of structure with flexible array member	FLEXIBLE_ARRAY_MEMBER_STRUCT_MISUSE

Defect	Command-line Name
Modification of internal buffer returned from nonreentrant standard function	WRITE_INTERNAL_BUFFER_RETURNED_FROM_STD_FUNC
Non-initialized pointer	NON_INIT_PTR
Non-initialized variable	NON_INIT_VAR
Nonsecure hash algorithm	CRYPTO_MD_WEAK_HASH
Nonsecure parameters for key generation	CRYPTO_PKEY_WEAK_PARAMS
Nonsecure RSA public exponent	CRYPTO_RSA_LOW_EXPONENT
Nonsecure SSL/TLS protocol	CRYPTO_SSL_WEAK_PROTOCOL
Null pointer	NULL_PTR
Object slicing	OBJECT_SLICING
Opening previously opened resource	DOUBLE_RESOURCE_OPEN
Overlapping assignment	OVERLAPPING_ASSIGN
Partially accessed array	PARTIALLY_ACCESSED_ARRAY
Partial override of overloaded virtual functions	PARTIAL_OVERRIDE
Pointer access out of bounds	OUT_BOUND_PTR
Pointer dereference with tainted offset	TAINED_PTR_OFFSET
Pointer or reference to stack variable leaving scope	LOCAL_ADDR_ESCAPE

Defect	Command-line Name
Pointer to non-initialized value converted to const pointer	NON_INIT_PTR_CONV
Possible invalid operation on boolean operand	INVALID_OPERATION_ON_BOOLEAN
Possible misuse of sizeof	SIZEOF_MISUSE
Possibly unintended evaluation of expression because of operator precedence rules	OPERATOR_PRECEDENCE
Precision loss in integer to float conversion	INT_TO_FLOAT_PRECISION_LOSS
Predefined macro used as an object	MACRO_USED_AS_OBJECT
Predictable block cipher initialization vector	CRYPTO_CIPHER_PREDICTABLE_IV
Predictable cipher key	CRYPTO_CIPHER_PREDICTABLE_KEY
Predictable random output from predictable seed	RAND_SEED_PREDICTABLE
Preprocessor directive in macro argument	PRE_DIRECTIVE_MACRO_ARG
Privilege drop not verified	MISSING_PRIVILEGE_DROP_CHECK
Qualifier removed in conversion	QUALIFIER_MISMATCH
Resource leak	RESOURCE_LEAK

Defect	Command-line Name
Returned value of a sensitive function not checked	RETURN_NOT_CHECKED
Return from computational exception signal handler	SIG_HANDLER_COMP_EXCP_RETURN
Return of non const handle to encapsulated data member	BREAKING_DATA_ENCAPSULATION
Self assignment not tested in operator	MISSING_SELF_ASSIGN_TEST
Sensitive data printed out	SENSITIVE_DATA_PRINT
Sensitive heap memory not cleared before release	SENSITIVE_HEAP_NOT_CLEARED
Shared data access within signal handler	SIG_HANDLER_SHARED_OBJECT
Shift of a negative value	SHIFT_NEG
Shift operation overflow	SHIFT_OVFL
Side effect in arguments to unsafe macro	SIDE_EFFECT_IN_UNSAFE_MACRO_ARG
Side effect of expression ignored	SIDE_EFFECT_IGNORED
Signal call from within signal handler	SIG_HANDLER_CALLING_SIGNAL
Signal call in multithreaded program	SIGNAL_USE_IN_MULTITHREADED_PROGRAM
Sign change integer conversion overflow	SIGN_CHANGE
Standard function call with incorrect arguments	STD_FUNC_ARG_MISMATCH

Defect	Command-line Name
Static uncalled function	UNCALLED_FUNC
Stream argument with possibly unintended side effects	STREAM_WITH_SIDE_EFFECT
Subtraction or comparison between pointers to different arrays	PTR_TO_DIFF_ARRAY
Tainted division operand	TAINTED_INT_DIVISION
Tainted modulo operand	TAINTED_INT_MOD
Tainted NULL or non-null-terminated string	TAINTED_STRING
Tainted sign change conversion	TAINTED_SIGN_CHANGE
Tainted size of variable length array	TAINTED_VLA_SIZE
Tainted string format	TAINTED_STRING_FORMAT
Thread-specific memory leak	THREAD_MEM_LEAK
Too many va_arg calls for current argument list	TOO_MANY_VA_ARG_CALLS
Typedef mismatch	TYPEDEF_MISMATCH
Umask used with chmod-style arguments	BAD_UMASK
Uncleared sensitive data in stack	SENSITIVE_STACK_NOT_CLEARED
Universal character name from token concatenation	PRE_UCNAME_JOIN_TOKENS
Unprotected dynamic memory allocation	UNPROTECTED_MEMORY_ALLOCATION
Unreachable code	UNREACHABLE

Defect	Command-line Name
Unreliable cast of function pointer	FUNC_CAST
Unreliable cast of pointer	PTR_CAST
Unsafe call to a system function	UNSAFE_SYSTEM_CALL
Unsafe conversion between pointer and integer	BAD_INT_PTR_CAST
Unsafe conversion from string to numerical value	UNSAFE_STR_TO_NUMERIC
Unsafe standard encryption function	UNSAFE_STD_CRYPT
Unsafe standard function	UNSAFE_STD_FUNC
Unsigned integer constant overflow	UINT_CONSTANT_OVFL
Unsigned integer conversion overflow	UINT_CONV_OVFL
Unsigned integer overflow	UINT_OVFL
Unused parameter	UNUSED_PARAMETER
Useless if	USELESS_IF
Use of automatic variable as putenv-family function argument	PUTENV_AUTO_VAR
Use of dangerous standard function	DANGEROUS_STD_FUNC
Use of externally controlled environment variable	TAINTED_ENV_VARIABLE

Defect	Command-line Name
Use of indeterminate string	INDETERMINATE_STRING
Use of memset with size argument zero	MEMSET_INVALID_SIZE
Use of non-secure temporary file	NON_SECURE_TEMP_FILE
Use of obsolete standard function	OBSOLETE_STD_FUNC
Use of path manipulation function without maximum sized buffer checking	PATH_BUFFER_OVERFLOW
Use of plain char type for numerical value	BAD_PLAIN_CHAR_USE
Use of previously closed resource	CLOSED_RESOURCE_USE
Use of previously freed pointer	FREED_PTR
Use of setjmp/longjmp	SETJMP_LONGJMP_USE
Use of signal to kill thread	THREAD_KILLED_WITH_SIGNAL
Use of tainted pointer	TAINTED_PTR
Variable length array with nonpositive size	NON_POSITIVE_VLA_SIZE
Variable shadowing	VAR_SHADOWING
Vulnerable path manipulation	PATH_TRAVERSAL
Vulnerable permission assignments	DANGEROUS_PERMISSIONS
Vulnerable pseudo-random number generator	VULNERABLE_PRNG
Weak cipher algorithm	CRYPTO_CIPHER_WEAK_CIPHER
Weak cipher mode	CRYPTO_CIPHER_WEAK_MODE

Defect	Command-line Name
Weak padding for RSA algorithm	CRYPTO_RSA_WEAK_PADDING
Write without a further read	USELESS_WRITE
Writing to const qualified object	CONSTANT_OBJECT_WRITE
Writing to read-only resource	READ_ONLY_RESOURCE_WRITE
Wrong allocated object size for cast	OBJECT_SIZE_MISMATCH
Wrong type used in sizeof	PTR_SIZEOF_MISMATCH

See Also

More About

- “Choose Specific Bug Finder Defect Checkers” on page 9-2

Bug Finder Defect Groups

In this section...

“Concurrency” on page 9-21
“Cryptography” on page 9-22
“Data flow” on page 9-22
“Dynamic Memory” on page 9-23
“Good Practice” on page 9-23
“Numerical” on page 9-23
“Object Oriented” on page 9-24
“Programming” on page 9-24
“Resource Management” on page 9-24
“Static Memory” on page 9-25
“Security” on page 9-25
“Tainted data” on page 9-25

For convenience, the defect checkers in Bug Finder are classified into various groups.

- In certain projects, you can choose to focus only on specific groups of defects. Specify the group name for the option `Find defects (-checkers)`.
- When reviewing results, you can review all results of a certain group together. Filter out other results during review. See “Manage Results” (Polyspace Bug Finder Access).

This topic gives an overview of the various groups.

Concurrency

These defects are related to multitasking code.

Data Race Defects

The data race defects occur when multiple tasks operate on a shared variable or call a nonreentrant standard library function without protection.

For the specific defects, see “Concurrency Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: concurrency

Locking Defects

The locking defects occur when the critical sections are not set up appropriately. For example:

- The critical sections are involved in a deadlock.
- A lock function does not have the corresponding unlock function.
- A lock function is called twice without an intermediate call to an unlock function.

Critical sections protect shared variables from concurrent access. Polyspace expects critical sections to follow a certain format. The critical section must lie between a call to a lock function and a call to an unlock function.

For the specific defects, see “Concurrency Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: concurrency

Cryptography

These defects are related to incorrect use of cryptography routines from the OpenSSL library. For instance:

- Use of cryptographically weak algorithms
- Absence of essential elements such as cipher key or initialization vector
- Wrong order of cryptographic operations

For the specific defects, see “Cryptography Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: cryptography

Data flow

These defects are errors relating to how information moves throughout your code. The defects include:

- Dead or unreachable code
- Unused code

- Non-initialized information

For the specific defects, see “Data Flow Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: `data_flow`

Dynamic Memory

These defects are errors relating to memory usage when the memory is dynamically allocated. The defects include:

- Freeing dynamically allocated memory
- Unprotected memory allocations

For specific defects, see “Dynamic Memory Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: `dynamic_memory`

Good Practice

These defects allow you to observe good coding practices. The defects by themselves might not cause a crash, but they sometimes highlight more serious logic errors in your code. The defects also make your code vulnerable to attacks and hard to maintain.

The defects include:

- Hard-coded constants such as buffer size and loop boundary
- Unused function parameters

For specific defects, see “Good Practice Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: `good_practice`

Numerical

These defects are errors relating to variables in your code; their values, data types, and usage. The defects include:

- Mathematical operations

- Conversion overflow
- Operational overflow

For specific defects, see “Numerical Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: `numerical`

Object Oriented

These defects are related to the object-oriented aspect of C++ programming. The defects highlight class design issues or issues in the inheritance hierarchy.

The defects include:

- Data member not initialized or incorrectly initialized in constructor
- Incorrect overriding of base class methods
- Breaking of data encapsulation

For specific defects, see “Object Oriented Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: `object_oriented`

Programming

These defects are errors relating to programming syntax. These defects include:

- Assignment versus equality operators
- Mismatches between variable qualifiers or declarations
- Badly formatted strings

For specific defects, see “Programming Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: `programming`

Resource Management

These defects are related to file handling. The defects include:

- Unclosed file stream

- Operations on a file stream after it is closed

For specific defects, see “Resource Management Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: `resource_management`

Static Memory

These defects are errors relating to memory usage when the memory is statically allocated. The defects include:

- Accessing arrays outside their bounds
- Null pointers
- Casting of pointers

For specific defects, see “Static Memory Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: `static_memory`

Security

These defects highlight places in your code which are vulnerable to hacking or other security attacks. Many of these defects do not cause runtime errors, but instead point out risky areas in your code. The defects include:

- Managing sensitive data
- Using dangerous or obsolete functions
- Generating random numbers
- Externally controlled paths and commands

For more details about specific defects, see “Security Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: `security`

Tainted data

These defects highlight elements in your code which are from unsecured sources. Malicious attackers can use input data or paths to attack your program and cause

failures. These defects highlight elements in your code that are vulnerable. Defects include:

- Use of tainted variables or pointers
- Externally controlled paths

For more details about specific defects, see “Tainted Data Defects” (Polyspace Bug Finder Access).

Command-Line Parameter: `tainted_data`

See Also

Find defects (-checkers)

Polyspace Bug Finder Defects Checkers Enabled by Default

When you launch a Bug Finder analysis, the default setting for Find defects (-checkers) enables these defect checkers.

Defect	Command-line Name
Absorption of float operand	FLOAT_ABSORPTION
Accessing object with temporary lifetime	TEMP_OBJECT_ACCESS
Alignment changed after memory reallocation	ALIGNMENT_CHANGE
Alternating input and output from a stream without flush or positioning call	IO_INTERLEAVING
Array access out of bounds	OUT_BOUND_ARRAY
Assertion	ASSERT
Atomic load and store sequence not atomic	ATOMIC_VAR_SEQUENCE_NOT_ATOMIC
Atomic variable accessed twice in an expression	ATOMIC_VAR_ACCESS_TWICE
Base class assignment operator not called	MISSING_BASE_ASSIGN_OP_CALL
Base class destructor not virtual	DTOR_NOT_VIRTUAL
Buffer overflow from incorrect string format specifier	STR_FORMAT_BUFFER_OVERFLOW
Call through non-prototyped function pointer	UNPROTOTYPED_FUNC_CALL
Character value absorbed into EOF	CHAR_EOF_CONFUSED
Closing a previously closed resource	DOUBLE_RESOURCE_CLOSE
Conversion or deletion of incomplete class pointer	INCOMPLETE_CLASS_PTR

Defect	Command-line Name
Copy constructor not called in initialization list	MISSING_COPY_CTOR_CALL
Copy operation modifying source operand	COPY_MODIFYING_SOURCE
Data race	DATA_RACE
Data race through standard library function call	DATA_RACE_STD_LIB
Dead code	DEAD_CODE
Deadlock	DEADLOCK
Deallocation of previously deallocated pointer	DOUBLE_DEALLOCATION
Declaration mismatch	DECL_MISMATCH
Destination buffer overflow in string manipulation	STRLIB_BUFFER_OVERFLOW
Destination buffer underflow in string manipulation	STRLIB_BUFFER_UNDERFLOW
Double lock	DOUBLE_LOCK
Double unlock	DOUBLE_UNLOCK
Environment pointer invalidated by previous operation	INVALID_ENV_POINTER
Errno not reset	MISSING_ERRNO_RESET
Exception caught by value	EXCP_CAUGHT_BY_VALUE
Exception handler hidden by previous handler	EXCP_HANDLER_HIDDEN
Float conversion overflow	FLOAT_CONV_OVFL
Float division by zero	FLOAT_ZERO_DIV
Format string specifiers and arguments mismatch	STRING_FORMAT
Improper array initialization	IMPROPER_ARRAY_INIT

Defect	Command-line Name
Incompatible types prevent overriding	VIRTUAL_FUNC_HIDING
Incorrect data type passed to va_arg	VA_ARG_INCORRECT_TYPE
Incorrect pointer scaling	BAD_PTR_SCALING
Incorrect type data passed to va_start	VA_START_INCORRECT_TYPE
Incorrect use of offsetof in C++	OFFSETOF_MISUSE
Incorrect use of va_start	VA_START_MISUSE
Inline constraint not respected	INLINE_CONSTRAINT_NOT_RESPECTED
Integer conversion overflow	INT_CONV_OVFL
Integer division by zero	INT_ZERO_DIV
Invalid assumptions about memory organization	INVALID_MEMORY_ASSUMPTION
Invalid free of pointer	BAD_FREE
Invalid use of standard library floating point routine	FLOAT_STD_LIB
Invalid use of standard library integer routine	INT_STD_LIB
Invalid use of standard library memory routine	MEM_STD_LIB
Invalid use of standard library routine	OTHER_STD_LIB
Invalid use of standard library string routine	STR_STD_LIB
Invalid va_list argument	INVALID_VA_LIST_ARG
Memory comparison of padding data	MEMCMP_PADDING_DATA
Memory comparison of strings	MEMCMP_STRINGS
Missing lock	BAD_UNLOCK
Missing return statement	MISSING_RETURN

Defect	Command-line Name
Missing unlock	BAD_LOCK
Misuse of a FILE object	FILE_OBJECT_MISUSE
Misuse of errno	ERRNO_MISUSE
Misuse of errno in a signal handler	SIG_HANDLER_ERRNO_MISUSE
Misuse of sign-extended character value	CHARACTER_MISUSE
Misuse of structure with flexible array member	FLEXIBLE_ARRAY_MEMBER_STRUCT_MISUSE
Non-initialized pointer	NON_INIT_PTR
Non-initialized variable	NON_INIT_VAR
Null pointer	NULL_PTR
Object slicing	OBJECT_SLICING
Opening previously opened resource	DOUBLE_RESOURCE_OPEN
Partial override of overloaded virtual functions	PARTIAL_OVERRIDE
Pointer access out of bounds	OUT_BOUND_PTR
Pointer or reference to stack variable leaving scope	LOCAL_ADDR_ESCAPE
Possible misuse of sizeof	SIZEOF_MISUSE
Possibly unintended evaluation of expression because of operator precedence rules	OPERATOR_PRECEDENCE
Predefined macro used as an object	MACRO_USED_AS_OBJECT
Preprocessor directive in macro argument	PRE_DIRECTIVE_MACRO_ARG
Resource leak	RESOURCE_LEAK
Return from computational exception signal handler	SIG_HANDLER_COMP_EXCP_RETURN

Defect	Command-line Name
Shared data access within signal handler	SIG_HANDLER_SHARED_OBJECT
Side effect of expression ignored	SIDE_EFFECT_IGNORED
Sign change integer conversion overflow	SIGN_CHANGE
Signal call from within signal handler	SIG_HANDLER_CALLING_SIGNAL
Standard function call with incorrect arguments	STD_FUNC_ARG_MISMATCH
Stream argument with possibly unintended side effects	STREAM_WITH_SIDE_EFFECT
Subtraction or comparison between pointers to different arrays	PTR_TO_DIFF_ARRAY
Too many va_arg calls for current argument list	TOO_MANY_VA_ARG_CALLS
Typedef mismatch	TYPEDEF_MISMATCH
Universal character name from token concatenation	PRE_UCNAME_JOIN_TOKENS
Unreachable code	UNREACHABLE
Unreliable cast of function pointer	FUNC_CAST
Unreliable cast of pointer	PTR_CAST
Unsigned integer conversion overflow	UINT_CONV_OVFL
Use of automatic variable as putenv-family function argument	PUTENV_AUTO_VAR
Use of previously closed resource	CLOSED_RESOURCE_USE
Use of previously freed pointer	FREED_PTR
Useless if	USELESS_IF
Variable length array with nonpositive size	NON_POSITIVE_VLA_SIZE

Defect	Command-line Name
Writing to const qualified object	CONSTANT_OBJECT_WRITE
Writing to read-only resource	READ_ONLY_RESOURCE_WRITE

Bug Finder Results Found in Fast Analysis Mode

In fast analysis mode, Bug Finder checks for a subset of defects and coding rules only. The tables below list the results that can be found in a fast analysis. See also `Use fast analysis mode for Bug Finder (-fast-analysis)`.

Polyspace Bug Finder Defects

Static Memory

Name	Description
Buffer overflow from incorrect string format specifier (<code>str_format_buffer_overflow</code>)	String format specifier causes buffer argument of standard library functions to overflow
Unreliable cast of function pointer (<code>func_cast</code>)	Function pointer cast to another function pointer with different argument or return type
Unreliable cast of pointer (<code>ptr_cast</code>)	Pointer implicitly cast to different data type

Programming

Name	Description
Copy of overlapping memory (overlapping_copy)	Source and destination arguments of a copy function have overlapping memory
Exception caught by value (excp_caught_by_value)	catch statement accepts an object by value
Exception handler hidden by previous handler (excp_handler_hidden)	catch statement is not reached because of an earlier catch statement for the same exception
Format string specifiers and arguments mismatch (string_format)	String specifiers do not match corresponding arguments
Improper array initialization (improper_array_init)	Incorrect array initialization when using initializers
Invalid use of == (equality) operator (bad_equal_equal_use)	Equality operation in assignment statement
Invalid use of = (assignment) operator (bad_equal_use)	Assignment in conditional statement
Invalid use of floating point operation (bad_float_op)	Imprecise comparison of floating point variables
Missing null in string array (missing_null_char)	String does not terminate with null character
Overlapping assignment (overlapping_assign)	Memory overlap between left and right sides of an assignment
Possibly unintended evaluation of expression because of operator precedence rules (operator_precedence)	Operator precedence rules cause unexpected evaluation order in arithmetic expression
Unsafe conversion between pointer and integer (bad_int_ptr_cast)	Misaligned or invalid results from conversions between pointer and integer types
Wrong type used in sizeof (ptr_sizeof_mismatch)	sizeof argument does not match pointed type

Data Flow

Name	Description
Code deactivated by constant false condition (deactivated_code)	Code segment deactivated by <code>#if 0</code> directive or <code>if(0)</code> condition
Missing return statement (missing_return)	Function does not return value though return type is not void
Static uncalled function (uncalled_func)	Function with static scope not called in file
Variable shadowing (var_shadowing)	Variable hides another variable of same name with nested scope

Object Oriented

Name	Description
*this not returned in copy assignment operator (return_not_ref_to_this)	operator= method does not return a pointer to the current object
Base class assignment operator not called (missing_base_assign_op_call)	Copy assignment operator does not call copy assignment operators of base subobjects
Base class destructor not virtual (dtor_not_virtual)	Class cannot behave polymorphically for deletion of derived class objects
Copy constructor not called in initialization list (missing_copy_ctor_call)	Copy constructor does not call copy constructors of some members or base classes
Incompatible types prevent overriding (virtual_func_hiding)	Derived class method hides a virtual base class method instead of overriding it
Member not initialized in constructor (non_init_member)	Constructor does not initialize some members of a class
Missing explicit keyword (missing_explicit_keyword)	Constructor missing the explicit specifier
Missing virtual inheritance (missing_virtual_inheritance)	A base class is inherited virtually and nonvirtually in the same hierarchy
Object slicing (object_slicing)	Derived class object passed by value to function with base class parameter
Partial override of overloaded virtual functions (partial_override)	Class overrides fraction of inherited virtual functions with a given name
Return of non const handle to encapsulated data member (breaking_data_encapsulation)	Method returns pointer or reference to internal member of object
Self assignment not tested in operator (missing_self_assign_test)	Copy assignment operator does not test for self-assignment

Security

Name	Description
Function pointer assigned with absolute address (func_ptr_absolute_addr)	Constant expression is used as function address is vulnerable to code injection

Good Practice

Name	Description
Bitwise and arithmetic operation on the same data (bitwise_arith_mix)	Statement with mixed bitwise and arithmetic operations
Delete of void pointer (delete_of_void_ptr)	delete operates on a void* pointer pointing to an object
Hard-coded buffer size (hard_coded_buffer_size)	Size of memory buffer is a numerical value instead of symbolic constant
Hard-coded loop boundary (hard_coded_loop_boundary)	Loop boundary is a numerical value instead of symbolic constant
Large pass-by-value argument (pass_by_value)	Large argument passed by value between functions
Line with more than one statement (more_than_one_statement)	Multiple statements on a line
Missing break of switch case (missing_switch_break)	No comments at the end of switch case without a break statement
Missing reset of a freed pointer (missing_freed_ptr_reset)	Pointer free not followed by a reset statement to clear leftover data
Unused parameter (unused_parameter)	Function prototype has parameters not read or written in function body

MISRA C: 2004 and MISRA AC AGC Rules

The software checks the following rules early in the analysis.

Language Extensions

Rule	Description
2.1	Assembly language shall be encapsulated and isolated.
2.2	Source code shall only use /* */ style comments.
2.3	The character sequence /* shall not be used within a comment.

Documentation

Rule	Description
3.4	All uses of the <code>#pragma</code> directive shall be documented and explained.

Character Sets

Rule	Description
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.
4.2	Trigraphs shall not be used.

Identifiers

Rule	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

Types

Rule	Description
6.1	The plain <code>char</code> type shall be used only for the storage and use of character values.
6.2	Signed and unsigned <code>char</code> type shall be used only for the storage and use of numeric values.
6.3	<code>typedefs</code> that indicate size and signedness should be used in place of the basic types.
6.4	Bit fields shall only be defined to be of type <code>unsigned int</code> or <code>signed int</code> .
6.5	Bit fields of type <code>signed int</code> shall be at least 2 bits long.

Constants

Rule	Description
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and Definitions

Rule	Description
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
8.5	There shall be no definitions of objects or functions in a header file.
8.6	Functions shall always be declared at file scope.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8	An external object or function shall be declared in one file and only one file.
8.9	An identifier with external linkage shall have exactly one external definition.
8.11	The <code>static</code> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

Rule	Description
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.
9.3	In an enumerator list, the <code>=</code> construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic Type Conversion

Rule	Description
10.1	<p>The value of an expression of integer type shall not be implicitly converted to a different underlying type if:</p> <ul style="list-style-type: none">• It is not a conversion to a wider integer type of the same signedness, or• The expression is complex, or• The expression is not constant and is a function argument, or• The expression is not constant and is a return expression
10.2	<p>The value of an expression of floating type shall not be implicitly converted to a different type if</p> <ul style="list-style-type: none">• It is not a conversion to a wider floating type, or• The expression is complex, or• The expression is a function argument, or• The expression is a return expression
10.3	<p>The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.</p>
10.4	<p>The value of a complex expression of float type may only be cast to narrower floating type.</p>
10.5	<p>If the bitwise operator <code>~</code> and <code><<</code> are applied to an operand of underlying type <code>unsigned char</code> or <code>unsigned short</code>, the result shall be immediately cast to the underlying type of the operand</p>
10.6	<p>The "U" suffix shall be applied to all constants of unsigned types.</p>

Pointer Type Conversion

Rule	Description
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to <code>void</code> .
11.3	A cast should not be performed between a pointer type and an integral type.
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.
11.5	A cast shall not be performed that removes any <code>const</code> or <code>volatile</code> qualification from the type addressed by a pointer

Expressions

Rule	Description
12.1	Limited dependence should be placed on C's operator precedence rules in expressions.
12.3	The <code>sizeof</code> operator should not be used on expressions that contain side effects.
12.5	The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions.
12.6	Operands of logical operators (<code>&&</code> , <code> </code> and <code>!</code>) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (<code>&&</code> , <code> </code> or <code>!</code>).
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed.
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
12.10	The comma operator shall not be used.
12.11	Evaluation of constant unsigned expression should not lead to wraparound.
12.12	The underlying bit representations of floating-point values shall not be used.
12.13	The increment (<code>++</code>) and decrement (<code>--</code>) operators should not be mixed with other operators in an expression

Control Statement Expressions

Rule	Description
13.1	Assignment operators shall not be used in expressions that yield Boolean values.
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a <code>for</code> statement shall not contain any objects of floating type.
13.5	The three expressions of a <code>for</code> statement shall be concerned only with loop control.
13.6	Numeric variables being used within a <code>for</code> loop for iteration counting should not be modified in the body of the loop.

Control Flow

Rule	Description
14.3	All non-null statements shall either <ul style="list-style-type: none"> • have at least one side effect however executed, or • cause control flow to change.
14.4	The <code>goto</code> statement shall not be used.
14.5	The <code>continue</code> statement shall not be used.
14.6	For any iteration statement, there shall be at most one <code>break</code> statement used for loop termination.
14.7	A function shall have a single point of exit at the end of the function.
14.8	The statement forming the body of a <code>switch</code> , <code>while</code> , <code>do while</code> or <code>for</code> statement shall be a compound statement.
14.9	An <code>if</code> (expression) construct shall be followed by a compound statement. The <code>else</code> keyword shall be followed by either a compound statement, or another <code>if</code> statement.
14.10	All <code>if else if</code> constructs should contain a final <code>else</code> clause.

Switch Statements

Rule	Description
15.0	Unreachable code is detected between <code>switch</code> statement and first <code>case</code> .
15.1	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement
15.2	An unconditional <code>break</code> statement shall terminate every non-empty <code>switch</code> clause.
15.3	The final clause of a <code>switch</code> statement shall be the <code>default</code> clause.
15.4	A <code>switch</code> expression should not represent a value that is effectively Boolean.
15.5	Every <code>switch</code> statement shall have at least one <code>case</code> clause.

Functions

Rule	Description
16.1	Functions shall not be defined with variable numbers of arguments.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.
16.5	Functions with no parameters shall be declared with parameter type <code>void</code> .
16.6	The number of arguments passed to a function shall match the number of parameters.
16.8	All exit paths from a function with non- <code>void</code> return type shall have an explicit return statement with an expression.
16.9	A function identifier shall only be used with either a preceding <code>&</code> , or with a parenthesized parameter list, which may be empty.

Pointers and Arrays

Rule	Description
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	A type should not contain more than 2 levels of pointer indirection.

Structures and Unions

Rule	Description
18.1	All structure or union types shall be complete at the end of a translation unit.
18.4	Unions shall not be used.

Preprocessing Directives

Rule	Description
19.1	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments.
19.2	Nonstandard characters should not occur in header file names in <code>#include</code> directives.
19.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or "filename" sequence.
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
19.5	Macros shall not be <code>#define-d</code> and <code>#undef-d</code> within a block.
19.6	<code>#undef</code> shall not be used.
19.7	A function should be used in preference to a function like-macro.
19.8	A function-like macro shall not be invoked without all of its arguments.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
19.10	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of <code>#</code> or <code>##</code> .
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in <code>#ifdef</code> and <code>#ifndef</code> preprocessor directives and the <code>defined()</code> operator.
19.12	There shall be at most one occurrence of the <code>#</code> or <code>##</code> preprocessor operators in a single macro definition.
19.13	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
19.14	The <code>defined</code> preprocessor operator shall only be used in one of the two standard forms.
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.
19.17	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> or <code>#ifdef</code> directive to which they are related.

Standard Libraries

Rule	Description
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
20.2	The names of standard library macros, objects and functions shall not be reused.
20.4	Dynamic heap memory allocation shall not be used.
20.5	The error indicator <code>errno</code> shall not be used.
20.6	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.
20.7	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.
20.8	The signal handling facilities of <code><signal.h></code> shall not be used.
20.9	The input/output library <code><stdio.h></code> shall not be used in production code.
20.10	The library functions <code>atof</code> , <code>atoi</code> and <code>atoll</code> from library <code><stdlib.h></code> shall not be used.
20.11	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <code><stdlib.h></code> shall not be used.
20.12	The time handling functions of library <code><time.h></code> shall not be used.

MISRA C: 2012 Rules

Standard C Environment

Rule	Description
1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.
1.2	Language extensions should not be used.

Unused Code

Rule	Description
2.6	A function should not contain unused label declarations.
2.7	There should be no unused parameters in functions.

Comments

Rule	Description
3.1	The character sequences /* and // shall not be used within a comment.
3.2	Line-splicing shall not be used in // comments.

Character Sets and Lexical Conventions

Rule	Description
4.1	Octal and hexadecimal escape sequences shall be terminated.
4.2	Trigraphs should not be used.

Identifiers

Rule	Description
5.2	Identifiers declared in the same scope and name space shall be distinct.
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
5.4	Macro identifiers shall be distinct.
5.5	Identifiers shall be distinct from macro names.

Types

Rule	Description
6.1	Bit-fields shall only be declared with an appropriate type.
6.2	Single-bit named bit fields shall not be of a signed type.

Literals and Constants

Rule	Description
7.1	Octal constants shall not be used.
7.2	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
7.3	The lowercase character "l" shall not be used in a literal suffix.
7.4	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".

Declarations and Definitions

Rule	Description
8.1	Types shall be explicitly specified.
8.2	Function types shall be in prototype form with named parameters.
8.4	A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5	An external object or function shall be declared once in one and only one file.
8.6	An identifier with external linkage shall have exactly one external definition.
8.8	The <code>static</code> storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.10	An inline function shall be declared with the <code>static</code> storage class.
8.11	When an array with external linkage is declared, its size should be explicitly specified.
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.14	The <code>restrict</code> type qualifier shall not be used.

Initialization

Rule	Description
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.
9.4	An element of an object shall not be initialized more than once.
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The Essential Type Model

Rule	Description
10.1	Operands shall not be of an inappropriate essential type.
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
10.5	The value of an expression should not be cast to an inappropriate essential type.
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer Type Conversion

Rule	Description
11.1	Conversions shall not be performed between a pointer to a function and any other type.
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
11.9	The macro NULL shall be the only permitted form of integer null pointer constant.

Expressions

Rule	Description
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used.
12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side Effects

Rule	Description
13.3	A full expression containing an increment (++) or decrement (- -) operator should have no other potential side effects other than that caused by the increment or decrement operator.
13.4	The result of an assignment operator should not be used.
13.6	The operand of the sizeof operator shall not contain any expression which has potential side effects.

Control Statement Expressions

Rule	Description
14.4	The controlling expression of an <code>if</code> statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control Flow

Rule	Description
15.1	The <code>goto</code> statement should not be used.
15.2	The <code>goto</code> statement shall jump to a label declared later in the same function.
15.3	Any label referenced by a <code>goto</code> statement shall be declared in the same block, or in any block enclosing the <code>goto</code> statement.
15.4	There should be no more than one <code>break</code> or <code>goto</code> statement used to terminate any iteration statement.
15.5	A function should have a single point of exit at the end
15.6	The body of an iteration-statement or a selection-statement shall be a compound statement.
15.7	All <code>if ... else if</code> constructs shall be terminated with an <code>else</code> statement.

Switch Statements

Rule	Description
16.1	All <code>switch</code> statements shall be well-formed.
16.2	A <code>switch</code> label shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement.
16.3	An unconditional <code>break</code> statement shall terminate every <code>switch</code> -clause.
16.4	Every <code>switch</code> statement shall have a <code>default</code> label.
16.5	A <code>default</code> label shall appear as either the first or the last <code>switch</code> label of a <code>switch</code> statement.
16.6	Every <code>switch</code> statement shall have at least two <code>switch</code> -clauses.
16.7	A <code>switch</code> -expression shall not have essentially Boolean type.

Functions

Rule	Description
17.1	The features of <starg.h> shall not be used.
17.3	A function shall not be declared implicitly.
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
17.6	The declaration of an array parameter shall not contain the <code>static</code> keyword between the [].
17.7	The value returned by a function having non-void return type shall be used.

Pointers and Arrays

Rule	Description
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type.
18.5	Declarations should contain no more than two levels of pointer nesting.
18.7	Flexible array members shall not be declared.
18.8	Variable-length array types shall not be used.

Overlapping Storage

Rule	Description
19.2	The union keyword should not be used.

Preprocessing Directives

Rule	Description
20.1	<code>#include</code> directives should only be preceded by preprocessor directives or comments.
20.2	The <code>'</code> , <code>"</code> , or <code>\</code> characters and the <code>/*</code> or <code>//</code> character sequences shall not occur in a header file name.
20.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or <code>"filename\"</code> sequence.
20.4	A macro shall not be defined with the same name as a keyword.
20.5	<code>#undef</code> should not be used.
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument.
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.
20.8	The controlling expression of a <code>#if</code> or <code>#elif</code> preprocessing directive shall evaluate to 0 or 1.
20.9	All identifiers used in the controlling expression of <code>#if</code> or <code>#elif</code> preprocessing directives shall be <code>#define</code> 'd before evaluation.
20.10	The <code>#</code> and <code>##</code> preprocessor operators should not be used.
20.11	A macro parameter immediately following a <code>#</code> operator shall not immediately be followed by a <code>##</code> operator.
20.12	A macro parameter used as an operand to the <code>#</code> or <code>##</code> operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
20.13	A line whose first token is <code>#</code> shall be a valid preprocessing directive.
20.14	All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as the <code>#if</code> , <code>#ifdef</code> or <code>#ifndef</code> directive to which they are related.

Standard Libraries

Rule	Description
21.1	<code>#define</code> and <code>#undef</code> shall not be used on a reserved identifier or reserved macro name.
21.2	A reserved identifier or macro name shall not be declared.
21.3	The memory allocation and deallocation functions of <code><stdlib.h></code> shall not be used.
21.4	The standard header file <code><setjmp.h></code> shall not be used.
21.5	The standard header file <code><signal.h></code> shall not be used.
21.6	The Standard Library input/output functions shall not be used.
21.7	The <code>atof</code> , <code>atoi</code> , <code>atol</code> , and <code>atoll</code> functions of <code><stdlib.h></code> shall not be used.
21.8	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> of <code><stdlib.h></code> shall not be used.
21.9	The library functions <code>bsearch</code> and <code>qsort</code> of <code><stdlib.h></code> shall not be used.
21.10	The Standard Library time and date functions shall not be used.
21.11	The standard header file <code><tgmath.h></code> shall not be used.
21.12	The exception handling features of <code><fenv.h></code> should not be used.

MISRA C++ 2008 Rules

Language Independent Issues

Rule	Description
0-1-7	The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.
0-1-11	There shall be no unused parameters (named or unnamed) in non- virtual functions.
0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.
0-2-1	An object shall not be assigned to an overlapping object.

General

Rule	Description
1-0-1	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".

Lexical Conventions

Rule	Description
2-3-1	Trigraphs shall not be used.
2-5-1	Digraphs should not be used.
2-7-1	The character sequence /* shall not be used within a C-style comment.
2-10-1	Different identifiers shall be typographically unambiguous.
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
2-10-3	A typedef name (including qualification, if any) shall be a unique identifier.
2-10-4	A class, union or enum name (including qualification, if any) shall be a unique identifier.
2-10-6	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.
2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.
2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.
2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.
2-13-4	Literal suffixes shall be upper case.
2-13-5	Narrow and wide string literals shall not be concatenated.

Basic Concepts

Rule	Description
3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.
3-1-2	Functions shall not be declared at block scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-1	Objects or functions with external linkage shall be declared in a header file.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-1	The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.
3-9-2	Typedefs that indicate size and signedness should be used in place of the basic numerical types.
3-9-3	The underlying bit representations of floating-point values shall not be used.

Standard Conversions

Rule	Description
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, , !, the equality operators == and !=, the unary & operator, and the conditional operator.
4-5-2	Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.
4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.

Expressions

Rule	Description
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.
5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type.
5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.
5-0-5	There shall be no implicit floating-integral conversions.
5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.
5-0-10	If the bitwise operators <code>~</code> and <code><<</code> are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.
5-0-11	The plain char type shall only be used for the storage and use of character values.
5-0-12	signed char and unsigned char type shall only be used for the storage and use of numeric values.
5-0-13	The condition of an if-statement and the condition of an iteration-statement shall have type bool.
5-0-14	The first operand of a conditional-operator shall have type bool.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	<code>></code> , <code>>=</code> , <code><</code> , <code><=</code> shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type.
5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type.

Rule	Description
5-2-1	Each operand of a logical && or shall be a postfix - expression.
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.
5-2-3	Casts from a base class to a derived class should not be performed on polymorphic types.
5-2-4	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
5-2-10	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.
5-2-11	The comma operator, && operator and the operator shall not be overloaded.
5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer.
5-3-1	Each operand of the ! operator, the logical && or the logical operators shall have type bool.
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3	The unary & operator shall not be overloaded.
5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects.
5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.
5-14-1	The right hand operand of a logical && or operator shall not contain side effects.
5-18-1	The comma operator shall not be used.
5-19-1	Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Statements

Rule	Description
6-2-1	Assignment operators shall not be used in sub-expressions.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.
6-4-2	All if ... else if constructs shall be terminated with an else clause.
6-4-3	A switch statement shall be a well-formed switch statement.
6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.
6-4-5	An unconditional throw or break statement shall terminate every non - empty switch-clause.
6-4-6	The final clause of a switch statement shall be the default-clause.
6-4-7	The condition of a switch statement shall not have bool type.
6-4-8	Every switch statement shall have at least one case-clause.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.
6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Rule	Description
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-3	The continue statement shall only be used within a well-formed for loop.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.

Declarations

Rule	Description
7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations.
7-3-2	The identifier main shall not be used for a function other than the global function main.
7-3-3	There shall be no unnamed namespaces in header files.
7-3-4	using-directives shall not be used.
7-3-5	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.
7-3-6	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.
7-4-2	Assembler instructions shall only be introduced using the asm declaration.
7-4-3	Assembly language shall be encapsulated and isolated.

Declarators

Rule	Description
8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.
8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.
8-4-1	Functions shall not be defined using the ellipsis notation.
8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.
8-4-3	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.
8-5-2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Classes

Rule	Description
9-3-1	const member functions shall not return non-const pointers or references to class-data.
9-3-2	Member functions shall not return non-const handles to class-data.
9-5-1	Unions shall not be used.
9-6-2	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.
9-6-3	Bit-fields shall not have enum type.
9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit.

Derived Classes

Rule	Description
10-1-1	Classes should not be derived from virtual bases.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and non-virtual in the same hierarchy.
10-2-1	All accessible entity names within a multiple inheritance hierarchy should be unique.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Member Access Control

Rule	Description
11-0-1	Member data in non- POD class types shall be private.

Special Member Functions

Rule	Description
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.
12-1-2	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.
12-1-3	All constructors that are callable with a single argument of fundamental type shall be declared explicit.
12-8-1	A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.

Templates

Rule	Description
14-5-2	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.
14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.
14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.
14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.
14-7-3	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.
14-8-1	Overloaded function templates shall not be explicitly specialized.
14-8-2	The viable function set for a function call should either contain no function specializations, or only contain function specializations.

Exception Handling

Rule	Description
15-0-2	An exception object should not have pointer type.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-2	NULL shall not be thrown explicitly.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-2	There should be at least one exception handler to catch all otherwise unhandled exceptions
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).

Preprocessing Directives

Rule	Description
16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments.
16-0-2	Macros shall only be #define 'd or #undef 'd in the global namespace.
16-0-3	#undef shall not be used.
16-0-4	Function-like macros shall not be defined.
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.
16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.
16-1-1	The defined preprocessor operator shall only be used in one of the two standard forms.
16-1-2	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.
16-2-1	The pre-processor shall only be used for file inclusion and include guards.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
16-2-3	Include guards shall be provided.
16-2-4	The ', ", /* or // characters shall not occur in a header file name.
16-2-5	The \ character should not occur in a header file name.
16-2-6	The #include directive shall be followed by either a <filename> or "filename" sequence.
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.
16-3-2	The # and ## operators should not be used.
16-6-1	All uses of the #pragma directive shall be documented.
17-0-1	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Rule	Description
17-0-2	The names of standard library macros and objects shall not be reused.
17-0-5	The setjmp macro and the longjmp function shall not be used.

Language Support Library

Rule	Description
18-0-1	The C library shall not be used.
18-0-2	The library functions atof, atoi and atol from library <cstdlib> shall not be used.
18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.
18-0-4	The time handling functions of library <ctime> shall not be used.
18-0-5	The unbounded functions of library <cstring> shall not be used.
18-2-1	The macro offsetof shall not be used.
18-4-1	Dynamic heap memory allocation shall not be used.
18-7-1	The signal handling facilities of <csignal> shall not be used.

Diagnostic Library

Rule	Description
19-3-1	The error indicator errno shall not be used.

Input/Output Library

Rule	Description
27-0-1	The stream input/output library <stdio> shall not be used.

CWE Coding Standard and Polyspace Results

Common Weakness Enumeration (CWE) is a dictionary of common software weakness types that can occur in software architecture, design, code, or implementation. These weaknesses can lead to security vulnerabilities.

CWE and Polyspace Bug Finder

The CWE dictionary assigns a unique identifier to each software weakness type. These identifiers serve as a common language for describing software security weaknesses and a standard for software security tools targeting these weaknesses. For more information, see Common Weakness Enumeration.

Polyspace Bug Finder results can be mapped to CWE identifiers. Using Bug Finder, you can check and document if your software has weaknesses listed in the CWE dictionary. Bug Finder supports the following aspects of the CWE Compatibility and Effectiveness Program:

- **CWE Searchable:** For each supported CWE identifier, you can see all instances in your code that have weaknesses corresponding to the identifier.
- **CWE Output:** For each Polyspace Bug Finder defect:
 - You can view the associated CWE identifier.
 - You can report the associated CWE identifier.

Bug Finder results are mapped to CWE identifiers (IDs). Using the Bug Finder results, you can evaluate your code against the CWE standard. For instance, CWE ID 119 (Improper restriction of operations within the bounds of a memory buffer) maps to the Bug Finder defects, **Array access out of bounds** and **Pointer access out of bounds**.

For more information on the CWE Compatibility and Effectiveness Program, see CWE Compatibility.

Find CWE IDs from Polyspace Results

Use the following workflow if you want to focus your Bug Finder analysis on the CWE standard.

- *Analysis:* Check your code only for those Bug Finder defects that correspond to the standard. Use the option `Find defects (-checkers)` with value `CWE`.

- *Results:* If you enable only the defect checkers corresponding to the CWE standard, you see only the defects that correspond to the standard. Fix or justify each defect.

Along with defects, you can see the CWE IDs mapped to each defect in the **CWE ID** column on the **Results List** pane. If the column is not enabled by default, right-click any column header and select **CWE ID**.

- *Report:* When you generate a report, choose the **SecurityCWE** template tailored for the CWE standard. The report shows the CWE ID-s corresponding to each result.

Mapping Between CWE Identifiers and Polyspace Results

The following table lists the CWE IDs (version 2.8) addressed by Polyspace Bug Finder with its corresponding defect checkers. Using Polyspace Bug Finder defect checkers, you can check for 133 CWE IDs.

There are three types of CWE identifiers: Class, Base and Variant. Identifiers of type Class define security weaknesses at an abstract level independent of a specific language or technology, while identifiers of type Base and Variant are more concrete. On the other hand, Polyspace Bug Finder results are designed to be specific so that users can have a precise diagnosis of the defect in their code and understand the defect quickly. Therefore:

- The Bug Finder results are mapped to the specific identifiers of type Base and Variant rather than the generic identifiers of type Class.

Only when a result covers more ground than a specific CWE identifier is the result mapped to its more general parent type. For instance, the defect checker **Array access out of bounds** covers many kinds of buffer overflows, while CWE-788 refers only to “Access of Memory Location After End of Buffer”. Therefore, the defect checker is mapped to its parent, CWE-119, which refers to “Improper Restriction of Operations within the Bounds of a Memory Buffer”. However, to keep the mapping precise, an attempt is made to map to specific CWE identifiers.

- Often, more than one Bug Finder result is mapped to a certain CWE identifier.

For instance, CWE-908 refers to “Use of Uninitialized Resource”. To highlight specific kinds of uninitialized resources, Bug Finder has three different checkers: **Member not initialized in constructor**, **Non-initialized pointer**, and **Non-initialized variable**.

For mapping to the subsets CWE-658 and CWE-659, see “Mapping Between CWE-658 or 659 and Polyspace Results” on page 9-100.

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
15	External control of system or configuration setting	Host change using externally controlled elements Use of externally controlled environment variable
20	Improper input validation	Unsafe conversion from string to numerical value
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	Vulnerable path manipulation
23	Relative path traversal	Vulnerable path manipulation
36	Absolute path traversal	Vulnerable path manipulation
67	Improper Handling of Windows Device Names	Inappropriate I/O operation on device files
77	Improper neutralization of special elements used in a command	Execution of externally controlled command Unsafe call to a system function
78	Improper neutralization of special elements used in an OS command	Execution of externally controlled command Unsafe call to a system function
88	Argument injection or modification	Execution of externally controlled command Unsafe call to a system function

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
114	Process control	<p>Command executed from externally controlled path</p> <p>Execution of a binary from a relative path can be controlled by an external actor</p> <p>Execution of externally controlled command</p> <p>Library loaded from externally controlled path</p> <p>Load of library from a relative path can be controlled by an external actor</p>
119	Improper restriction of operations within the bounds of a memory buffer	<p>Array access out of bounds</p> <p>Pointer access out of bounds</p>
120	Buffer copy without checking size of input ('Classic buffer overflow')	<p>Invalid use of standard library memory routine</p> <p>Invalid use of standard library string routine</p> <p>Tainted NULL or non-null-terminated string</p>
121	Stack-based buffer overflow	<p>Array access with tainted index</p> <p>Destination buffer overflow in string manipulation</p>
122	Heap-based buffer overflow	<p>Pointer dereference with tainted offset</p>

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
124	Buffer underwrite ('Buffer underflow')	Array access with tainted index Buffer overflow from incorrect string format specifier Destination buffer underflow in string manipulation Pointer dereference with tainted offset
125	Out-of-bounds read	Array access with tainted index Buffer overflow from incorrect string format specifier Destination buffer overflow in string manipulation
126	Buffer over-read	Buffer overflow from incorrect string format specifier
127	Buffer under-read	Buffer overflow from incorrect string format specifier
128	Wrap-around error	Integer constant overflow Tainted sign change conversion Unsigned integer conversion overflow Memory allocation with tainted size Unsigned integer overflow Tainted size of variable length array Unsigned integer constant overflow Integer overflow Integer conversion overflow

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
129	Improper validation of array index	Array access with tainted index Pointer dereference with tainted offset
130	Improper handling of length parameter inconsistency	Mismatch between data length and size
131	Incorrect calculation of buffer size	Pointer access out of bounds Tainted sign change conversion Unsigned integer conversion overflow Memory allocation with tainted size Unsigned integer overflow Array access out of bounds Tainted size of variable length array
134	Uncontrolled format string	Tainted string format
135	Incorrect Calculation of Multi-Byte String Length	Destination buffer overflow in string manipulation Misuse of narrow or wide character string Unreliable cast of pointer
170	Improper null termination	Missing null in string array Misuse of readlink() Tainted NULL or non-null-terminated string

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
188	Reliance on data/ memory layout	Invalid assumptions about memory organization Memory comparison of padding data Memory comparison of strings Missing byte reordering when transferring data Pointer access out of bounds
189	Numeric Errors	Absorption of float operand Float conversion overflow Float division by zero Float overflow Integer constant overflow Integer conversion overflow Integer division by zero Integer overflow Precision loss in integer to float conversion Shift of a negative value Shift operation overflow Tainted division operand Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
190	Integer overflow or wraparound	Integer constant overflow Integer conversion overflow Integer precision exceeded Integer overflow Possible invalid operation on boolean operand Shift operation overflow Tainted division operand Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow
191	Integer underflow (Wrap or wraparound)	Integer constant overflow Integer conversion overflow Integer overflow Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
192	Integer coercion error	Tainted sign change conversion Unsigned integer conversion overflow Unsigned integer overflow Sign change integer conversion overflow Integer overflow Integer conversion overflow
194	Unexpected sign extension	Sign change integer conversion overflow Tainted sign change conversion
195	Signed to unsigned conversion error	Sign change integer conversion overflow Tainted sign change conversion
196	Unsigned to signed conversion error	Sign change integer conversion overflow
197	Numeric truncation error	Integer conversion overflow Float conversion overflow Unsigned integer conversion overflow
198		Missing byte reordering when transferring data
226	Sensitive information uncleared before release	Uncleared sensitive data in stack

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
227	Improper fulfillment of API contract	Invalid use of standard library floating point routine Invalid use of standard library integer routine Invalid use of standard library memory routine Invalid use of standard library routine Invalid use of standard library string routine Writing to const qualified object
240	Improper handling of inconsistent structural elements	Mismatch between data length and size
242	Use of inherently dangerous function	Use of dangerous standard function
243	Creation of chroot jail without changing working directory	File manipulation after chroot without chdir
244	Improper clearing of heap memory before release	Sensitive heap memory not cleared before release
250	Execution with unnecessary privileges	Bad order of dropping privileges Privilege drop not verified
251	Often misused: string management	Destination buffer overflow in string manipulation
252	Unchecked return value	Returned value of a sensitive function not checked

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
253	Incorrect Check of Function Return Value	Errno not checked Errno not reset Returned value of a sensitive function not checked Unprotected dynamic memory allocation Unsafe conversion from string to numerical value
273	Improper check for dropped privileges	Privilege drop not verified

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
310	Cryptographic issues	Constant cipher key Weak cipher mode Context initialized incorrectly for cryptographic operation Nonsecure hash algorithm Constant block cipher initialization vector Context initialized incorrectly for digest operation Missing parameters for key generation Missing data for encryption, decryption or signing operation Nonsecure SSL/TLS protocol Missing peer key Missing cipher key Missing cipher algorithm Missing private key Missing public key Predictable block cipher initialization vector Nonsecure parameters for key generation Predictable cipher key Weak padding for RSA algorithm

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
		Nonsecure RSA public exponent Missing padding for RSA algorithm Missing blinding for RSA algorithm Incorrect key for cryptographic algorithm Missing block cipher initialization vector Weak cipher algorithm Incompatible padding for RSA algorithm operation
311	Missing encryption of sensitive data	Missing cipher final step Missing cipher data to process
312	Cleartext Storage of Sensitive Information	Sensitive heap memory not cleared before release Uncleared sensitive data in stack
316	Cleartext Storage of Sensitive Information in Memory	Sensitive heap memory not cleared before release Uncleared sensitive data in stack
320	Key management errors	Constant cipher key Missing peer key Missing cipher key Missing private key Missing public key
321	Use of hard-coded cryptographic key	Constant cipher key

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
325	Missing required cryptographic step	Missing block cipher initialization vector Missing cipher algorithm Missing cipher data to process Missing cipher final step Missing cipher key Weak cipher algorithm Weak cipher mode Context initialized incorrectly for cryptographic operation Missing parameters for key generation Missing data for encryption, decryption or signing operation Incorrect key for cryptographic algorithm

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
326	Inadequate encryption strength	Weak cipher algorithm Constant cipher key Weak cipher mode Constant block cipher initialization vector Nonsecure parameters for key generation Predictable cipher key Weak padding for RSA algorithm Nonsecure RSA public exponent Missing padding for RSA algorithm Missing blinding for RSA algorithm Missing block cipher initialization vector
327	Use of a broken or risky cryptographic algorithm	Unsafe standard encryption function Weak cipher algorithm Weak cipher mode Nonsecure hash algorithm Nonsecure SSL/TLS protocol Nonsecure parameters for key generation Weak padding for RSA algorithm Nonsecure RSA public exponent Missing padding for RSA algorithm

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
328	Reversible one-way hash	Nonsecure hash algorithm
329	Not using a random IV with CBC mode	Missing block cipher initialization vector Predictable block cipher initialization vector Constant block cipher initialization vector
330	Use of insufficiently random values	Deterministic random output from constant seed Predictable random output from predictable seed Vulnerable pseudo-random number generator Predictable block cipher initialization vector Predictable cipher key
336	Same seed in PRNG	Deterministic random output from constant seed
337	Predictable seed in PRNG	Predictable random output from predictable seed
338	Use of cryptographically weak pseudo-random number generator (PRNG)	Vulnerable pseudo-random number generator Predictable block cipher initialization vector Predictable cipher key

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
353	Missing Support for Integrity Check	Context initialized incorrectly for digest operation Nonsecure hash algorithm
354	Improper Validation of Integrity Check Value	Context initialized incorrectly for digest operation
362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	File descriptor exposure to child process Opening previously opened resource
364	Signal handler race condition	Shared data access within signal handler Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe
366	Race condition within a thread	Data race Data race including atomic operations Data race through standard library function call
367	Time-of-check time-of-use (TOCTOU) race condition	File access between time of check and use (TOCTOU)

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
369	Divide by zero	Float division by zero Integer division by zero Invalid use of standard library floating point routine Invalid use of standard library integer routine Tainted division operand Tainted modulo operand
372	Incomplete internal state distinction	Context initialized incorrectly for cryptographic operation Context initialized incorrectly for digest operation Missing parameters for key generation Missing data for encryption, decryption or signing operation Inconsistent cipher operations Missing cipher data to process Incompatible padding for RSA algorithm operation Missing cipher final step
375	Returning a mutable object to an untrusted caller	Return of non const handle to encapsulated data member
377	Insecure temporary file	Use of non-secure temporary file

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
387	Signal errors	Return from computational exception signal handler Signal call from within signal handler Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict)
391	Unchecked error condition	Errno not checked
398	Indicator of poor code quality	Write without a further read
401	Improper release of memory before removing last reference	Memory leak Thread-specific memory leak
404	Improper resource shutdown or release	Invalid deletion of pointer Invalid free of pointer Memory leak Mismatched alloc/dealloc functions on Windows Thread-specific memory leak

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
413	Improper Resource Locking	Data race Data race including atomic operations Data race through standard library function call Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict) Opening previously opened resource Shared data access within signal handler
415	Double free	Deallocation of previously deallocated pointer Missing reset of a freed pointer
416	Use after free	Missing reset of a freed pointer Use of previously freed pointer
426	Untrusted search path	Command executed from externally controlled path Library loaded from externally controlled path

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
427	Uncontrolled search path element	<p>Execution of a binary from a relative path can be controlled by an external actor</p> <p>Library loaded from externally controlled path</p> <p>Load of library from a relative path can be controlled by an external actor</p> <p>Use of externally controlled environment variable</p>
456	Missing initialization of a variable	<p>Errno not reset</p> <p>Member not initialized in constructor</p> <p>Non-initialized pointer</p> <p>Non-initialized variable</p>
457	Use of uninitialized variable	<p>Member not initialized in constructor</p> <p>Non-initialized pointer</p> <p>Non-initialized variable</p>
465	Pointer Issues	Unsafe conversion between pointer and integer
466	Return of pointer value outside of expected range	<p>Array access out of bounds</p> <p>Pointer access out of bounds</p> <p>Unsafe conversion between pointer and integer</p>
467	Use of sizeof() on a pointer type	<p>Possible misuse of sizeof</p> <p>Wrong type used in sizeof</p>
468	Incorrect pointer scaling	Incorrect pointer scaling

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
469	Use of pointer subtraction to determine size	Subtraction or comparison between pointers to different arrays
471	Modification of assumed-immutable data	Writing to const qualified object
474	Use of function with inconsistent implementations	Signal call from within signal handler Use of obsolete standard function
475	Undefined behavior for input to API	Copy of overlapping memory
476	NULL pointer dereference	Null pointer Tainted NULL or non-null-terminated string
477	Use of obsolete functions	Use of obsolete standard function
478	Missing default case in switch statement	Missing case for switch condition
479	Signal handler use of a non-reentrant function	Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict)
480	Use of incorrect operator	Invalid use of == (equality) operator Invalid use of = (assignment) operator
481	Assigning instead of comparing	Invalid use of = (assignment) operator
482	Comparing instead of assigning	Invalid use of == (equality) operator
484	Omitted break statement in switch	Missing break of switch case

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
522	Insufficiently Protected Credentials	Constant cipher key Nonsecure hash algorithm Nonsecure parameters for key generation Nonsecure RSA public exponent Nonsecure SSL/TLS protocol Unsafe standard encryption function
532	Information exposure through log files	Sensitive data printed out
534	Information exposure through debug log files	Sensitive data printed out
535	Information exposure through shell error message	Sensitive data printed out
547	Use of hard-coded, security-relevant constants	Hard coded buffer size Hard coded loop boundary
558	Use of getlogin() in multithreaded application	Unsafe standard function
560	Use of umask() with chmod-style argument	Umask used with chmod-style arguments
561	Dead code	Dead code Static uncalled function Unreachable code

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
562	Return of stack variable address	<p>Pointer or reference to stack variable leaving scope</p> <p>Use of automatic variable as putenv-family function argument</p>
573	Improper following of specification by caller	<p>Missing cipher algorithm</p> <p>Missing cipher data to process</p> <p>Missing cipher final step</p> <p>Missing cipher key</p> <p>Modification of internal buffer returned from nonreentrant standard function</p> <p>Context initialized incorrectly for cryptographic operation</p> <p>Context initialized incorrectly for digest operation</p> <p>Missing parameters for key generation</p> <p>Missing data for encryption, decryption or signing operation</p> <p>Missing peer key</p> <p>Missing private key</p> <p>Missing public key</p> <p>Missing blinding for RSA algorithm</p> <p>Incorrect key for cryptographic algorithm</p> <p>Incompatible padding for RSA algorithm operation</p>

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
587	Assignment of a fixed address to a pointer	Unsafe conversion between pointer and integer Function pointer assigned with absolute address
590	Free of memory not on the heap	Invalid free of pointer
606	Unchecked input for loop condition	Loop bounded with tainted value
628	Function call with incorrectly specified arguments	Bad file access mode or status Copy of overlapping memory Invalid va_list argument Modification of internal buffer returned from nonreentrant standard function Standard function call with incorrect arguments
658	See "Mapping Between CWE-658 or 659 and Polyspace Results" on page 9-100.	
659	See "Mapping Between CWE-658 or 659 and Polyspace Results" on page 9-100.	
663	Use of a non-reentrant function in a concurrent context	Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict) Unsafe standard encryption function Unsafe standard function

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
664	Improper control of a resource through its lifetime	Context initialized incorrectly for cryptographic operation Context initialized incorrectly for digest operation Missing peer key Missing cipher key Missing private key Missing public key Inconsistent cipher operations Missing cipher data to process Incorrect key for cryptographic algorithm Incompatible padding for RSA algorithm operation Missing cipher final step
665	Improper initialization	Call to memset with unintended value Improper array initialization Overlapping assignment Use of memset with size argument zero
666	Operation on resource in wrong phase of lifetime	Incorrect order of network connection operations

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
667	Improper locking	Blocking operation while holding lock Missing unlock Destruction of locked mutex
672	Operation on a resource after expiration or release	Use of previously closed resource Closing a previously closed resource
675	Duplicate operations on resource	Opening previously opened resource
676	Use of potentially dangerous function	Unsafe conversion from string to numerical value Use of dangerous standard function
681	Incorrect conversion between numeric types	Float conversion overflow Precision loss in integer to float conversion
682	Incorrect calculation	Absorption of float operand Float overflow Invalid use of standard library floating point routine Invalid use of standard library integer routine Tainted modulo operand Bitwise operation on negative value Use of plain char type for numerical value
683	Function Call With Incorrect Order of Arguments	Call to memset with unintended value Format string specifiers and arguments mismatch

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
685	Function call with incorrect number of arguments	Declaration mismatch Format string specifiers and arguments mismatch Standard function call with incorrect arguments Too many va_arg calls for current argument list
686	Function call with incorrect argument type	Bad file access mode or status Declaration mismatch Format string specifiers and arguments mismatch Incorrect data type passed to va_arg Standard function call with incorrect arguments Use of automatic variable as putenv-family function argument Writing to const qualified object
687	Function call with incorrectly specified argument value	Copy of overlapping memory Standard function call with incorrect arguments Variable length array with nonpositive size

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
690	Unchecked return value to null pointer dereference	Invalid use of standard library memory routine Use of tainted pointer Null pointer Returned value of a sensitive function not checked Invalid use of standard library string routine Unprotected dynamic memory allocation Tainted NULL or non-null-terminated string Standard function call with incorrect arguments Invalid use of standard library routine
691	Insufficient control flow management	Use of setjmp/longjmp
693	Protection mechanism failure	Nonsecure SSL/TLS protocol
696	Incorrect behavior order	Bad order of dropping privileges
703	Improper check or handling of exceptional conditions	Errno not reset Misuse of errno

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
704	Incorrect type conversion or cast	Character value absorbed into EOF Qualifier removed in conversion Precision loss in integer to float conversion Misuse of sign-extended character value Unreliable cast of pointer Wrong allocated object size for cast
705	Incorrect control flow scoping	Abnormal termination of exit handler
710	Coding standard violation	Bitwise and arithmetic operation on the same data
732	Incorrect permission assignment for critical resource	Vulnerable permission assignments
754	Improper check for unusual or exceptional conditions	Returned value of a sensitive function not checked
755	Improper handling of exceptional conditions	Exception handler hidden by previous handler
758	Reliance on undefined, unspecified, or implementation-defined behavior	Unsafe conversion between pointer and integer Use of plain char type for numerical value Bitwise operation on negative value
762	Mismatched memory management routines	Invalid free of pointer Mismatched alloc/dealloc functions on Windows

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
764	Multiple locks of a critical resource	Double lock
765	Multiple unlocks of a critical resource	Double unlock
767	Access to critical private variable via public method	Return of non const handle to encapsulated data member
770	Allocation of resources without limits or throttling	Tainted size of variable length array
772	Missing release of resource after effective lifetime	Resource leak
780	Use of rsa algorithm without oaep	Weak padding for RSA algorithm Missing padding for RSA algorithm
783	Operator precedence logic error	Possibly unintended evaluation of expression because of operator precedence rules
785	Use of path manipulation function without maximum-sized buffer	Use of path manipulation function without maximum sized buffer checking
786	Access of memory location before start of buffer	Destination buffer underflow in string manipulation
787	Out-of-bounds write	Destination buffer overflow in string manipulation Destination buffer underflow in string manipulation

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
789	Uncontrolled memory allocation	Memory allocation with tainted size Tainted size of variable length array Unprotected dynamic memory allocation
805	Buffer access with incorrect length value	Hard-coded object size used to manipulate memory
822	Untrusted pointer dereference	Tainted NULL or non-null-terminated string Use of tainted pointer
823	Use of out-of-range pointer offset	Pointer access out of bounds Pointer dereference with tainted offset
824	Access of uninitialized pointer	Non-initialized pointer
825	Expired Pointer Dereference	Accessing object with temporary lifetime Deallocation of previously deallocated pointer Environment pointer invalidated by previous operation Missing reset of a freed pointer Pointer or reference to stack variable leaving scope Use of automatic variable as putenv-family function argument Use of previously freed pointer

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
826	Premature release of resource during expected lifetime	Closing a previously closed resource Destruction of locked mutex Use of previously closed resource
828	Signal handler with functionality that is not asynchronous-safe	Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict)
832	Unlock of a resource that is not locked	Missing lock
833	Deadlock	Deadlock
843	Access of resource using incompatible type ('Type confusion')	Unreliable cast of pointer
872	CERT C++ Secure Coding Section 04 - Integers (INT)	Invalid use of standard library integer routine
873	CERT C++ Secure Coding Section 05 - Floating point arithmetic (FLP)	Absorption of float operand Floating point comparison with equality operators Invalid use of standard library floating point routine Float overflow
908	Use of uninitialized resource	Member not initialized in constructor Non-initialized pointer Non-initialized variable

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
910	Use of expired file descriptor	Use of previously closed resource Closing a previously closed resource Standard function call with incorrect arguments
922	Insecure Storage of Sensitive Information	File manipulation after chroot without chdir Umask used with chmod-style arguments Use of non-secure temporary file Vulnerable permission assignments

Mapping Between CWE-658 or 659 and Polyspace Results

CWE-658: Weaknesses in Software Written in C

CWE-658 is a subset of CWE IDs found in C programs that are not common to all languages. See CWE-658.

The following table lists the CWE IDs (version 2.8) from this subset that are addressed by Polyspace Bug Finder, with its corresponding defect checkers.

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
119	Improper restriction of operations within the bounds of a memory buffer	Array access out of bounds Pointer access out of bounds
120	Buffer copy without checking size of input ('Classic buffer overflow')	Invalid use of standard library memory routine Invalid use of standard library string routine Tainted NULL or non-null-terminated string
121	Stack-based buffer overflow	Array access with tainted index Destination buffer overflow in string manipulation
122	Heap-based buffer overflow	Pointer dereference with tainted offset

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
124	Buffer underwrite ('Buffer underflow')	<p>Array access with tainted index</p> <p>Buffer overflow from incorrect string format specifier</p> <p>Destination buffer underflow in string manipulation</p> <p>Pointer dereference with tainted offset</p>
125	Out-of-bounds read	<p>Array access with tainted index</p> <p>Buffer overflow from incorrect string format specifier</p> <p>Destination buffer overflow in string manipulation</p>
126	Buffer over-read	Buffer overflow from incorrect string format specifier
127	Buffer under-read	Buffer overflow from incorrect string format specifier
128	Wrap-around error	<p>Integer constant overflow</p> <p>Tainted sign change conversion</p> <p>Unsigned integer conversion overflow</p> <p>Memory allocation with tainted size</p> <p>Unsigned integer overflow</p> <p>Tainted size of variable length array</p> <p>Unsigned integer constant overflow</p> <p>Integer overflow</p> <p>Integer conversion overflow</p>

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
129	Improper validation of array index	Array access with tainted index Pointer dereference with tainted offset
130	Improper handling of length parameter inconsistency	Mismatch between data length and size
131	Incorrect calculation of buffer size	Pointer access out of bounds Tainted sign change conversion Unsigned integer conversion overflow Memory allocation with tainted size Unsigned integer overflow Array access out of bounds Tainted size of variable length array
134	Uncontrolled format string	Tainted string format
135	Incorrect Calculation of Multi-Byte String Length	Destination buffer overflow in string manipulation Misuse of narrow or wide character string Unreliable cast of pointer
170	Improper null termination	Missing null in string array Misuse of readlink() Tainted NULL or non-null-terminated string

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
188	Reliance on data/ memory layout	Invalid assumptions about memory organization Memory comparison of padding data Memory comparison of strings Missing byte reordering when transferring data Pointer access out of bounds
191	Integer underflow (Wrap or wraparound)	Integer constant overflow Integer conversion overflow Integer overflow Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow
192	Integer coercion error	Tainted sign change conversion Unsigned integer conversion overflow Unsigned integer overflow Sign change integer conversion overflow Integer overflow Integer conversion overflow
194	Unexpected sign extension	Sign change integer conversion overflow Tainted sign change conversion
195	Signed to unsigned conversion error	Sign change integer conversion overflow Tainted sign change conversion

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
196	Unsigned to signed conversion error	Sign change integer conversion overflow
197	Numeric truncation error	Integer conversion overflow Float conversion overflow Unsigned integer conversion overflow
242	Use of inherently dangerous function	Use of dangerous standard function
243	Creation of chroot jail without changing working directory	File manipulation after chroot without chdir
244	Improper clearing of heap memory before release	Sensitive heap memory not cleared before release
362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	File descriptor exposure to child process Opening previously opened resource
364	Signal handler race condition	Shared data access within signal handler Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe
366	Race condition within a thread	Data race Data race including atomic operations Data race through standard library function call

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
375	Returning a mutable object to an untrusted caller	Return of non const handle to encapsulated data member
401	Improper release of memory before removing last reference	Memory leak Thread-specific memory leak
415	Double free	Deallocation of previously deallocated pointer Missing reset of a freed pointer
416	Use after free	Missing reset of a freed pointer Use of previously freed pointer
457	Use of uninitialized variable	Member not initialized in constructor Non-initialized pointer Non-initialized variable
466	Return of pointer value outside of expected range	Array access out of bounds Pointer access out of bounds Unsafe conversion between pointer and integer
467	Use of sizeof() on a pointer type	Possible misuse of sizeof Wrong type used in sizeof
468	Incorrect pointer scaling	Incorrect pointer scaling
469	Use of pointer subtraction to determine size	Subtraction or comparison between pointers to different arrays
474	Use of function with inconsistent implementations	Signal call from within signal handler Use of obsolete standard function

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
476	NULL pointer dereference	Null pointer Tainted NULL or non-null-terminated string
478	Missing default case in switch statement	Missing case for switch condition
479	Signal handler use of a non-reentrant function	Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict)
480	Use of incorrect operator	Invalid use of == (equality) operator Invalid use of = (assignment) operator
481	Assigning instead of comparing	Invalid use of = (assignment) operator
482	Comparing instead of assigning	Invalid use of == (equality) operator
484	Omitted break statement in switch	Missing break of switch case
558	Use of getlogin() in multithreaded application	Unsafe standard function
560	Use of umask() with chmod-style argument	Umask used with chmod-style arguments
562	Return of stack variable address	Pointer or reference to stack variable leaving scope Use of automatic variable as putenv-family function argument

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
587	Assignment of a fixed address to a pointer	Unsafe conversion between pointer and integer Function pointer assigned with absolute address
676	Use of potentially dangerous function	Unsafe conversion from string to numerical value Use of dangerous standard function
685	Function call with incorrect number of arguments	Declaration mismatch Format string specifiers and arguments mismatch Standard function call with incorrect arguments Too many va_arg calls for current argument list

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
690	Unchecked return value to null pointer dereference	Invalid use of standard library memory routine Use of tainted pointer Null pointer Returned value of a sensitive function not checked Invalid use of standard library string routine Unprotected dynamic memory allocation Tainted NULL or non-null-terminated string Standard function call with incorrect arguments Invalid use of standard library routine
704	Incorrect type conversion or cast	Character value absorbed into EOF Qualifier removed in conversion Precision loss in integer to float conversion Misuse of sign-extended character value Unreliable cast of pointer Wrong allocated object size for cast
762	Mismatched memory management routines	Invalid free of pointer Mismatched alloc/dealloc functions on Windows

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
783	Operator precedence logic error	Possibly unintended evaluation of expression because of operator precedence rules
785	Use of path manipulation function without maximum-sized buffer	Use of path manipulation function without maximum sized buffer checking
789	Uncontrolled memory allocation	Memory allocation with tainted size Tainted size of variable length array Unprotected dynamic memory allocation
805	Buffer access with incorrect length value	Hard-coded object size used to manipulate memory
843	Access of resource using incompatible type ('Type confusion')	Unreliable cast of pointer
910	Use of expired file descriptor	Use of previously closed resource Closing a previously closed resource Standard function call with incorrect arguments

CWE-659: Weaknesses in Software Written in C++

CWE-659 is a subset of CWE IDs found in C++ programs that are not common to all languages. See CWE-659.

The following table lists the CWE IDs (version 2.8) from this subset that are addressed by Polyspace Bug Finder, with its corresponding defect checkers.

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
119	Improper restriction of operations within the bounds of a memory buffer	Array access out of bounds Pointer access out of bounds
120	Buffer copy without checking size of input ('Classic buffer overflow')	Invalid use of standard library memory routine Invalid use of standard library string routine Tainted NULL or non-null-terminated string
121	Stack-based buffer overflow	Array access with tainted index Destination buffer overflow in string manipulation
122	Heap-based buffer overflow	Pointer dereference with tainted offset
124	Buffer underwrite ('Buffer underflow')	Array access with tainted index Buffer overflow from incorrect string format specifier Destination buffer underflow in string manipulation Pointer dereference with tainted offset
125	Out-of-bounds read	Array access with tainted index Buffer overflow from incorrect string format specifier Destination buffer overflow in string manipulation
126	Buffer over-read	Buffer overflow from incorrect string format specifier

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
127	Buffer under-read	Buffer overflow from incorrect string format specifier
128	Wrap-around error	Integer constant overflow Tainted sign change conversion Unsigned integer conversion overflow Memory allocation with tainted size Unsigned integer overflow Tainted size of variable length array Unsigned integer constant overflow Integer overflow Integer conversion overflow
129	Improper validation of array index	Array access with tainted index Pointer dereference with tainted offset
130	Improper handling of length parameter inconsistency	Mismatch between data length and size
131	Incorrect calculation of buffer size	Pointer access out of bounds Tainted sign change conversion Unsigned integer conversion overflow Memory allocation with tainted size Unsigned integer overflow Array access out of bounds Tainted size of variable length array

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
134	Uncontrolled format string	Tainted string format
135	Incorrect Calculation of Multi-Byte String Length	Destination buffer overflow in string manipulation Misuse of narrow or wide character string Unreliable cast of pointer
170	Improper null termination	Missing null in string array Misuse of readlink() Tainted NULL or non-null-terminated string
188	Reliance on data/memory layout	Invalid assumptions about memory organization Memory comparison of padding data Memory comparison of strings Missing byte reordering when transferring data Pointer access out of bounds
191	Integer underflow (Wrap or wraparound)	Integer constant overflow Integer conversion overflow Integer overflow Unsigned integer constant overflow Unsigned integer conversion overflow Unsigned integer overflow

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
192	Integer coercion error	Tainted sign change conversion Unsigned integer conversion overflow Unsigned integer overflow Sign change integer conversion overflow Integer overflow Integer conversion overflow
194	Unexpected sign extension	Sign change integer conversion overflow Tainted sign change conversion
195	Signed to unsigned conversion error	Sign change integer conversion overflow Tainted sign change conversion
196	Unsigned to signed conversion error	Sign change integer conversion overflow
197	Numeric truncation error	Integer conversion overflow Float conversion overflow Unsigned integer conversion overflow
242	Use of inherently dangerous function	Use of dangerous standard function
243	Creation of chroot jail without changing working directory	File manipulation after chroot without chdir
244	Improper clearing of heap memory before release	Sensitive heap memory not cleared before release

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
362	Concurrent execution using shared resource with improper synchronization ('Race Condition')	File descriptor exposure to child process Opening previously opened resource
364	Signal handler race condition	Shared data access within signal handler Function called from signal handler not asynchronous-safe (strict) Function called from signal handler not asynchronous-safe
366	Race condition within a thread	Data race Data race including atomic operations Data race through standard library function call
375	Returning a mutable object to an untrusted caller	Return of non const handle to encapsulated data member
401	Improper release of memory before removing last reference	Memory leak Thread-specific memory leak
415	Double free	Deallocation of previously deallocated pointer Missing reset of a freed pointer
416	Use after free	Missing reset of a freed pointer Use of previously freed pointer

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
457	Use of uninitialized variable	Member not initialized in constructor Non-initialized pointer Non-initialized variable
466	Return of pointer value outside of expected range	Array access out of bounds Pointer access out of bounds Unsafe conversion between pointer and integer
467	Use of sizeof() on a pointer type	Possible misuse of sizeof Wrong type used in sizeof
468	Incorrect pointer scaling	Incorrect pointer scaling
469	Use of pointer subtraction to determine size	Subtraction or comparison between pointers to different arrays
476	NULL pointer dereference	Null pointer Tainted NULL or non-null-terminated string
478	Missing default case in switch statement	Missing case for switch condition
479	Signal handler use of a non-reentrant function	Function called from signal handler not asynchronous-safe Function called from signal handler not asynchronous-safe (strict)
480	Use of incorrect operator	Invalid use of == (equality) operator Invalid use of = (assignment) operator
481	Assigning instead of comparing	Invalid use of = (assignment) operator

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
482	Comparing instead of assigning	Invalid use of == (equality) operator
484	Omitted break statement in switch	Missing break of switch case
558	Use of getlogin() in multithreaded application	Unsafe standard function
562	Return of stack variable address	Pointer or reference to stack variable leaving scope Use of automatic variable as putenv-family function argument
587	Assignment of a fixed address to a pointer	Unsafe conversion between pointer and integer Function pointer assigned with absolute address
676	Use of potentially dangerous function	Unsafe conversion from string to numerical value Use of dangerous standard function

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
690	Unchecked return value to null pointer dereference	Invalid use of standard library memory routine Use of tainted pointer Null pointer Returned value of a sensitive function not checked Invalid use of standard library string routine Unprotected dynamic memory allocation Tainted NULL or non-null-terminated string Standard function call with incorrect arguments Invalid use of standard library routine
704	Incorrect type conversion or cast	Character value absorbed into EOF Qualifier removed in conversion Precision loss in integer to float conversion Misuse of sign-extended character value Unreliable cast of pointer Wrong allocated object size for cast
762	Mismatched memory management routines	Invalid free of pointer Mismatched alloc/dealloc functions on Windows

CWE ID	CWE ID Description	Polyspace Bug Finder Defect Checker
767	Access to critical private variable via public method	Return of non const handle to encapsulated data member
783	Operator precedence logic error	Possibly unintended evaluation of expression because of operator precedence rules
785	Use of path manipulation function without maximum-sized buffer	Use of path manipulation function without maximum sized buffer checking
789	Uncontrolled memory allocation	Memory allocation with tainted size Tainted size of variable length array Unprotected dynamic memory allocation
805	Buffer access with incorrect length value	Hard-coded object size used to manipulate memory
843	Access of resource using incompatible type ('Type confusion')	Unreliable cast of pointer
910	Use of expired file descriptor	Use of previously closed resource Closing a previously closed resource Standard function call with incorrect arguments

See Also

More About


- “CWE Coding Standard and Polyspace Results” on page 9-66

Configure Comment Import from Previous Results

- “Import Comments from Previous Polyspace Analysis” on page 10-2
- “Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results” on page 10-6

Import Comments from Previous Polyspace Analysis

After you have reviewed analysis results, you can reuse your review comments for subsequent analyses. If you add comments to your results file, they carry over to the next analysis on the same project. If you add comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not. You can also hide results using code annotations. For more information on commenting, see Polyspace Bug Finder Access documentation.

This topic shows how to import comments from one result file to another. Importing comments saves you from reviewing already justified results. For instance, after you import comments, on the **Results List** pane (user interface of desktop products), clicking the  icon skips justified results. Using this icon, you can browse through unreviewed results. You can also filter the justified checks from display.

Automatic Comment Import from Last Analysis

By default, in the Polyspace user interface (desktop products only), comments are imported automatically from the most recent analysis on the project module. You can disable this default behavior.

- 1 Select **Tools > Preferences**, which opens the Polyspace Preferences dialog box.
- 2 Select the **Project and Results Folder** tab.
- 3 Under **Import Comments**, clear **Automatically import comments from last verification**.
- 4 Click **OK**.

If you upload results to the Polyspace Access web interface, comments from the last run of the same project are applied to the current run. You cannot disable the automatic comment import.

If you run analysis at the command line (and do not upload results to the Polyspace Access web interface), you have to explicitly import comments from another set of results. See “Command Line” on page 10-3.

Import Comments from Another Analysis Result

You can import comments directly from another Polyspace result to the current result.

If a result is found in both a Bug Finder and Code Prover analysis, you can comment on the Bug Finder result and import the comment to Code Prover. For instance, most coding rule checkers are common to Bug Finder and Code Prover. You can add comments to coding rule violations in Bug Finder and import the comments to the same violations in Code Prover.

User Interface (Desktop Products Only)

To import comments from another set of results:

- 1 Open the current analysis results.
- 2 Select **Tools > Import Comments**.
- 3 Navigate to the folder containing your previous results.
- 4 Select the other results file (with extension `.psbf` or `.pscp`) and then click **Open**.

The review comments from the previous results are imported into the current results.

Command Line

Use the option `-import-comments` during analysis to import comments from a previous verification.

To import comments from multiple results, use the `polyspace-comments-import` command.

Comment Import Algorithm

You can directly import review information from another set of results into the current results. However, it is possible that part of your review information is not imported to a subsequent analysis because:

- You have changed your source code so that the line with a previous result is not exactly identical to the line in the current run.

The comment import tool accounts for additional code that simply shifts an existing line. For instance, the tool recognizes that line 10 in Run 1 and line 12 in Run 2 have the same statement. If a division by zero occurs on line 10 in Run 1 and you have not fixed the issue in Run 2, the result along with associated review information are imported to line 12 in Run 2.

- Run 1:

```
10 baseLine = min/numRecipients;
11
12
```

- Run 2:

```
10 /* Calculate a baseline per recipient
11    based on minimum available resources */
12 baseLine = min/numRecipients;
```

However, if you change the line content itself, for instance, change `numRecipient` to `numReceiver`, the result and review information are not imported.

- You have changed your source code so that the Code Prover result color has changed.
- You entered new review information for the same result.

View Imported Comments That Do Not Apply

In the Polyspace user interface (desktop products only), the Import Checks and Comments Report highlights differences between two analysis results. When you import comments from a previous analysis, you can see this report. If you have closed the report after an import, to review the report again:

- Select **Window > Show/Hide View > Import Comments Report**.

The Import Checks and Comments Report opens, highlighting differences in the two results.

The table below contains a list of checks where:

- The check color has changed. How the review information is imported depends on the color, classification and status of the check. For details, see [Importing and Exporting Review Comments](#) in the user guide.
- The check is no longer found in the code or is already justified in the current results. The review information has not been imported.

Note: If you changed your source code or Polyspace configuration, the imported justifications may not be fully applicable to the new results.

File	Function	Line	Col	Check	Import details	Justified	Classification	Status	Comment
example.c	example.c	11		2019f	Check color has changed from Green to Orange	<input checked="" type="checkbox"/>	Not a defect	No action planned	[This might overflow...

2 Review the differences between the two results.

Your review information can differ between two results because of the following reasons:

- In Code Prover, if the check color changes, Polyspace imports the **Comment** field but not the **Status** field. In addition, Polyspace imports the **Severity** and **Justified** fields depending on the color change.

Color Change	Severity	Justified
Orange or red to green	Not imported	Imported
Gray to green	Not imported	Imported, if the Severity was set to High, Medium or Low.
Red to orange or vice versa	Imported	Imported
Green to red/orange/gray	Not imported	Not imported

- If a result no longer appears in the code, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review comments from the previous result.
- If you have already entered different review comments for the same check, Polyspace highlights only the change in the Import Checks and Comments Report. It does not import review comments from the previous result.

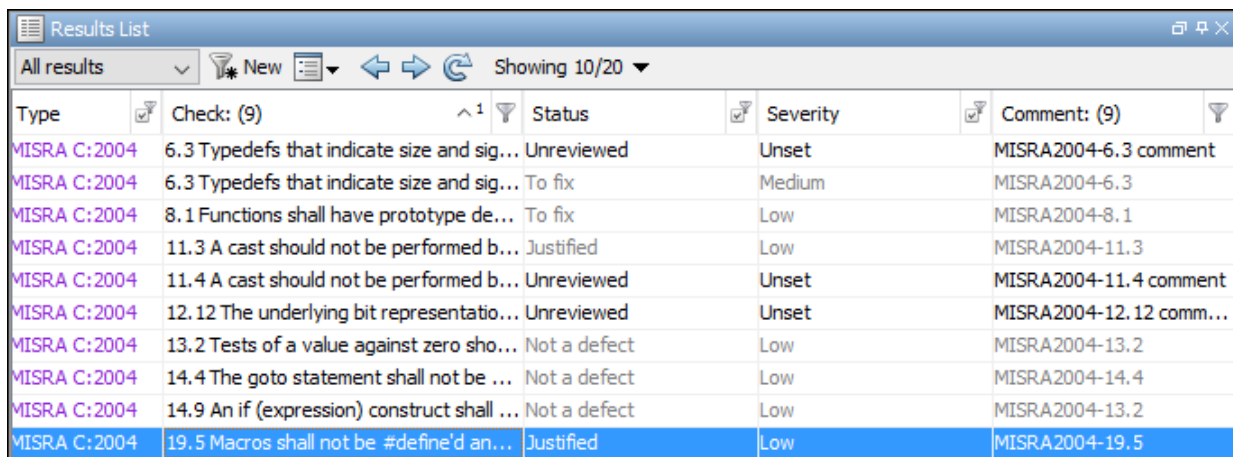
See Also

-import-comments | polyspace-comments-import

Import Existing MISRA C: 2004 Justifications to MISRA C: 2012 Results

When you check your code for MISRA C: 2012 violations, Polyspace imports justifications of MISRA C: 2004 violations from previous analyses (if they exist). You can upgrade from checking of MISRA C: 2004 rules to MISRA C: 2012 rules while retaining your justifications. For general rules on comment import, see “Import Comments from Previous Polyspace Analysis” on page 10-2.

The software maps MISRA C: 2004 **Status**, **Severity**, and **Comment** values that you added through the user interface or code annotations to the corresponding MISRA C: 2012 results, if the results exist. For more information about mapping, consult addendum one of the MISRA C: 2012 publication.



Type	Check: (9)	Status	Severity	Comment: (9)
MISRA C:2004	6.3 Typedefs that indicate size and sig...	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C:2004	6.3 Typedefs that indicate size and sig...	To fix	Medium	MISRA2004-6.3
MISRA C:2004	8.1 Functions shall have prototype de...	To fix	Low	MISRA2004-8.1
MISRA C:2004	11.3 A cast should not be performed b...	Justified	Low	MISRA2004-11.3
MISRA C:2004	11.4 A cast should not be performed b...	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C:2004	12.12 The underlying bit representatio...	Unreviewed	Unset	MISRA2004-12.12 comm...
MISRA C:2004	13.2 Tests of a value against zero sho...	Not a defect	Low	MISRA2004-13.2
MISRA C:2004	14.4 The goto statement shall not be ...	Not a defect	Low	MISRA2004-14.4
MISRA C:2004	14.9 An if (expression) construct shall ...	Not a defect	Low	MISRA2004-13.2
MISRA C:2004	19.5 Macros shall not be #define'd an...	Justified	Low	MISRA2004-19.5

If you are transitioning from MISRA C: 2004 to MISRA C: 2012, you do not have to review results that you have already justified.

Type	Check	Status	Severity	Comment: (7)
MISRA C:2012	Dir 4.6 typedefs that indicate size and...	Unreviewed	Unset	MISRA2004-6.3 comment
MISRA C:2012	Dir 4.6 typedefs that indicate size and...	To fix	Medium	MISRA2004-6.3
MISRA C:2012	8.4 A compatible declaration shall be v...	To fix	Low	MISRA2004-8.1
MISRA C:2012	11.3 A cast shall not be performed bet...	Unreviewed	Unset	MISRA2004-11.4 comment
MISRA C:2012	11.4 A conversion should not be perfo...	Justified	Low	MISRA2004-11.3
MISRA C:2012	14.4 The controlling expression of an i...	Not a defect	Low	MISRA2004-13.2
MISRA C:2012	15.1 The goto statement should not b...	Not a defect	Low	MISRA2004-14.4
MISRA C:2012	15.6 The body of an iteration-stateme...	Not a defect	Low	MISRA2004-13.2

Mapping Multiple MISRA C: 2004 Annotations to the Same MISRA C: 2012 Result

When you justify MISRA C: 2004 violations by using code block syntax or multiple line annotation syntax, and multiple violations map to the same MISRA C: 2012 rule, Polyspace does not import each result justification. Instead, the software imports only one set of **Status**, **Severity**, and **Comment** values and applies these values to all the instances of that particular MISRA C: 2012 rule violation.

For example, suppose that you analyze your code and find violations of MISRA C: 2004 rules 16.3 and 16.5. You can justify these results by using the annotation syntax where you enter a different status and explanatory comment for each rule.

```
//polyspace-begin misra2004:16.3 [Status 1] "Explanatory comment 1"
//polyspace-begin misra2004:16.5 [Status 2] "Explanatory comment 2"

code block start;
/* This block of code contains violations of
MISRA C:2004 rules 16.3 and 16.5 */
code block end;

//polyspace-end misra2004:16.3
//polyspace-end misra2004:16.5
```

The previous violations map to MISRA C: 2012 rule 8.2. When you check your annotated code against MISRA C: 2012 rules, Polyspace imports only the first line of annotations (for rule 16.3) and applies it to all rule 8.2 results. The second line of annotations for rule 16.5 is ignored. In the **Results List** pane, all violations of rule 8.2 have the **Status** column set to **Status 1** and the **Comment** column set to **"Explanatory comment 1"**.

Note The **Output Summary** pane displays a warning message for every result where the imported annotation conflicts with the original annotation. After you import your MISRA C: 2004 annotations, check that a justified status has not been assigned to results you intend to investigate or fix.

See Also

Check MISRA C:2004 (-misra2) | Check MISRA C:2012 (-misra3)

Troubleshooting in Polyspace Bug Finder Server

- “License Error -4,0” on page 11-3
- “Read Error Information When Polyspace Analysis Stops” on page 11-4
- “Contact Technical Support” on page 11-5
- “Compiler Not Supported for Project Creation from Build Systems” on page 11-8
- “Slow Build Process When Polyspace Traces the Build” on page 11-18
- “Check if Polyspace Supports Build Scripts” on page 11-19
- “Troubleshooting Project Creation from MinGW Build” on page 11-21
- “Troubleshooting Project Creation from Visual Studio Build” on page 11-22
- “Error Processing Macro with Semicolon in Build System” on page 11-24
- “Polyspace Cannot Find the Server” on page 11-25
- “Job Manager Cannot Write to Database” on page 11-26
- “Undefined Identifier Error” on page 11-28
- “Unknown Function Prototype Error” on page 11-32
- “Error Related to #error Directive” on page 11-34
- “Large Object Error” on page 11-36
- “Errors Related to Generic Compiler” on page 11-39
- “Errors Related to Keil or IAR Compiler” on page 11-41
- “Errors Related to Diab Compiler” on page 11-42
- “Errors Related to Green Hills Compiler” on page 11-45
- “Errors Related to TASKING Compiler” on page 11-47
- “Errors from Conflicts with Polyspace Header Files” on page 11-49
- “Errors from Assertion or Memory Allocation Functions” on page 11-51
- “Error from Special Characters” on page 11-52
- “Errors from In-Class Initialization (C++)” on page 11-53

- “Errors from Double Declarations of Standard Template Library Functions (C++)” on page 11-54
- “Errors Related to GNU Compiler” on page 11-55
- “Errors Related to Visual Compilers” on page 11-56
- “Error or Slow Runs from Disk Defragmentation and Anti-virus Software” on page 11-58
- “SQLite I/O Error” on page 11-60

License Error -4,0

Issue

When you try to run Polyspace, you get this error message:

```
License Error -4,0
```

Possible Cause: Another Polyspace Instance Running

You can open multiple instances of Polyspace, but you can only run one code analysis at a time.

If you try to run Polyspace processes from multiple windows, you will get a License Error -4,0 error.

Solution

Only run one analysis at a time, including any command-line or plugin analyses.

Possible Cause: Prior Polyspace Run in Simulink or MATLAB Coder

If you run Polyspace on generated code in the Simulink user interface or in the MATLAB Coder app, you can get a license error if you try to run a subsequent analysis in the Polyspace user interface. You get the error even if the previous run is over.

Solution

Run the subsequent analysis using the method that you used before, that is, in the Simulink user interface or MATLAB Coder app.

If you want to run the analysis in the Polyspace user interface, close Simulink or MATLAB Coder and then rerun the analysis.

Read Error Information When Polyspace Analysis Stops

When you run a Polyspace analysis on your C/C++ code, if one or more of your files fail to compile, the analysis continues with the remaining files. You can choose to stop the analysis on compilation errors using the option `Stop analysis if a file does not compile (-stop-if-compile-error)`.

However, it is more convenient to let the analysis complete and capture all compilation errors. In a continuous integration process, you can send a notification to the build engineer with a list of compilation errors.

The compilation errors are displayed in the analysis log in addition to the options used and the various stages of analysis. The analysis log is a text file generated in your results folder and titled `Polyspace_version_project_date_time.txt`. The lines that indicate errors begin with the `Error:` string and the lines that indicate warnings begin with the `Warning:` string. Find these lines and extract them to another text file for easier scanning.

Contact Technical Support

To contact MathWorks Technical Support, use this page. You need a MathWorks Account login and password. For faster turnaround with an issue in Polyspace, besides the required system information, provide appropriate code that reproduces the issue or the verification log file.

Provide System Information

When you enter a support request, provide the following system information:

- Hardware configuration
- Operating system
- Polyspace and MATLAB license numbers
- Specific version numbers for Polyspace products
- Installed Bug Report patches

To obtain your configuration information, do one of the following:

- In the Polyspace user interface, select **Help > About**.
- At the command line, run the following command, replacing *polyspaceroot* with your Polyspace installation folder:
 - UNIX — `polyspaceroot/polyspace/bin/polyspace-code-prover -ver`
 - Windows — `polyspaceroot\polyspace\bin\polyspace-code-prover -ver`

Provide Information About the Issue

Depending on the issue, provide appropriate artifacts to help Technical Support understand and reproduce the issue.

Compilation Errors

If you face compilation issues with your project, see “Troubleshoot Compilation Errors”. If you are still having issues, contact technical support with the following information:

- The analysis log.

The analysis log is a text file generated in your results folder and titled `Polyspace_version_project_date_time.txt`. It contains the error message, the options used for the analysis and other relevant information.

- The source files related to the compilation error, if possible.

If you cannot provide the source files:

- Try to provide a screenshot of the source code section that causes the compilation issue.
- Try to reproduce the issue with a different code. Provide that code to technical support.

Errors in Project Creation from Build Systems

If you face errors in creating a project from your build system, see “Troubleshoot Project Creation”.

If you are still having issues, contact technical support with debug information. To provide the debug information:

- 1 Run `polyspace-configure` at the command line with the options `-debug` and `-build-trace`. For instance:

```
polyspace-configure options -debug -build-trace build.txt  
                    buildCommand 2>&1 > out.log
```

Here:

- *options* is the list of `polyspace-configure` options that you typically use.
- *buildCommand* is the build command that you use, for instance, `make`.

Make sure that you do not use the option `-verbose` or `-silent` after `-debug`.

- 2 Provide the files `build.txt` and `out.log`.

Verification Result

If you are having trouble understanding a result, see “Polyspace Bug Finder Results” (Polyspace Bug Finder Access).

If you are still having trouble understanding the result, contact technical support with the following information:

- The analysis log.

The analysis log is a text file generated in your results folder and titled `Polyspace_version_project_date_time.txt`. It contains the options used for the analysis and other relevant information.

- The source files related to the result if possible.

If you cannot provide the source files:

- Try provide a screenshot of the relevant source code from the **Source** pane on the Polyspace user interface.
- Try to reproduce the problem with a different code. Provide that code to technical support.

Compiler Not Supported for Project Creation from Build Systems

Issue

Your compiler is not supported for automatic project creation from build commands.

Cause

For automatic project creation from your build system, your compiler configuration must be available to Polyspace. Polyspace provides a compiler configuration file only for certain compilers.

For information on which compilers are supported, see “Requirements for Project Creation from Build Systems” on page 5-21.

Solution

To enable automatic project creation for an unsupported compiler, you can write your own compiler configuration file.

- 1 Copy one of the existing configuration files from *polyspaceroot\polyspace\configure\compiler_configuration*. Select the configuration that most closely corresponds to your compiler using the .
- 2 Save the file as *my_compiler.xml*. *my_compiler* can be a name that helps you identify the file.

To edit the file, save it outside the installation folder. After you have finished editing, you must copy the file back to *polyspaceroot\polyspace\configure\compiler_configuration*.

- 3 Edit the contents of the file to represent your compiler. Replace the entries between the XML elements with appropriate content.
- 4 After saving the edited XML file to *polyspaceroot\polyspace\configure\compiler_configuration*, create a project automatically using your build command.

If you see errors, for instance, compilation errors, contact MathWorks Technical Support. After tracing your build command, the software compiles certain files using

the compiler specifications detected from your configuration file and build command. Compilation errors might indicate issues in the configuration file.

Tip To quickly see if your compiler configuration file works, run the automatic project setup on a sample build that does not take much time to complete. After you have set up a project with your compiler configuration file, you can use this file for larger builds.

Elements of Compiler Configuration File

The following table lists the XML elements in the compiler configuration file file with a description of what the content within the element represents.

XML Element	Content Description	Content Example for GNU C Compiler
<pre><compiler_names><name> ... </name><compiler_names></pre>	<p>Name of the compiler executable. This executable transforms your .c files into object files. You can add several binary names, each in a separate <code><name>...</name></code> element. The software checks for each of the provided names and uses the compiler name for which it finds a match.</p> <p>You must not specify the linker binary inside the <code><name>...</name></code> elements.</p> <p>If the name that you specify is present in an existing compiler configuration file, an error occurs. To avoid the error, use the additional option <code>-compiler-config my_compiler.xml</code> when tracing the build so that the software explicitly uses your compiler configuration file.</p>	<ul style="list-style-type: none"> • gcc • gpp

XML Element	Content Description	Content Example for GNU C Compiler
<pre><include_options><opt> ... </opt></include_options></pre>	<p>The option that you use with your compiler to specify include folders.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-I</p>
<pre><system_include_options> <opt> ... </opt> </system_include_options></pre>	<p>The option that you use with your compiler to specify system headers.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-isystem</p>
<pre><preinclude_options><opt> ... </opt></preinclude_options></pre>	<p>The option that you use with your compiler to force inclusion of a file in the compiled object.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	<p>-include</p>

XML Element	Content Description	Content Example for GNU C Compiler
<code><define_options><opt> ... </opt></define_options></code>	<p>The option that you use with your compiler to predefine a macro.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	-D
<code><undefine_options><opt> ... </opt></undefine_options></code>	<p>The option that you use with your compiler to undo any previous definition of a macro.</p> <p>To specify options where the argument immediately follows the option, use an <code>isPrefix</code> attribute for <code>opt</code> and set it to <code>true</code>.</p>	-U

XML Element	Content Description	Content Example for GNU C Compiler
<pre><semantic_options><opt> ... </opt></semantic_options></pre>	<p>The options that you use to modify the compiler behavior. These options specify the language settings to which the code must conform.</p> <p>You can use the <code>isPrefix</code> attribute to specify multiple options that have the same prefix and the <code>numArgs</code> attribute to specify options with multiple arguments. For instance:</p> <ul style="list-style-type: none"> • Instead of <pre><opt>-m32</opt> <opt>-m64</opt></pre> <p>You can write <code><opt isPrefix="true">-m</opt></code>.</p> • Instead of <pre><opt>-std=c90</opt> <opt>-std=c99</opt></pre> <p>You can write <code><opt numArgs="1">-std</opt></code>. If your makefile uses <code>-std c90</code> instead of <code>-std=c90</code>, this notation also supports that usage.</p> 	<ul style="list-style-type: none"> • <code>-ansi</code> • <code>-std =C90</code> • <code>-std =c++11</code> • <code>-fun signed -char</code>

XML Element	Content Description	Content Example for GNU C Compiler
<pre><dialect> ... </dialect></pre>	<p>The Polyspace dialect that corresponds to or closely matches your compiler dialect. The content of this element directly translates to the option Dialect in your Polyspace project or options file.</p> <p>For the complete list of compilers available, see Compiler (-compiler).</p>	<p>gnu4.7</p>
<pre><preprocess_options_list> <opt> ... </opt> </preprocess_options_list></pre>	<p>The options that specify how your compiler generates a preprocessed file.</p> <p>You can use the macro \$ (OUTPUT_FILE) if your compiler does not allow sending the preprocessed file to the standard output. Instead it defines the preprocessed file internally.</p>	<p>-E</p> <p>For an example of the \$ (OUTPUT_FILE) macro, see the existing compiler configuration file <code>cl2000.xml</code>.</p>

XML Element	Content Description	Content Example for GNU C Compiler
<pre><preprocessed_output_file> ... </preprocessed_output_file></pre>	<p>The name of file where the preprocessed output is stored.</p> <p>You can use the following macros when the name of the preprocessed output file is adapted from the source file:</p> <ul style="list-style-type: none"> • <code>\$(SOURCE_FILE)</code>: Source file name • <code>\$(SOURCE_FILE_EXT)</code>: Source file extension • <code>\$(SOURCE_FILE_NO_EXT)</code>: Source file name without extension <p>For instance, use <code>\$(SOURCE_FILE_NO_EXT).pre</code> when the preprocessor file name has the same name as the source file, but with extension <code>.pre</code>.</p>	<p>For an example of this element, see the existing compiler configuration file <code>xc8.xml</code>.</p>
<pre><src_extensions><ext> ... </ext></src_extensions></pre>	<p>The file extensions for source files.</p>	<ul style="list-style-type: none"> • <code>c</code> • <code>cpp</code> • <code>c++</code>
<pre><obj_extensions><ext> ... </ext></obj_extensions></pre>	<p>The file extensions for object files.</p>	
<pre><precompiled_header_extensions> ... </precompiled_header_extensions></pre>	<p>The file extensions for precompiled headers (if available).</p>	

XML Element	Content Description	Content Example for GNU C Compiler
<pre><polyspace_extra_options_list> <opt> ... </opt> <opt> ... </opt> </polyspace_extra_options_list></pre>	<p>Additional options that are used for the subsequent analysis.</p> <p>For instance, to avoid compilation errors in the subsequent analysis due to non-ANSI extension keywords, enter <code>-D keyword=value</code>, for example:</p> <pre><polyspace_extra_options_list> <opt>-D MACRO1</opt> <opt>-D MACRO2=VALUE</opt> </polyspace_extra_options_list></pre> <p>For more information, see Preprocessor definitions (-D).</p>	

Mapping Between Existing Configuration Files and Compiler Names

Select the configuration file in `polyspaceroot\polyspace\configure\compiler_configuration\` that most closely resembles the configuration of your compiler. Use the following table to map compilers to their configuration files.

Compiler Name	Vendor	XML File
ARM®	ARM Keil	armcc.xml armclang.xml
Visual C++	Microsoft	cl.xml
Clang	Not applicable	clang.xml
CodeWarrior	NXP	cw_ppc.xml cw_s12z.xml
cx6808	Cosmic	cx6808.xml

Compiler Name	Vendor	XML File
Diab	Wind River	diab.xml
gcc	Not applicable	gcc.xml
Green Hills	Green Hills Software	ghs_arm.xml
		ghs_arm64.xml
		ghs_i386.xml
		ghs_ppc.xml
		ghs_rh850.xml
		ghs_tricore.xml
IAR Embedded Workbench	IAR	iar.xml
		iar-arm.xml
		iar-avr.xml
		iar-msp430.xml
		iar-rh850.xml
		iar-rl78.xml
Renesas	Renesas	renesas-rh850.xml
		renesas-rl78.xml
		renesas-rx.xml
TASKING®	Altium	tasking.xml
		tasking-166.xml
		tasking-850.xml
		tasking-arm.xml
Tiny C	Not applicable	tcc.xml
TM320 and its derivatives	Texas Instruments	ti_arm.xml
		ti_c28x.xml
		ti_c6000.xml
		ti_msp430.xml
xc8 (PIC)	Microchip	xc8.xml

Slow Build Process When Polyspace Traces the Build

Issue

In some cases, your build process can run slower when Polyspace traces the build.

Cause

Polyspace caches information in files stored in the system temporary folder, such as `C:\Users\User_Name\AppData\Local\Temp`, in Windows. Your build can take a long time to perform read/write operations to this folder. Therefore, the overall build process is slow.

Solution

You can work around the slow build process by changing the location where Polyspace stores cache information. For instance, you can use a cache path local to the drive from which you run build tracing. To create and use a local folder `ps_cache` for storing cache information, use the advanced option `-cache-path ./ps_cache`.

- If you trace your build from the Polyspace user interface, enter this flag in the field **Add advanced configure options**.
- If you trace your build from the DOS/ UNIX or MATLAB command line, use this flag with the `polyspace-configure` command.

For more information, see `polyspace-configure`.

Check if Polyspace Supports Build Scripts

Issue

This topic is relevant only if you are creating a Polyspace project in Windows from your build scripts.

When Polyspace traces your build script in a Windows console application other than `cmd.exe`, the command fails. However, the build command by itself executes to completion.

For instance, your build script executes to completion from the Cygwin shell. However, when Polyspace traces the build, the build script throws an error.

Possible Cause

When you launch a Windows console application, your environment variables are appropriately set. Alternate console applications such as the Cygwin shell can set your environment differently from `cmd.exe`.

Polyspace attempts to trace your build script with the assumption that the script runs to completion in `cmd.exe`. Therefore, even if your script runs to completion in the alternate console application, when Polyspace traces the build, the script can fail.

Solution

Make sure that your build script executes to completion in the `cmd.exe` interface. If the build executes successfully, create a wrapper `.bat` file around your script and trace this file.

For instance, before you trace a build command that executes to completion in the Cygwin shell, do one of the following:

- Launch the Cygwin shell from `cmd.exe` and then run your build script. For instance, if you use a script `build.sh` to build your code, enter the following command at the DOS command line:

```
cmd.exe /C "C:\cygwin64\bin\bash.exe" -c build.sh
```
- Find the full path to your build script and then run this script from `cmd.exe`.

For instance, enter the following command at the DOS command line:

```
cmd.exe /C path_to_script
```

path_to_script is the full path to your build script. For instance, C:\my_scripts\build.sh.

If the steps do not execute to completion, Polyspace cannot trace your build.

If the steps complete successfully, trace the build command after launching it from `cmd.exe`. For instance, on the command-line, do the following to create a Polyspace options file.

- 1 Enter your build commands in a `.bat` file.

```
rem @echo off  
cmd.exe /C "C:\cygwin64\bin\bash.exe" -c build.sh
```

Name the file, for instance, `launching.bat`.

- 2 Trace the build commands in the `.bat` file and create a Polyspace options file.

```
"C:\Program Files\MATLAB\R2017b\polyspace\bin\polyspace-configure.exe"  
-output-options-file myOptions.txt launching.bat
```

You can now run `polyspace-bug-finder-server` on the options file.

Troubleshooting Project Creation from MinGW Build

Issue

You create a project from a MinGW build, but get an error when running an analysis on the project. The error message comes from using one of these keywords: `__declspec`, `__cdecl`, `__fastcall`, `__thiscall` or `__stdcall`.

Cause

When you create a project from a MinGW build, the project uses a GNU compiler. Polyspace does not recognize these keywords for the GNU compilers.

Solution

Replace these keywords with equivalent keywords just for the purposes of analysis.

Before analysis, for the option Preprocessor definitions (-D), enter:

- `__declspec(x)=__attribute__((x))`
- `__cdecl=__attribute__((__cdecl__))`
- `__fastcall=__attribute__((__fastcall__))`
- `__thiscall=__attribute__((__thiscall__))`
- `__stdcall=__attribute__((__stdcall__))`

If you are running Polyspace on the command line in a UNIX shell, add double quotes around the -D option. For instance, use:

```
"-D __cdecl=__attribute__((__cdecl__))"
```

Troubleshooting Project Creation from Visual Studio Build

In this section...
“Cannot Create Project from Visual Studio Build” on page 11-22
“Compilation Error After Creating Project from Visual Studio Build” on page 11-22

Cannot Create Project from Visual Studio Build

If you are trying to import a Visual Studio 2010 or Visual Studio 2012 project and `polyspace-configure` does not work properly, do the following:

- 1 Stop the `MSBuild.exe` process.
- 2 Set the environment variable `MSBUILDDISABLENODEREUSE` to 1.
- 3 Specify `MSBuild.exe` with the `/nodereuse:false` option.
- 4 Restart the Polyspace configuration tool:

```
polyspace-configure.exe -lang cpp <MSVS path>/msbuild sample.sln
```

Compilation Error After Creating Project from Visual Studio Build

Issue

After you automatically set up your project from a Visual Studio 2010 build, you face compilation errors.

Possible Cause

By default, Polyspace assigns the latest version of the compiler, `visual12.0` to your project. This assignment can cause compilation errors. For more information on the option to specify compilers, see `Compiler (-compiler)`.

Solution

To avoid the errors, do one of the following:

- After automatic project setup:
 - 1 Open the project in the user interface. On the **Configuration** pane, select **Target & Compiler**.
 - 2 Check the setting for **Compiler**. If it is set to `visual12.0`, change it to `visual10`.

Note If you are creating an options file from your Visual Studio 2010 build, check the `-compiler` argument. If it is set to `visual12.0`, change it to `visual10`.

- Before automatic project setup:
 - 1 Open the file `cl.xml` in `polyspaceroot\polyspace\configure\compiler_configuration\` where `polyspaceroot` is your Polyspace installation folder such as `C:\Program Files\Polyspace\R2019a`.
 - 2 Change the line


```
<dialect>visual12.0</dialect>
```

to

```
<dialect>visual10</dialect>
```
 - 3 Create your project or options file. The compiler is already assigned to `visual10`.

Error Processing Macro with Semicolon in Build System

Issue

You see this error when creating a Polyspace project or options file from your build system:

```
Could not process macro containing a semicolon
```

Cause

Some options in your build system use semicolons in the replacement list of a macro. Automatic project creation from build systems does not support this usage. For instance, a macro OK with this replacement list can cause issues:

```
{printf("OK");flush();}
```

The use of semicolons in replacement lists is not supported because a Polyspace project or options file created from your build system itself uses semicolon separators to separate macro definitions. For details on the Polyspace options that define macros, see:

- `Preprocessor definitions (-D)`: This option defines macros.
- `-options-for-sources`: This option collects several macro definitions, separated by semicolon.

Solution

Define the macro in a header file instead of in the build system. For instance, define the macro OK like this in a header file:

```
#ifndef OK_DEFINED
#define OK_DEFINED
#define OK {printf("OK");flush();}
#endif
```

Provide the header file only for the purposes of Polyspace analysis using the option `Include (-include)`.

Polyspace Cannot Find the Server

Message

```
Error: Cannot instantiate Polyspace cluster
| Check the -scheduler option validity or your default cluster profile
| Could not contact an MJS lookup service using the host computer_name.
  The hostname, computer_name, could not be resolved.
```

Possible Cause

Polyspace uses information provided in the preferences of a Polyspace desktop product to locate the server. If this information is incorrect, the software cannot locate the server.

Solution

Open the user interface of the Polyspace desktop product. Check if the server information provided is correct.

- 1 Select **Tools > Preferences**.
- 2 Select the **Server Configuration** tab. Check your server information.

For instance, the entry in **Job scheduler host name** must match the host name of the computer that forms the head node of the MATLAB Parallel Server cluster. For more information, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Job Manager Cannot Write to Database

Message

Unable to write data to the job manager database

Possible Cause

If the computer that forms the head node of the MATLAB Parallel Server cluster cannot send data to the client computer, you see this error. The most likely reasons for the remote computer being unable to connect to the client computer are:

- Firewalls do not allow traffic from the MATLAB Job Scheduler to the client.
- The MATLAB Job Scheduler cannot resolve the short hostname of the client computer.

Workaround

Add localhost IP to configuration.

- 1 In the user interface of the Polyspace desktop products, select **Tools > Preferences**.
- 2 On the **Server Configuration** tab, in the **Localhost IP address** field, enter the IP address of your local computer.

To retrieve your IP address:

- Windows
 - 1 Open **Control Panel > Network and Sharing Center**.
 - 2 Select your active network.
 - 3 In the Status window, click **Details**. Your IP address is listed under **IPv4 address**.
- Linux — Run the `ifconfig` command and find the `inet addr` corresponding to your network connection.
- Mac — Open **System Preferences > Network**.

See Also

Related Examples

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Connection Problems Between the Client and MATLAB Job Scheduler” (Parallel Computing Toolbox)

Undefined Identifier Error

Issue

Polyspace verification fails during the compilation phase with a message about undefined identifiers.

The message indicates that Polyspace cannot find a variable definition. Therefore, it cannot identify the variable type.

Possible Cause: Missing Files

The source code you provided does not contain the variable definition. For instance, the variable is defined in an include file that Polyspace cannot find.

If you `#include-d` the include file in your source code but did not add it to your Polyspace project, you see a previous warning:

```
Warning: could not find include file "my_include.h"
```

Solution

If the variable definition occurs in an include file, add the folder that contains the include file.

- In the user interface of the Polyspace desktop products, add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Bug Finder).

- At the command line, use the flag `-I` with the `polyspace-bug-finder-server` command.

For more information, see `-I`.

Possible Cause: Unrecognized Keyword

The variable represents a keyword that your compiler recognizes but is not part of the ANSI C standard. Therefore, Polyspace does not recognize it.

For instance, some compilers interpret `__SP` as a reference to the stack pointer.

Solution

If the variable represents a keyword that Polyspace does not recognize, replace or remove the keyword from your source code or preprocessed code.

If you remove or replace the keyword from the preprocessed code, you can avoid the compilation error while keeping your source code intact. You can do one of the following:

- Replace or remove each individual unknown keyword using an analysis option. Replace the compiler-specific keyword with an equivalent keyword from the ANSI C Standard.

For information on the analysis option, see `Preprocessor definitions (-D)`.

- Declare the unknown keywords in a separate header file using `#define` directives. Specify that header file using an analysis option.

For information on the analysis option, see `Include (-include)`. For a sample header file, see “Gather Compilation Options Efficiently” on page 5-30.

Possible Cause: Declaration Embedded in #ifdef Statements

The variable is declared in a branch of an `#ifdef macro_name` preprocessor directive. For instance, the declaration of a variable `max_power` occurs as follows:

```
#ifdef _WIN32
    #define max_power 31
#endif
```

Your compilation toolchain might consider the macro `macro_name` as implicitly defined and execute the `#ifdef` branch. However, the Polyspace compilation might not consider the macro as defined. Therefore, the `#ifdef` branch is not executed and the variable `max_power` is not declared.

Solution

To work around the compilation error, do one of the following:

- Use **Target & Compiler** options to directly specify your compiler. For instance, to emulate a Visual C++ compiler, set the **Compiler** to `visual12.0`. See “Target and Compiler”.
- Define the macro explicitly using the option `Preprocessor definitions (-D)`.

Note If you create a Polyspace by tracing your build commands, most **Target & Compiler** options are automatically set.

Possible Cause: Project Created from Non-Debug Build

This can be a possible cause only if the undefined identifier occurs in an `assert` statement (or an equivalent Visual C++ macro such as `ASSERT` or `VERIFY`).

Typically, you come across this error in the following way. You create a Polyspace project from a build system in non-debug mode. When you run an analysis on the project, you face a compilation error from an undefined identifier in an `assert` statement. You find that the identifier `my_identifier` is defined in a `#ifndef NDEBUG` statement, for instance as follows:

```
#ifndef NDEBUG
int my_identifier;
#endif
```

The C standard states that when the `NDEBUG` macro is defined, all `assert` statements must be disabled.

Most IDEs define the `NDEBUG` macro in their build systems. When you build your source code in your IDE in non-debug mode, code in a `#ifndef NDEBUG` statement is removed during preprocessing. For instance, in the preceding example, `my_identifier` is not defined. If `my_identifier` occurs only in `assert` statements, it is not used either, because `NDEBUG` disables `assert` statements. You do not have compilation errors from undefined identifiers and your build system executes successfully.

Polyspace does not disable `assert` statements even if `NDEBUG` macro is defined because the software uses `assert` statements internally to enhance verification.

When you create a Polyspace project from your build system, if your build system defines the `NDEBUG` macro, it is also defined for your Polyspace project. Polyspace removes code in a `#ifndef NDEBUG` statement during preprocessing, but does not disable `assert` statements. If `assert` statements in your code rely on the code in a `#ifndef NDEBUG` statement, compilation errors can occur.

In the preceding example:

- The definition of `my_identifier` is removed during preprocessing.

- `assert` statements are not disabled. When `my_identifier` is used in an `assert` statement, you get an error because of undefined identifier `my_identifier`.

Solution

To work around this issue, create a Polyspace project from your build system in debug mode. When you execute your build system in debug mode, `NDEBUG` is not defined. When you create a Polyspace project from this build, `NDEBUG` is not defined for your Polyspace project.

Depending on your project settings, use the option that enables building in debug mode. For instance, if your build system is `gcc`-based, you can define the `DEBUG` macro and undefine `NDEBUG`:

```
gcc -DDEBUG=1 -UNDEBUG *.c
```

Alternatively, you can disable the `assert` statements in your preprocessed code using the option `Preprocessor definitions (-D)`. However, Polyspace will not be able to emulate the `assert` statements.

Unknown Function Prototype Error

Issue

During the compilation phase, the software displays a warning or error message about unknown function prototype.

```
the prototype for function 'myfunc' is unknown
```

The message indicates that Polyspace cannot find a function prototype. Therefore, it cannot identify the data types of the function argument and return value, and has to infer them from the calls to the function.

To determine the data types for such functions, Polyspace follows the C99 Standard (ISO/IEC 9899:1999, Chapter 6.5.2.2: Function calls).

- The return type is assumed to be `int`.
- The number and type of arguments are determined by the first call to the function. For instance, if the function takes one `double` argument in the first call, for subsequent calls, the software assumes that it takes one `double` argument. If you pass an `int` argument in a subsequent call, a conversion from `int` to `double` takes place.

During the linking phase, if a mismatch occurs between the number or type of arguments or the return type in different compilation units, the analysis follows an internal algorithm to resolve this mismatch and determine a common prototype.

Cause

The source code you provided does not contain the function prototype. For instance, the function is declared in an include file that Polyspace cannot find.

If you `#include`-d the include file in your source code but did not add it to your Polyspace project, you see a previous warning:

```
Warning: could not find include file "my_include.h"
```

Solution

Search for the function declaration in your source repository.

If you find the function declaration in an include file, add the folder that contains the include file.

- In the user interface of the Polyspace desktop products, add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Bug Finder).

- At the command line, use the flag `-I` with the `polyspace-bug-finder-server` command.

For more information, see `-I`.

Error Related to #error Directive

Issue

The analysis stops with a message containing a `#error` directive. For instance, the following message appears: `#error directive: !Unsupported platform; stopping!`.

Cause

You typically use the `#error` directive in your code to trigger a fatal error in case certain macros are not defined. Your compiler implicitly defines the macros, therefore the error is not triggered when you compile your code. However, the default Polyspace compilation does not consider the macros as defined, therefore, the error occurs.

For instance, in the following example, the `#error` directive is reached only if the macros `__BORLANDC__`, `__VISUALC32__` or `__GNUC__` are not defined. If you use a GNU C compiler, for instance, the compiler considers the macro `__GNUC__` as defined and the error does not occur. However, if you use the default Polyspace compilation, it does not consider the macros as defined.

```
#if defined(__BORLANDC__) || defined(__VISUALC32__)
#define MYINT int
#elif defined(__GNUC__)
#define MYINT long
#else
#error !Unsupported platform; stopping!
#endif
```

Solution

For successful compilation, do one of the following:

- Specify a compiler such as `visual12.0` or `gnu4.9`. Specifying a compiler defines some of the compilation flags for the analysis.

For more information, see `Compiler (-compiler)`.

- If the available compiler options do not match your compiler, explicitly define one of the compilation flags `__BORLANDC__`, `__VISUALC32__`, or `__GNUC__`.

For more information, see Preprocessor definitions (-D).

Large Object Error

Issue

The analysis stops during compilation with a message indicating that an object is too large.

Cause

The error happens when the software detects an object such as an array, union, structure, or class, that is too big for the pointer size of the selected target.

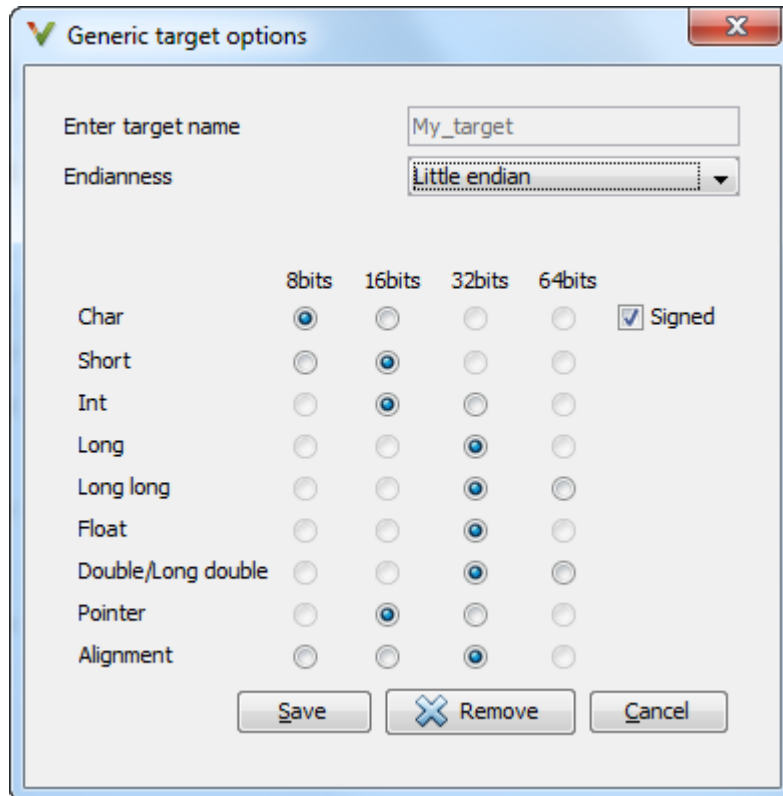
For instance, you get the message, `Limitation: struct or union is too large` in the following example. You specify a pointer size of 16 bits. The maximum object size allocated to a pointer, and therefore the maximum allowed size for an object, can be $2^{16}-1$ bytes. However, you declare a structure as follows:

- ```
struct S
{
 char tab[65536];
};
```
- ```
struct S
{
    char tab[65534];
    int val;
};
```

Solution

- 1 Check the pointer size that you specified through your target processor type. For more information, see `Target processor type (-target)`.

For instance, in the following, the pointer size for a custom target `My_target` is 16 bits.



- 2 Change your code or specify a different pointer size.

For instance, you can:

- Declare an array of smaller size in the structure.

If you are using a predefined target processor type, the pointer size is likely to be the same as the pointer size on your target architecture. Therefore, your declaration might cause errors on your target architecture.

- Change the pointer size of the target processor type that you specified, if possible.

Otherwise, specify another target processor type with larger pointer size or define your own target processor type. For more information on defining your own processor type, see [Generic target options](#).

Note Polyspace imposes an internal limit of 128 MB on the size of data structures. Even if your target processor type specification allows data structures of larger size, this internal limit constrains the data structure sizes.

Errors Related to Generic Compiler

If you use a generic compiler, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Issue

The analysis stops with an error message related to a non-ANSI C keyword, for instance, `data` or attributes such as `__attribute__((weak))`.

Depending on the location of the keyword, the error message can vary. For instance, this line causes the error message: `expected a ";"`.

```
data int tab[10];
```

Cause

The generic Polyspace compiler supports only ANSI C keywords. If you use a language extension, the generic compiler does not recognize it and treats the keyword as a regular identifier.

Solution

Specify your compiler by using the option `Compiler (-compiler)`.

If your compiler is not directly supported or is not based on a supported compiler, you can use the generic compiler. To work around the compilation errors:

- If the keyword is related to memory modelling, remove it from the preprocessed code. For instance, to remove the `data` keyword, enter `data=` for the option `Preprocessor definitions (-D)`.
- If the keyword is related to an attribute, remove attributes from the preprocessed code. Enter `__attribute__(x)=` for the option `Preprocessor definitions (-D)`.

If your code has this line:

```
void __attribute__((weak)) func(void);
```

And you remove attributes, the analysis reads the line as:

```
void func(void);
```

When you use these workarounds, your source code is not altered.

Errors Related to Keil or IAR Compiler

If you use the compiler, Keil or IAR, you can encounter this issue. For more information, see `Compiler (-compiler)`.

Missing Identifiers

Issue

The analysis stops with the error message, `expected an identifier`, as if an identifier is missing. However, in your source code, you can see the identifier.

Cause

If you select Keil or IAR as your compiler, the software removes certain keywords during preprocessing. If you use these keywords as identifiers such as variable names, a compilation error occurs.

For a list of keywords that are removed, see “Supported Keil or IAR Language Extensions” on page 5-24.

Solution

Specify that Polyspace must not remove the keywords during preprocessing. Define the macros `__PST_KEIL_NO_KEYWORDS__` or `__PST_IAR_NO_KEYWORDS__`.

For more information, see `Preprocessor definitions (-D)`.

Errors Related to Diab Compiler

If you choose `diab` for the option `Compiler (-compiler)`, you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a keyword specific to the Diab compiler. For instance, you see an error related to the `restrict` keyword.

Cause

You typically use a compiler flag to enable the keyword. The Polyspace analysis does not enable these keywords by default. You have to make Polyspace aware of your compiler flags.

The Polyspace analysis does not enable these keywords by default to prevent compilation errors. Another user might not enable the keyword and instead use the keyword name as a regular identifier. If Polyspace treats the identifier as a keyword, a compilation error will occur.

Solution

Use the command-line option `-compiler-parameter` in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface of the Polyspace desktop products, you can enter the command-line option in the field `Other`. You can enter the option multiple times.

The argument of `-compiler-parameter` depends on the keyword that causes the error. Once you enable the keyword, do not use the keyword name as a regular identifier. For instance, once you enable the keyword `pixel`, do not use `pixel` as a variable name. The statement `int pixel = 1` causes a compilation error.

- `restrict` keyword:

You typically use the compiler flag `-Xlibc-new` or `-Xc-new`. For your Polyspace analysis, use

```
-compiler-parameter -Xc-new
```

The following code will not compile with Polyspace unless you specify the compiler flag.

```
int sscanf(const char *restrict, const char *restrict, ...);
```

- PowerPC AltiVec vector extensions such as the vector type qualifier:

You typically use the compiler flag `-tPPCALLAV:.` For your Polyspace analysis, use

```
-compiler-parameter -tPPCALLAV:
```

The following code will not compile with Polyspace unless you specify the compiler flag.

```
vector unsigned char vbyte;
vector bool vbool;
vector pixel vpx;
```

```
int main(int argc, char** argv)
{
    return 0;
}
```

- Extended keywords such as `pascal`, `inline`, `packed`, `interrupt`, `extended`, `__X`, `__Y`, `vector`, `pixel`, `bool` and others:

You typically use the compiler flag `-Xkeywords=.` For your Polyspace analysis, use

```
-compiler-parameter -Xkeywords=0xFFFFFFFF
```

The following code will not compile with Polyspace unless you specify the compiler flag.

```
packed(4) struct s2_t {
    char b;
    int i;
} s2;
```

```
packed(4,2) struct s3_t {
    char b;
} s3;
```

```
int pascal foo = 4;
```

```
int main(int argc, char** argv) {
    foo++;
}
```

```
    return 0;  
}
```

Errors Related to Green Hills Compiler

If you choose `greenhills` for the option `Compiler (-compiler)`, you encounter this issue.

Issue

During Polyspace analysis, you see an error related to vector data types specific to Green Hills target `rh850`. For instance, you see an error related to identifier `__ev64_u16__`.

Cause

When compiling code using the Green Hills compiler with target `rh850`, to enable single instruction multiple data (SIMD) vector instructions, you specify two flags:

- `-rh850_simd`: You enable intrinsic functions that support SIMD vector instructions. The functions are defined in your compiler header files. These data types are available:
 - `__ev64_u16__`
 - `__ev64_s16__`
 - `__ev64_u32__`
 - `__ev64_s32__`
 - `__ev64_u64__`
 - `__ev64_s64__`
 - `__ev64_opaque__`
 - `__ev128_opaque__`
- `-rh850_fpsimd`: You enable intrinsic functions that support floating-point SIMD vector instructions. The functions are defined in your compiler header files. These data types are available:
 - `__ev128_f32__`
 - `__ev256_f32__`

The Polyspace analysis does not enable SIMD support by default. You must identify your compiler flags to Polyspace.

Solution

In your Polyspace analysis, use the command-line option `-compiler-parameter`. In the user interface, you can enter the command-line option in the **Other** field, under the **Advanced Settings** in the **Configuration** pane.

- `-rh850_simd`: For your Polyspace analysis, use
`-compiler-parameter -rh850_simd`
- `-rh850_fpsimd`: For your Polyspace analysis, use
`-compiler-parameter -rh850_fpsimd`

Note

- `__ev128_opaque__` is 16 bytes aligned in Polyspace.
 - `__ev256_f32__` is 32 bytes aligned in Polyspace.
-

Errors Related to TASKING Compiler

If you choose tasking for the option Compiler (-compiler), you can encounter this issue.

Issue

During Polyspace analysis, you see an error related to a Special Function Register data type.

Cause

When compiling with the TASKING compiler, you typically use the following compiler flags to specify where Special Function Register (SFR) data types are declared:

- `--cpu=xxx`: The compiler implicitly `#includes` the file `sfr/regxxx.sfr` in your source files. Once `#include-ed`, you can use Special Function Registers (SFR-s) declared in that `.sfr` file.
- `--alternative-sfr-file`: The compiler uses an alternative SFR file instead of the regular SFR file. You can use Special Function Registers (SFR-s) declared in that alternative SFR file.

If you specify the TASKING compiler for your Polyspace analysis, the analysis makes the following assumptions about these compiler flags:

- `--cpu=xxx`: The analysis chooses a specific value of `xxx`. If you use a different value with your TASKING compiler, you can encounter an error during Polyspace analysis.

The `xxx` value that the Polyspace analysis uses depends on your choice of Target processor type (`-target`):

- `tricore`: tc1793b
- `c166`: xc167ci
- `rh850`: r7f701603
- `arm`: ARMv7M
- `--alternative-sfr-file`: The analysis assumes that you do not use an alternative SFR file. If you use one, you can encounter an error.

Solution

Use the command-line option `-compiler-parameter` in your Polyspace analysis as follows. You use this command-line option to make Polyspace aware of your compiler flags. In the user interface, you can enter the command-line option in the field **Other**. You can enter the option multiple times.

- `--cpu=xxx`: For your Polyspace analysis, use
`-compiler-parameter --cpu=xxx`

Here, `xxx` is the value that you use when compiling with your compiler.

- `--alternative-sfr-file`: For your Polyspace analysis, use
`-compiler-parameter --alternative-sfr-file`

If you still encounter an error because Polyspace is not able to locate your `.asfr` file, explicitly `#include` your `.asfr` file in the preprocessed code using the option `Include (-include)`.

Typically, the path to the file is `Tasking_C166_INSTALL_DIR\include\sfr\regCPUNAME.asfr`. For instance, if your TASKING compiler is installed in `C:\Program Files\Tasking\C166-VX_v4.0r1\` and you use the CPU-related flag `-Cxc2287m_104f` or `--cpu=xc2287m_104f`, the path is `C:\Program Files\Tasking\C166-VX_v4.0r1\include\sfr\regxc2287m.asfr`.

You can also encounter the same issue with alternative sfr files when you trace your build command. For more information, see “Requirements for Project Creation from Build Systems” on page 5-21.

Errors from Conflicts with Polyspace Header Files

Issue

You see compilation errors from header files included by Polyspace.

For instance, the error message refers to one of the subfolders of *polyspaceroot* \polyspace\verifier\cxx\include.

Typically, the error message is related to a standard library function.

Cause

If your compiler defines a standard library function or another construct and you do not provide the path to your compiler header files, Polyspace uses its own implementation of the function.

If your compiler definitions differ from the corresponding Polyspace definitions, the verification stops with an error.

Solution

Specify the folder containing your compiler header files.

- In the user interface, add the folder to your project.

For more information, see “Add Source Files for Analysis in Polyspace User Interface” (Polyspace Bug Finder).

- At the command line, use the flag `-I` with the `polyspace-bug-finder-server` command.

For more information, see `-I`.

For compilation with GNU C on UNIX-based platforms, use `/usr/include`. On embedded compilers, the header files are typically in a subfolder of the compiler installation folder. Examples of include folders are given for some compilers.

- Wind River Diab: For instance, `/apps/WindRiver/Diab/5.9.4/diab/5.9.4.8/include/`.

- IAR Embedded Workbench: For instance, C:\Program Files\IAR Systems\Embedded Workbench 7.5\arm\inc.
- Microsoft Visual Studio: For instance, C:\Program Files\Microsoft Visual Studio 14.0\VC\include.

Consult your compiler documentation for the path to your compiler header files. Alternatively, see “Provide Standard Library Headers for Polyspace Analysis” on page 5-19.

Errors from Assertion or Memory Allocation Functions

Issue

Polyspace uses its own implementation of standard library functions for more efficient analysis. If you redefine a standard library function and provide the function body to Polyspace, the analysis uses your definition.

However, for certain standard library functions, Polyspace continues to use its own implementations, even if you redefine the function and provide the function body. The functions include `assert` and memory allocation functions such as `malloc`, `calloc` and `alloca`.

You see a warning message like the following:

```
Body of routine "malloc" was discarded.
```

Cause

These functions have special meaning for the Polyspace analysis, so you are not allowed to redefine them. For instance:

- The Polyspace implementation of the `malloc` function allows the software to check if memory allocated using `malloc` is freed later.
- The Polyspace implementation of `assert` is used internally to enhance analysis.

Solution

Unless you particularly want your own redefinitions to be used, ignore the warning. The analysis results are based on Polyspace implementations of the standard library function, which follow the original function specifications.

If you want your own redefinitions to be used and you are sure that your redefined function behaves the same as the original function, rename the functions. You can rename the function only for the purposes of analysis using the option `Preprocessor definitions (-D)`. For instance, to rename a function `malloc` to `my_malloc`, use `malloc=my_malloc` for the option argument.

Error from Special Characters

Issue

Your file or folder names contain extended ASCII characters, such as accented letters or Kanji characters. You face file access errors during analysis. Error messages you might see include:

- No source files to analyze
- Control character not valid
- Cannot create directory *Folder_Name*

Cause

Polyspace does not fully support these characters. If you use extended ASCII in your file or folder names, your Polyspace analysis may fail due to file access errors.

Workaround

Change the unsupported ASCII characters to standard US-ASCII characters.

Errors from In-Class Initialization (C++)

When a data member of a class is declared `static` in the class definition, it is a *static member* of the class. You must initialize static data members outside the class because they exist even when no instance of the class has been created.

```
class Test
{
public:

    static int m_number = 0;
};
```

Error message:

Error: a member with an in-class initializer must be const

Corrected code:

in file Test.h	in file Test.cpp
<pre>class Test { public: static int m_number; };</pre>	<pre>int Test::m_number = 0;</pre>

Errors from Double Declarations of Standard Template Library Functions (C++)

Consider the following code.

```
#include <list>

void f(const std::list<int*>::const_iterator it) {}
void f(const std::list<int*>::iterator it) {}
void g(const std::list<int*>::const_reverse_iterator it) {}
void g(const std::list<int*>::reverse_iterator it) {}
```

The declared functions belong to `list` container classes with different iterators. However, the software generates the following compilation errors:

```
error: function "f" has already been defined
error: function "g" has already been defined
```

You would also see the same error if, instead of `list`, the specified container was `vector`, `set`, `map`, or `deque`.

To avoid the double declaration errors, do one of the following:

- Deactivate automatic stubbing of standard template library functions. For more information, see `No STL stubs (-no-stl-stubs)`.
- Define the following Polyspace preprocessing directives:
 - `__PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_VECTOR_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_SET_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_MAP_CONST_ITERATOR_DIFFER_ITERATOR__`
 - `__PST_STL_DEQUE_CONST_ITERATOR_DIFFER_ITERATOR__`

For example, for the given code, run analysis at the command line with the following flag. The flag defines the appropriate directive for the `list` container.

```
-D __PST_STL_LIST_CONST_ITERATOR_DIFFER_ITERATOR__
```

For more information on defining preprocessor directives, see `Preprocessor definitions (-D)`.

Errors Related to GNU Compiler

If you choose `gnu` for the option `Compiler` (`-compiler`), you can encounter this issue.

Issue

The Polyspace analysis stops with a compilation error.

Cause

You are using certain advanced compiler-specific extensions that Polyspace does not support. See “Limitations”.

Solution

For easier portability of your code, avoid using the extensions.

If you want to use the extensions and still analyze your code, wrap the unsupported extensions in a preprocessor directive. For instance:

```
#ifdef POLYSPACE
    // Supported syntax
#else
    // Unsupported syntax
#endif
```

For regular compilation, do not define the macro `POLYSPACE`. For Polyspace analysis, enter `POLYSPACE` for the option `Preprocessor definitions` (`-D`).

If the compilation error is related to assembly language code, use the option `-asm-begin` `-asm-end`.

Errors Related to Visual Compilers

The following messages appear if the compiler is based on a Visual compiler. For more information, see `Compiler (-compiler)`.

Import Folder

When a Visual application uses `#import` directives, the Visual C++ compiler generates a header file with extension `.tlh` that contains some definitions. To avoid compilation errors during Polyspace analysis, you must specify the folder containing those files.

Original code:

```
#include "stdafx.h"
#include <comdef.h>
#import <MsXml.tlb>
MSXML::_xml_error e ;
MSXML::DOMDocument* doc ;
int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

Error message:

```
"../sources/ImportDir.cpp", line 7: catastrophic error: could not
open source file "../MsXml.tlh"
    #import <MsXml.tlb>
```

The Visual C++ compiler generates these files in its “build-in” folder (usually Debug or Release). In order to provide those files:

- Build your Visual C++ application.
- Specify your build folder for the Polyspace analysis.

pragma Pack

Using a different value with the compile flag (`#pragma pack`) can lead to a linking error message.

Original code:

test1.cpp	type.h	test2.cpp
<pre>#pragma pack(4) #include "type.h"</pre>	<pre>struct A { char c ; int i ; } ;</pre>	<pre>#pragma pack(2) #include "type.h"</pre>

Error message:

```
Pre-linking C++ sources ...
"../sources/type.h", line 2: error: declaration of class "A" had
a different meaning during compilation of "test1.cpp"
(class types do not match)
    struct A
      ^
      detected during compilation of secondary translation unit
"test2.cpp"
```

To continue the analysis, use the option `Ignore pragma pack directives (-ignore-pragma-pack)`.

C++/CLI

Polyspace does not support Microsoft C++/CLI, a set of language extensions for .NET programming.

You can get errors such as:

```
error: name must be a namespace name
|         using namespace System;
```

Or:

```
error: expected a declaration
|         public ref class Form1 : public System::Windows::Forms::Form
```

Error or Slow Runs from Disk Defragmentation and Anti-virus Software

Issue

In some cases, anti-virus software checks can noticeably slow down a Polyspace analysis. This reduction occurs because the software checks the temporary files produced by the Polyspace analysis.

You see noticeably slow analysis for a simple project or the analysis stops with an error message like the following:

```
Some stats on aliases use:
  Number of alias writes:      22968
  Number of must-alias writes: 3090
  Number of alias reads:       0
  Number of invisibles:       949
Stats about alias writes:
  biggest sets of alias writes: fool:a (733), foo2:x (728), fool1:b (728)
  procedures that write the biggest sets of aliases: fool (2679), foo2 (2266),
                                                    fool3 (1288)
**** C to intermediate language translation - 17 (P_PT) took 44real, 44u + 0s (1.4gc)
exception SysErr(OS.SysErr(name="Directory not empty", syserror=notempty)) raised.
unhandled exception: SysErr: No such file or directory [noent]
```

```
-----
---
--- Verifier has encountered an internal error. ---
--- Please contact your technical support. ---
---
-----
```

Possible Cause

A disk defragmentation tool or anti-virus software is running on your machine.

After starting an analysis, check the processes running and see if an anti-virus process is causing large amount of CPU usage (and possibly memory usage).

Solution

Try:

- Stopping the disk defragmentation tool.
- Deactivating the anti-virus software. Or, configuring exception rules for the anti-virus software to allow Polyspace to run without a failure.

For instance, you can try the following:

- Configure the anti-virus software to whitelist the Polyspace executables.

For instance, in Windows, with the anti-virus software Windows Defender, you can add an exclusion for the Polyspace installation folder `C:\Program Files\Polyspace\R2019a`, in particular, the `.exe` files in the subfolder `polyspace\bin` and the `.exe` files starting with `ps_` in the subfolder `bin\win64`.

- Configure the anti-virus software to exclude your temporary folder, for example, `C:\Temp`, from the checking process.

SQLite I/O Error

Issue

When you try to run Polyspace, you get this error message:

Cause

Polyspace uses an SQLite database for storing results. This error can appear when SQLite databases are saved on NFS (Network File System) folders.

Solution

Check the folder where you save Polyspace results. For instance, if you run Polyspace at the command line, check the option `-results-dir`.

If the folder is an NFS folder, use a local folder instead.