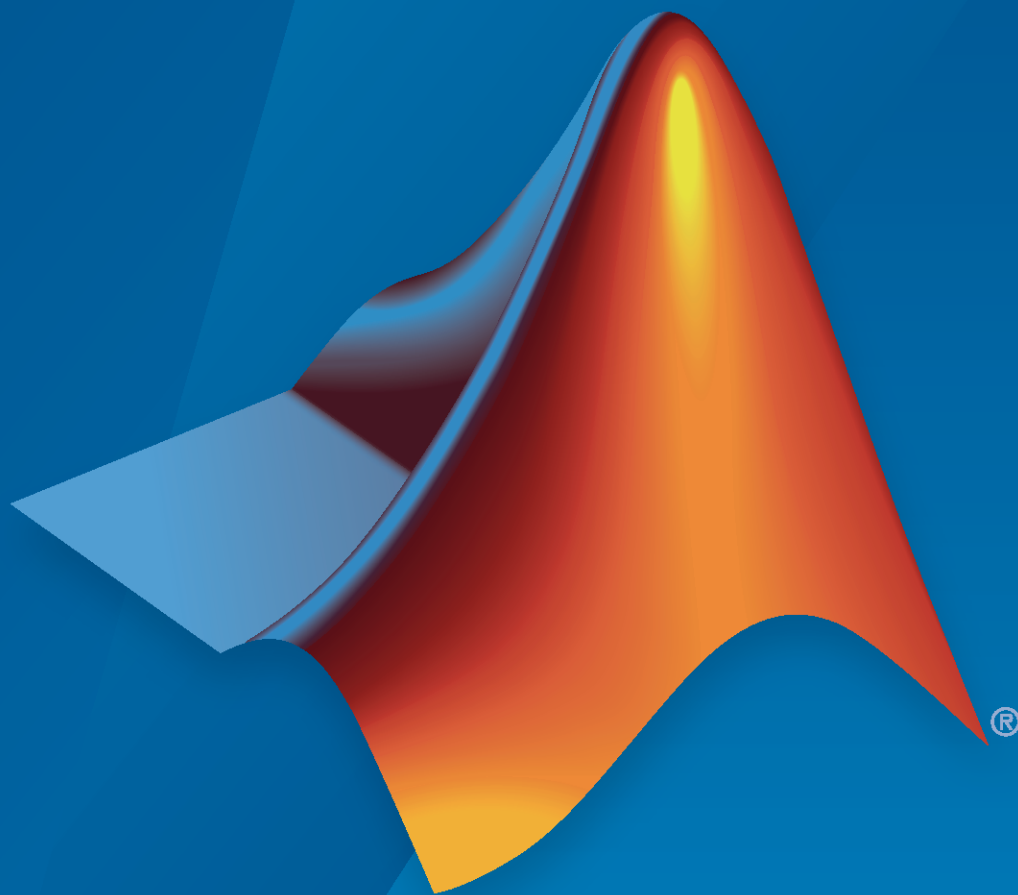


Polyspace® Bug Finder™ Access™

User's Guide



R2021a



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[™] Access[™] User's Guide

© COPYRIGHT 2019-2021 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 2.0 (Release R2019a)
September 2019	Online Only	Revised for Version 2.1 (Release 2019b)
March 2020	Online Only	Revised for Version 2.2 (Release 2020a)
September 2020	Online Only	Revised for Version 2.3 (Release 2020b)
March 2021	Online Only	Revised for Version 3.0 (Release 2021a)

1

Interpret Polyspace Bug Finder Results

Interpret Bug Finder Results in Polyspace Access Web Interface	1-2
Interpret Result Details Message	1-3
Find Root Cause of Result	1-4
Investigate the Cause of Empty Results List	1-7
Dashboard	1-9
Code Metrics Dashboard	1-11
Quality Objectives Dashboard	1-14
Customize Software Quality Objectives	1-15
Call Hierarchy	1-19
Configuration Settings	1-21
Result Details	1-24
Results List	1-26
Review History	1-28
Source Code	1-30
Tooltips	1-30
Examine Source Code	1-31
Expand Macros	1-31
View Code Block	1-32
Navigate from Code to Model	1-33
Track Issue in Bug Tracking Tool	1-35
Create a Ticket	1-35
Manage Existing Tickets	1-36
Bug Finder Quality Objectives	1-38
Comparing Analysis Results Against Quality Objectives	1-41
Software Quality Objective Subsets (C:2004)	1-43
Rules in SQO-Subset1	1-43
Rules in SQO-Subset2	1-44
Software Quality Objective Subsets (AC AGC)	1-47
Rules in SQO-Subset1	1-47

Rules in SQA-Subset2	1-47
Software Quality Objective Subsets (C:2012)	1-50
Guidelines in SQA-Subset1	1-50
Guidelines in SQA-Subset2	1-51
Avoid Violations of MISRA C 2012 Rules 8.x	1-53
Software Quality Objective Subsets (C++)	1-56
SQA Subset 1 - Direct Impact on Selectivity	1-56
SQA Subset 2 - Indirect Impact on Selectivity	1-57
Coding Rule Subsets Checked Early in Analysis	1-62
MISRA C: 2004 and MISRA AC AGC Rules	1-62
MISRA C: 2012 Rules	1-69
HIS Code Complexity Metrics	1-77
Project	1-77
File	1-77
Function	1-77

Fix or Comment Polyspace Results

2

Address Results in Polyspace Access Through Bug Fixes or Justifications	2-2
Add Review Information in Result Details pane	2-2
Comment or Annotate in Code	2-3
Hide Known or Acceptable Polyspace Results	2-5
Review Workflow Using Code Annotations	2-5
Code Annotation Syntax	2-6
Code Annotation Syntax Examples	2-9
Alternatives to Code Annotations	2-10
Short Names of Bug Finder Defect Checkers	2-12
Short Names of Code Complexity Metrics	2-26
Project Metrics	2-26
File Metrics	2-26
Function Metrics	2-26
Define Custom Annotation Format	2-28
Define Annotation Syntax Format	2-30
Map Your Annotation to the Polyspace Annotation Syntax	2-33
Define Multiple Custom Annotation Syntaxes	2-34
Annotation Description Full XML Template	2-36
Example	2-39

3

Filter and Sort Results in Polyspace Access Web Interface	3-2
Filter Results	3-4
Create Custom Filter Groups in Polyspace Access Web Interface	3-6
Compare Analysis Results to Previous Runs	3-8
Comparison Mode	3-8
Classification of Defects by Impact	3-11
High Impact Defects	3-11
Medium Impact Defects	3-14
Low Impact Defects	3-18
Bug Finder Defect Groups	3-22
Concurrency	3-22
Cryptography	3-23
Data flow	3-23
Dynamic Memory	3-23
Good Practice	3-23
Numerical	3-24
Object Oriented	3-24
Programming	3-24
Resource Management	3-25
Static Memory	3-25
Security	3-25
Tainted data	3-25

Troubleshooting Polyspace Access

4

Polyspace Access ETL and Web Server services do not start	4-2
Issue	4-2
Possible Cause: Hyper-V Network Configuration Cannot Resolve Local Host Names	4-2
Contact Technical Support About Polyspace Access Issues	4-5
Resolve -xml-annotations-description Errors	4-7
Issue	4-7
Possible Solutions	4-7

Configure Polyspace as You Code Extension in Visual Studio	5-2
General Settings	5-2
Project Settings	5-3
Configure Polyspace as You Code Extension in Visual Studio Code	5-6
Analysis Engine	5-6
Analysis Launch Mode	5-7
Analysis Setup	5-7
Baseline	5-15
Trace	5-16
Configure Polyspace as You Code Plugin in Eclipse	5-17
Preferences	5-17
Configure Project	5-18
Options Files for Polyspace Analysis	5-22
What are Options Files	5-22
Specifying Options Files	5-22
Specifying Multiple Options Files	5-24
Generate Build Options for Polyspace as You Code Analysis in Visual Studio	5-25
Configure Polyspace as You Code to Extract Build Configuration	5-25
Specify Analysis Options Manually	5-27
Import Analysis Options from Polyspace Desktop Project	5-28
Generate Build Options for Polyspace as You Code Analysis in Visual Studio Code	5-29
Configure Polyspace as You Code to Extract Build Configuration	5-29
Specify Analysis Options Manually	5-31
Import Analysis Options from Polyspace Desktop Project	5-32
Generate Build Options for Polyspace as You Code Analysis in Eclipse	5-34
Configure Polyspace as You Code to Extract Build Configuration	5-34
Specify Analysis Options Manually	5-36
Import Analysis Options from Polyspace Desktop Project	5-36
Generate Build Options for Polyspace as You Code Analysis at the Command Line	5-38
Use polyspace-configure to Generate Build Options File	5-38
Specify Analysis Options Manually	5-39
Import Analysis Options from Polyspace Desktop Project	5-40
Baseline Polyspace as You Code Results in Visual Studio	5-41
What Baselined Results Look Like	5-41
Baselining Steps	5-43
Baseline Polyspace as You Code Results in Visual Studio Code	5-45
What Baselined Results Look Like	5-45
Baselining Steps	5-46

Baseline Polyspace as You Code Results in Eclipse	5-50
What Baselined Results Look Like	5-50
Baselining Steps	5-51
Baseline Polyspace as You Code Results on Command Line	5-53
What Baselined Results Look Like	5-53
Baselining Steps	5-54
Step 1: Identify Project to Use as Baseline	5-54
Step 2: Download Baseline	5-55
Step 3: Use Baseline	5-55
Configure Checkers for Polyspace as You Code in Eclipse	5-57
Select Checkers and Coding Rules	5-57
Modify Checker Behavior	5-60
Configure Checkers for Polyspace as You Code in Visual Studio	5-61
Select Checkers and Coding Rules	5-61
Modify Checker Behavior	5-63
Configure Checkers for Polyspace as You Code in Visual Studio Code ..	5-64
Configure Checkers in Checkers File	5-64
Modify Checkers Behavior	5-66
Configure Checkers for Polyspace as You Code at the Command Line ..	5-68
Configure Checkers and Coding Rules Directly at the Command Line ...	5-68
Configure Checkers in Checkers file	5-69
Modify Checkers Behavior	5-71
Polyspace Bug Finder Defects Checkers Enabled by Default	5-73
Analysis Scope of Polyspace as You Code	5-78
Results Involve Current File Only	5-78
Headers Included in Current File Not Analyzed	5-78
Checkers Deactivated in Polyspace as You Code Default Analysis	5-80
Checkers and Coding Rule Deactivated in Polyspace as You Code	5-80
Checkers with Reduced Scope in Polyspace as You Code	5-81
Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code	5-83
Issue	5-83
Possible Solutions	5-83
Reduce Software Complexity by Using Polyspace Checkers	5-86
Configure Thresholds for Software Complexity Checkers	5-86
Identify and Reduce Software Complexity	5-87

Review Results in Polyspace as You Code

6

Run Polyspace as You Code in Visual Studio and Review Results	6-2
Confirm Installation of Extension	6-2

Run Analysis on Save	6-2
Run Analysis on Demand	6-3
Review Results	6-3
Justify Results Using Code Annotations	6-4
View Help	6-4
Configure Checkers and Other Settings	6-5
Run Polyspace as You Code in Visual Studio Code and Review Results . .	6-6
Confirm Installation of Extension	6-6
Run Analysis on Save	6-6
Run Analysis on Demand	6-6
Review Results	6-7
Justify Results Using Code Annotations	6-8
View Context-Sensitive Help for Result	6-8
Configure Checkers and Other Settings	6-9
Run Polyspace as You Code in Eclipse and Review Results	6-10
Confirm Installation of Plugin	6-10
Run Analysis on Save	6-10
Run Analysis on Demand	6-11
Review Results	6-11
Justify Results Using Code Annotations	6-12
View Context-Sensitive Help for Result	6-12
Configure Checkers and Other Settings	6-13
Run Polyspace as You Code from Command Line and Export Results . .	6-14
Add Install Folder to Path	6-14
Run Analysis and See Results on Console	6-14
Store Results in Specific Folder	6-14
Export Results to JSON Format (SARIF Output)	6-15
Specify Analysis Options by Using Options Files	6-15
Create Options File by Analyzing Build	6-15
Integrate Polyspace as You Code in IDEs and Editors Without Plugins	
.	6-17
Overview of Approach	6-17
Integration Steps	6-17
Further Exploration	6-19

Coding Rule Sets and Concepts

7

Polyspace MISRA C:2004 and MISRA AC AGC Checkers	7-2
MISRA C:2004 and MISRA AC AGC Coding Rules	7-3
Supported MISRA C:2004 and MISRA AC AGC Rules	7-3
Troubleshooting	7-3
List of Supported Coding Rules	7-3
Unsupported MISRA C:2004 and MISRA AC AGC Rules	7-36
Polyspace MISRA C:2012 Checkers	7-38

Essential Types in MISRA C:2012 Rules 10.x	7-39
Categories of Essential Types	7-39
How MISRA C:2012 Uses Essential Types	7-39
Unsupported MISRA C:2012 Guidelines	7-41
Polyspace MISRA C++ Checkers	7-42
Unsupported MISRA C++ Coding Rules	7-43
Language Independent Issues	7-43
General	7-44
Lexical Conventions	7-44
Expressions	7-44
Declarations	7-44
Classes	7-45
Templates	7-45
Exception Handling	7-45
Library Introduction	7-45
Polyspace JSF AV C++ Checkers	7-47
JSF AV C++ Coding Rules	7-48
Supported JSF C++ Coding Rules	7-48
Unsupported JSF++ Rules	7-66

Configure Target and Compiler Options

8

Specify Target Environment and Compiler Behavior	8-2
Extract Options from Build Command	8-2
Specify Options Explicitly	8-3
C/C++ Language Standard Used in Polyspace Analysis	8-5
Supported Language Standards	8-5
Default Language Standard	8-5
C11 Language Elements Supported in Polyspace	8-7
C++11 Language Elements Supported in Polyspace	8-9
C++14 Language Elements Supported in Polyspace	8-12
C++17 Language Elements Supported in Polyspace	8-15
Provide Standard Library Headers for Polyspace Analysis	8-19
Requirements for Project Creation from Build Systems	8-20
Compiler Requirements	8-20
Build Command Requirements	8-21
Supported Keil or IAR Language Extensions	8-23
Special Function Register Data Type	8-23

Keywords Removed During Preprocessing	8-24
Remove or Replace Keywords Before Compilation	8-25
Remove Unrecognized Keywords	8-25
Remove Unrecognized Function Attributes	8-27
Gather Compilation Options Efficiently	8-28

Approximations Used During Bug Finder Analysis

9

Inputs in Polyspace Bug Finder	9-2
Global Variables in Polyspace Bug Finder	9-3
Volatile Variables in Polyspace Bug Finder	9-4

Interpret Polyspace Bug Finder Results

- “Interpret Bug Finder Results in Polyspace Access Web Interface” on page 1-2
- “Investigate the Cause of Empty Results List” on page 1-7
- “Dashboard” on page 1-9
- “Code Metrics Dashboard” on page 1-11
- “Quality Objectives Dashboard” on page 1-14
- “Call Hierarchy” on page 1-19
- “Configuration Settings” on page 1-21
- “Result Details” on page 1-24
- “Results List” on page 1-26
- “Review History” on page 1-28
- “Source Code” on page 1-30
- “Track Issue in Bug Tracking Tool” on page 1-35
- “Bug Finder Quality Objectives” on page 1-38
- “Software Quality Objective Subsets (C:2004)” on page 1-43
- “Software Quality Objective Subsets (AC AGC)” on page 1-47
- “Software Quality Objective Subsets (C:2012)” on page 1-50
- “Avoid Violations of MISRA C 2012 Rules 8.x” on page 1-53
- “Software Quality Objective Subsets (C++)” on page 1-56
- “Coding Rule Subsets Checked Early in Analysis” on page 1-62
- “HIS Code Complexity Metrics” on page 1-77

Interpret Bug Finder Results in Polyspace Access Web Interface

This topic shows how to review Bug Finder results in the Polyspace Access web interface. For a similar workflow in the user interface of the Polyspace desktop products, see “Interpret Bug Finder Results in Polyspace Desktop User Interface” (Polyspace Bug Finder). To see how to review results of Polyspace as You Code in IDEs, see “Run Polyspace as You Code in IDEs and Review Results”.

When you open the results of a Bug Finder analysis in the **REVIEW** view of Polyspace Access, you see a list on the **Results List** pane. The results consist of defects, coding rule violations or code metrics.

You can first narrow down the focus of your review:

- Use filters in the toolstrip to narrow down the list. For instance, you can focus on the high-impact defects.
- Click the a column header in the **Results List** to sort the list according to the content of that column. For instance you can sort by **Group** or by **File**.

Once you narrow down and sort the list, you can begin reviewing individual results. This topic describes how to review a result.

The screenshot displays the Polyspace Access web interface. The top navigation bar includes tabs for Dashboard, Running Checks, Defects, Coding Standards, Code Metrics, and Global Variables. The main area is divided into several panes:

- Results List:** A table with columns for Family, ID, Type, Group, and Check. It shows a list of defects, with the row for ID 77758 (Invalid free of pointer) selected.
- Result Details:** A pane showing the details for the selected defect, including its status (Unreviewed), severity (Unset), and assigned user. It also includes a comment field and a track issue button.
- Event Log:** A table showing the sequence of events leading to the defect, such as taking the address of a variable and calling a function.
- Source Code:** A pane showing the source code for the selected defect, with the line `Free(p);` highlighted. A comment next to it reads: `/* Defect: free on a non-allocated memory */`.

Annotations with arrows point to these key elements:

- Select a result:** Points to the selected row in the Results List.
- Read result explanation:** Points to the Result Details pane.
- See source code:** Points to the Source Code pane.

To begin your review, select a result in the list.

Interpret Result Details Message

The screenshot displays the Polyspace Bug Finder interface. The main window is titled 'Result Details' and shows a bug entry for 'Invalid use of standard library integer routine (Impact: High)'. The bug is assigned to 'Unreviewed' and 'Unset' severity. The description states: 'Standard function 'div' is called with an invalid argument. second argument (denominator) is zero. return value may overflow.' Below the description is a table with columns 'Event', 'File', and 'Scope'. The event is 'Invalid use of standa...' in file 'numerical.c' at scope 'bug_intstdlib()'. The source code pane shows the relevant code snippet:


```

300
301
302 }
303
304
305 /*
306  * INVALID USE OF INTEGER STANDARD LIBRARY
307  *
308  * div_t bug_intstdlib(int num, int denom) {
309  *   div_t test;
310  *
311  *   if (denom == 0)
312  *     test = div(num, denom); /* Defect: Gener
313  *   else if ((num == INT_MIN) && (denom == -1))
314  *     test = div(num, denom); /* Defect: Generates an ove
315  *   else
316  *     test = div(num, denom);
317  *   return test;
318  * }
319  *
320  *
  
```



 To the right, a 'Contextual Help' pane is open, titled 'Invalid use of standard library integer routine'. It provides a detailed description: 'Wrong arguments to standard library function'. It lists examples of functions that trigger this defect: Character Conversion (toupper, tolower), Character Checks (isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit), Integer Division (div, ldiv), and Absolute Values (abs, labs). It also includes a 'Fix' section and a 'Check Information' section with fields for Group, Language, Default, Command-Line Syntax, Impact, and CWE ID.
 Three callout boxes with arrows point to specific parts of the interface:

- 'Open contextual help.' points to the help icon in the bug description.
- 'Read brief explanation.' points to the main description text.
- 'Read detailed explanation with examples.' points to the 'Contextual Help' pane.
- 'Check external standards.' points to the 'CWE ID' field in the 'Check Information' section.

Interpret Message

The first step is to understand what is wrong. Read the message on the **Result Details** pane and the related line of code on the **Source Code** pane.

Seek Additional Resources for Help

Sometimes, you need additional help for certain results. Click the  icon to open a help page for the selected result. See code examples illustrating the result. Check external standards such as CERT-C that provide additional rationale for fixing the issue. When available, click the  icon to see fix suggestions for the defect.

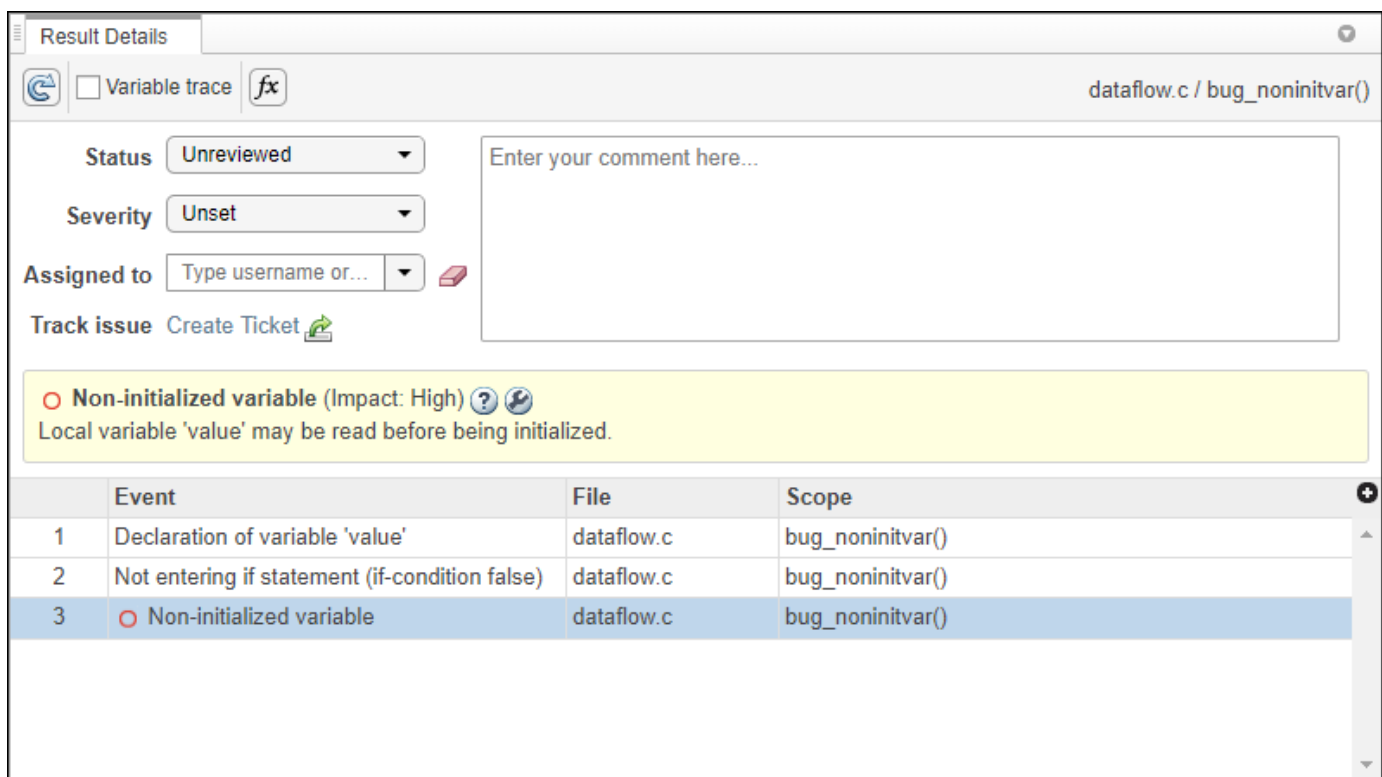
At this point, you might be ready to decide whether to fix the issue or not. Once you identify a fix, it might help to review all results of that type together.

Find Root Cause of Result

Sometimes, the root cause might be far from the actual location where the result is displayed. For instance, a variable that you read might be non-initialized because the initialization is not reachable. The defect is shown when you read the variable, but the root cause is perhaps a previous `if` or `while` condition that is always false.

Navigate to Related Events

Typically, the **Result Details** pane shows one sequence of events that leads to the result. The **Source Code** pane also highlights these events.



The screenshot shows the 'Result Details' pane for a bug. At the top, there's a 'Variable trace' checkbox and a 'fx' icon. The file path is 'dataflow.c / bug_noninitvar()'. Below this, there are dropdown menus for 'Status' (Unreviewed), 'Severity' (Unset), and 'Assigned to' (Type username or...). There's a 'Track issue' button with a 'Create Ticket' link. A comment box is present with the text 'Enter your comment here...'. A yellow warning box contains the message: 'Non-initialized variable (Impact: High) Local variable 'value' may be read before being initialized.' Below this is a table with three columns: 'Event', 'File', and 'Scope'. The table lists three events: 1. Declaration of variable 'value' in dataflow.c at scope bug_noninitvar(). 2. Not entering if statement (if-condition false) in dataflow.c at scope bug_noninitvar(). 3. Non-initialized variable in dataflow.c at scope bug_noninitvar().

	Event	File	Scope
1	Declaration of variable 'value'	dataflow.c	bug_noninitvar()
2	Not entering if statement (if-condition false)	dataflow.c	bug_noninitvar()
3	Non-initialized variable	dataflow.c	bug_noninitvar()

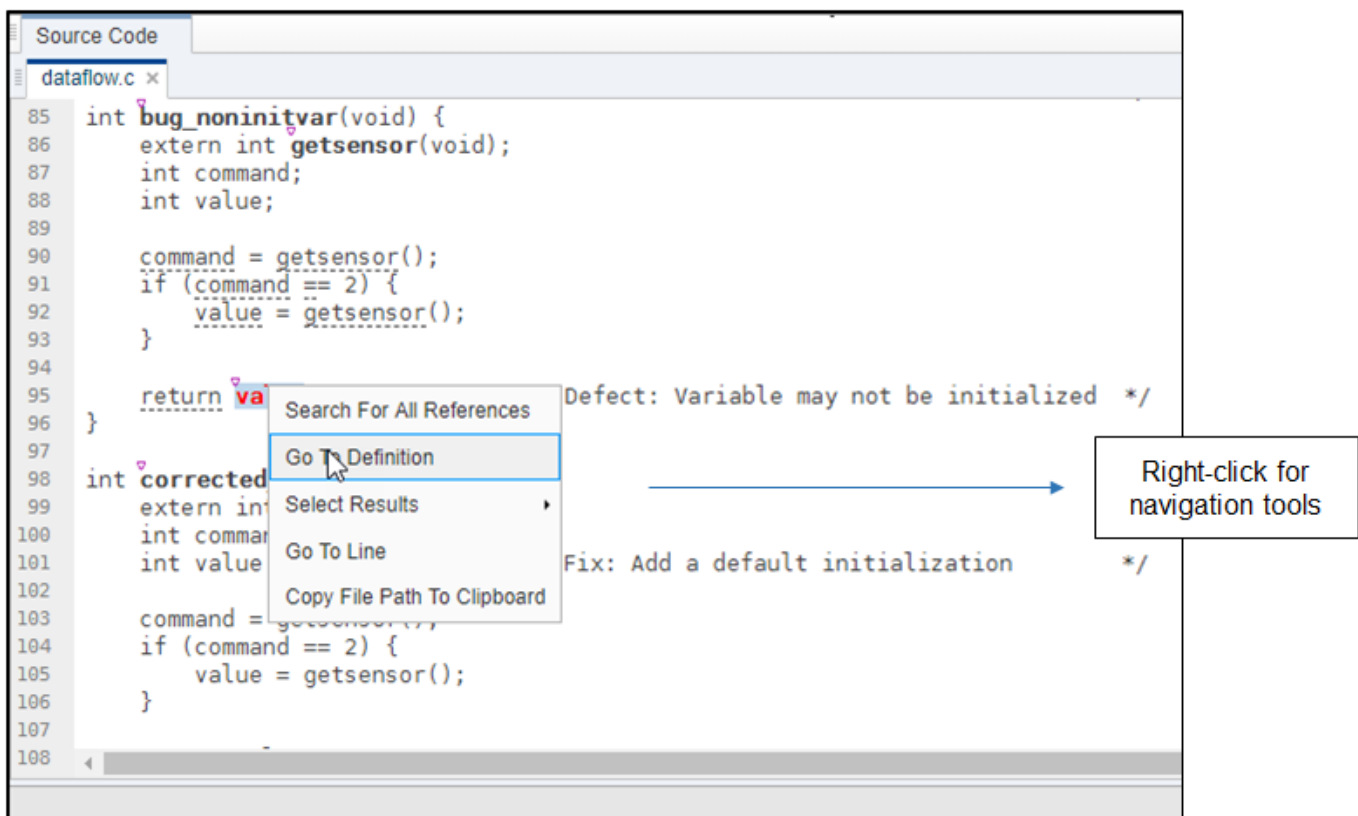
In the above event traceback, this sequence is shown:

- 1 A variable `value` is declared.
- 2 The execution path bypasses an `if` statement. This information might be relevant if the variable is initialized inside the `if` block.
- 3 Location of the current defect: **Non-initialized variable**

Typically, the traceback shows major points in the control flow: entering or bypassing conditional statements or loops, entering a function, and so on. For specific defects, the traceback shows other kinds of events relevant to the defect. For instance, for a **Declaration mismatch** defect, the traceback shows the two locations with conflicting declarations.

Create Your Own Navigation Path

If the event traceback is not available, use other navigation tools to trace your own path through the code.




Before you begin navigating through pathways in your code, ask the question: What am I looking for? Based on your answer, choose the appropriate navigation tool. For instance:

- To investigate a **Non-initialized variable** defect, you might want to make sure that the variable is not initialized at all. To look for previous instances of the variable, on the **Source Code** pane, right-click the variable and select **Search For All References**. This option lists only instances of a specific variable and not other variables with the same name in other scopes.
- To investigate a violation of **MISRA C:2012 Rule 17.7**:

The value returned by a function having non-void return type shall be used.

you might want to navigate from a function call to the function definition. Right-click the function and select **Go To Definition**.

After you navigate away from the current result, use the  icon on the **Result Details** pane to come back.

To select a different result from the **Source Code** pane, Ctrl-click the result or right-click and select **Select Results At This Location**. The **Results Details** pane updates but the result you select is not highlighted in the **Results List** pane. Clicking a result in the **Results List** updates the **Results Details** and **Source Code** panes.

See Also

More About

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 2-2
- “Filter and Sort Results in Polyspace Access Web Interface” on page 3-2

Investigate the Cause of Empty Results List

This topic shows how to interpret an empty results list in the Polyspace Access web interface. To see how to interpret a similar empty list in the user interface of the Polyspace desktop products, see “Investigate the Cause of Empty Results List” (Polyspace Bug Finder).

When you review results from a Polyspace Bug Finder or Polyspace Bug Finder Server™ analysis, the **Results List** pane can be empty or it can display this message:

No results available for currently selected filters,
or no results available for the selected project.

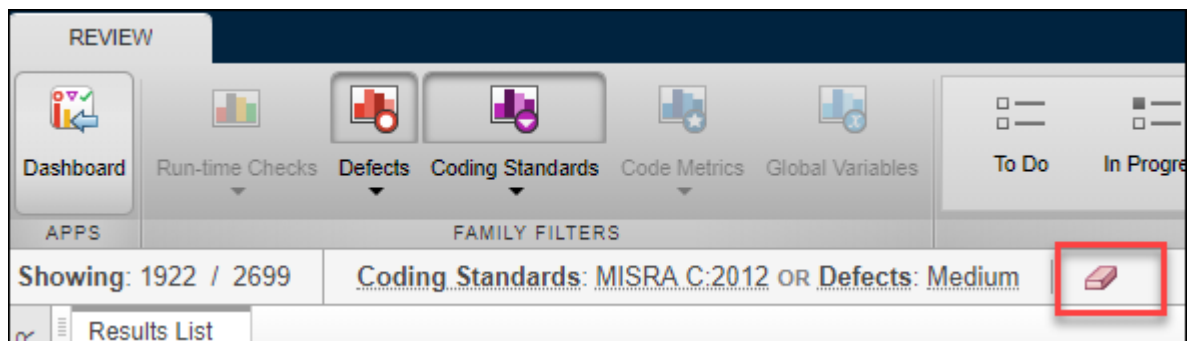
The message can indicate that your code has no defect or coding rule violation. However, before you reach this conclusion:

- 1 Open the **Run Log** pane by going to **Layout > Show/Hide View**.
- 2 Maximize the pane by double-clicking the **Run Log** tab, then use CTRL - F to check for the following.
 - Did all your source files compile?
Search for Failed compilation

If a file does not compile, Bug Finder can return some results, but only files with no compilation errors are fully analyzed.
 - Did you include all your source files in your project?
Search for verifying sources ...

Make sure that all the files that you want to analyze are listed under this message.
- 3 Open the **Configuration Settings** pane by going to **Layout > Show/Hide View**, then:
 - Verify that the appropriate options are activated to check for coding standards violations and to compute code metrics.
 - Check if the `-fast-analysis` option is activated. Bug Finder checks for only a subset of defects and coding rules in fast analysis mode.
 - Click **Checkers configuration** to see a list all the defects and coding rules checkers selected for this analysis.
- 4 Check whether you are applying any filters to the results.

To see which filters you are applying to the results, see the filter bar below the **FAMILY FILTERS** section of the toolbar. To clear all applied filters, click the eraser icon.



If you review results for an analysis you did not configure, discuss the possible causes of an empty results list with the project build master. If you use `polyspace-configure` as part of your analysis workflow, the **Run Log** and **Configuration Settings** panes might not contain all the analysis configuration parameters. For more information on analysis options and project configuration, see the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server.

See Also

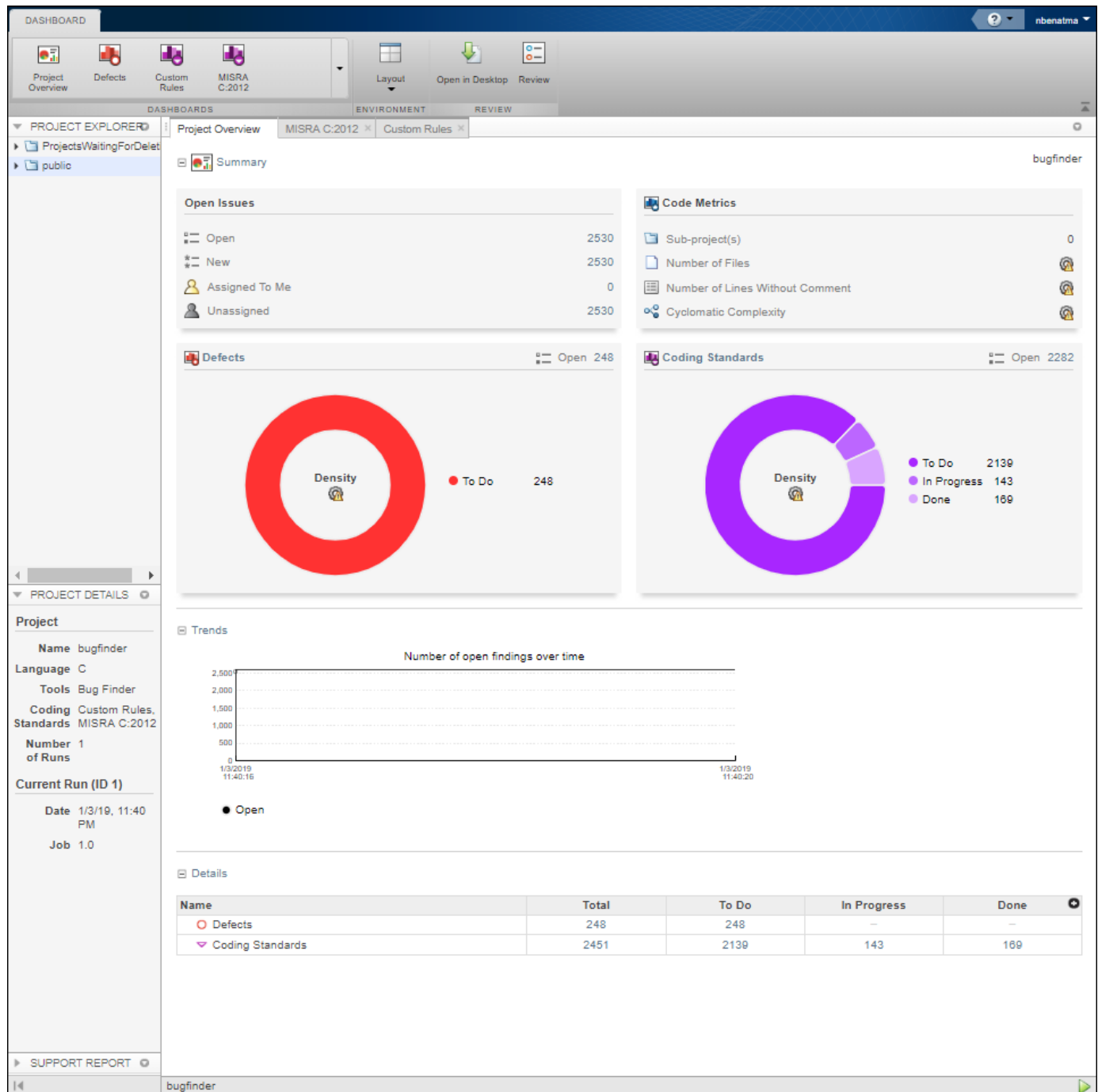
More About

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 2-2

Dashboard

The **Dashboard** perspective provides an overview of the analysis results in graphical format, with clickable fields that let you drill down into your findings by project, file, or category.

When you upload an analysis run to the Polyspace Access database, the **DASHBOARD** updates to display the statistics for the latest run.



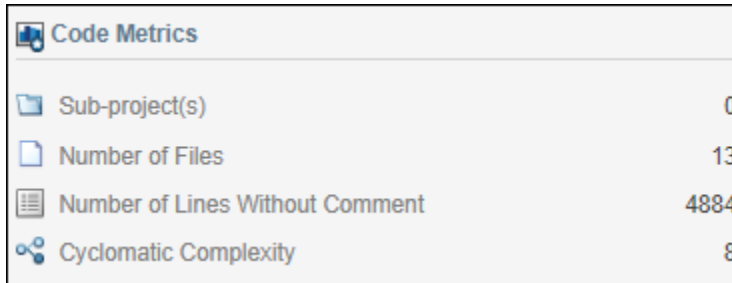
In this perspective, you can open additional dashboards to get a snapshot of the quality of your code. You can see a project overview, or an overview for a family of findings. You can also see an aggregate of statistics for multiple projects under the same folder.

You can also perform the following actions on this pane:

- Select elements on the graphs to filter results from the **Results List** pane. See “Filter and Sort Results in Polyspace Access Web Interface” on page 3-2.
- Open the current project findings in the Polyspace desktop interface.
- Manage projects and user authorizations. See “Manage Project Permissions”.

Code Metrics Dashboard

To view the code complexity metrics that Polyspace computes, use the **Code Metrics** dashboard. See “Code Metrics”. Only when you use the option `Calculate code metrics (-code-metrics)` does Polyspace compute the code complexity metrics during analysis. For more information on analysis options, see the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server.



Code Metrics	
Sub-project(s)	0
Number of Files	13
Number of Lines Without Comment	4884
Cyclomatic Complexity	8

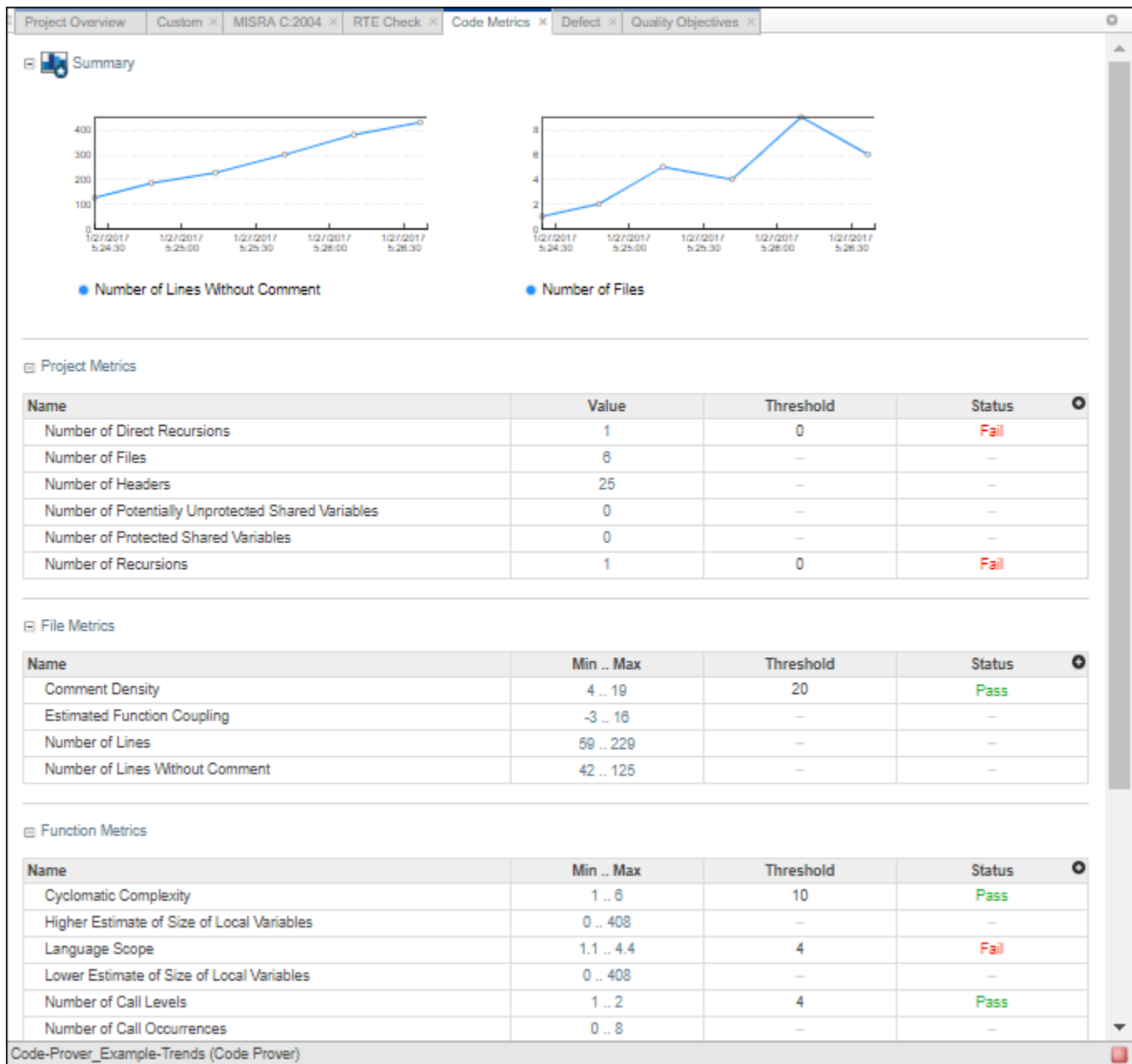
In the **Project Explorer**, select a project. Use the **Code Metrics** card in the **Project Overview** dashboard to get a quick overview of these code metrics:

- Number of Files
- Number of Lines Without Comment
- Cyclomatic Complexity

If you select a folder in the **Project Explorer**, you see the number of **Sub-project(s)** in that folder and an aggregate of the metrics for all the subprojects.

To open the **Code Metrics** dashboard, click the **Code Metrics** icon in the **DASHBOARD** section of the toolstrip. Or, click **Code Metrics** on the card in the **Project Overview** dashboard.

1 Interpret Polyspace Bug Finder Results



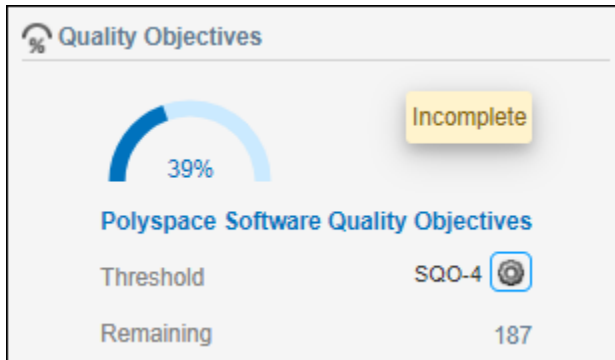
In the **Summary** section, you see trend charts of the **Number of lines Without Comment** and **Number of Files** for the project.

The other sections of the dashboard display tables with the computed value or range of the different project, file, and function metrics. When applicable, the table shows the predefined threshold and pass/fail status for the corresponding code metric. For a list of code complexity metrics thresholds, see “HIS Code Complexity Metrics” on page 1-77. If you select a folder in the **Project Explorer**, the tables in the **Code Metrics** dashboard do not show the threshold or pass/fail status. The value or range of the metrics are aggregate of all subprojects in the selected folder. To drill down to a project from this aggregate view, expand a table row and click the project name.

To improve your code quality, use the pass/fail status to identify and lower metrics values that exceeds a threshold. For instance, if the **Number of Called Functions** range exceeds the predefined threshold, click the range in the **Min..Max** column to open the **Results List** for the computed **Number of Called Functions** metric. Review the results that exceed the metric threshold. If 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.

Quality Objectives Dashboard

To monitor the quality of your code against predefined on page 1-38 software quality thresholds or user-defined on page 1-15 thresholds, use the **Quality Objectives** dashboard.



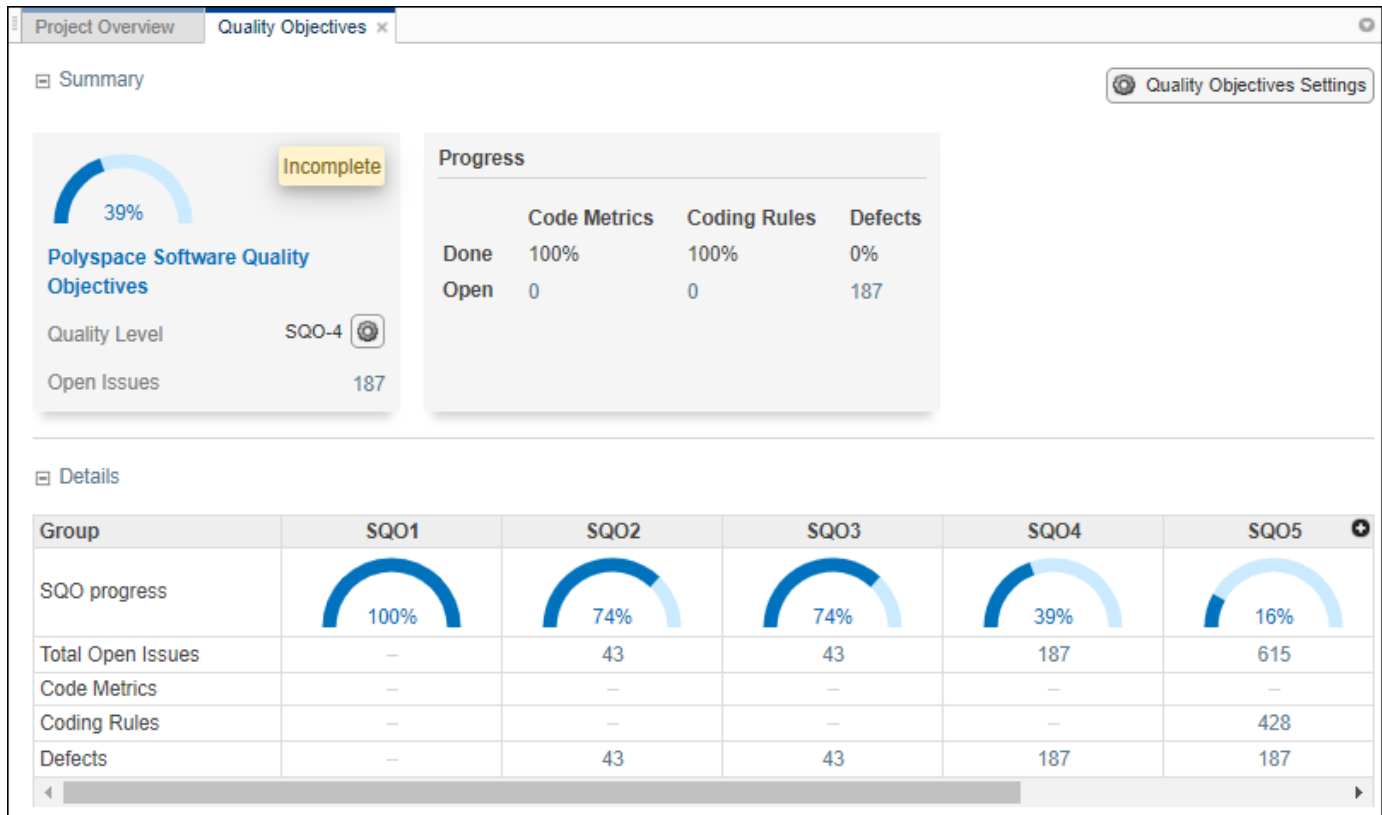
In the **Project Overview** dashboard, use the **Quality Objectives** card to get a quick overview of your progress in achieving a quality objective threshold. The card shows:

- The percentage of findings already addressed to achieve the selected threshold.
- These labels:
 - **Pass:** All findings for this threshold have been addressed.
 - **In progress:** Some findings for this threshold are still open. A finding is open if it has a review status of *Unreviewed*, *To fix*, *To investigate*, or *Other*.
 - **Incomplete:** Some checkers required for this threshold were not activated in the analysis. For instance, if a threshold requires that you address all Bug Finder defects, but the analysis includes only **Numerical** defects, the level is incomplete, even if you address all findings. To see a list of checkers you must activate, click **Incomplete**.
 - **Not computed:** No quality objective results were computed.
- The name of the quality objectives definition currently assigned to the project. In the previous card, the **Polyspace Software Quality Objectives** definition is assigned to the project.
- The assigned **Threshold**. To select a different threshold or quality objectives definition, click the gear icon. You must be an **Administrator** or project **Owner** to assign quality objective definitions or thresholds to a project. You can also assign quality objectives by right-clicking a project in the **Project Explorer**.
- The **Remaining** number of findings that you need to address to reach the threshold. Click this number to open the **REVIEW** perspective and see these findings in the **Results List**.

For a more comprehensive view, open the **Quality Objectives** dashboard. In the **Summary** section, click the gear icon on the **Quality Level** line to pick a threshold and see the remaining open issues, including a breakdown for each category, such as code metrics or coding rules.

In this **Quality Objectives** dashboard, 39% of the findings required to achieve threshold SQQ4 have been addressed. There are 187 open issues, which are all **Defects**.

This table shows the current progress of code quality for all quality objective thresholds. To view the **Results List** for a set of open issues, click the corresponding value in the table.

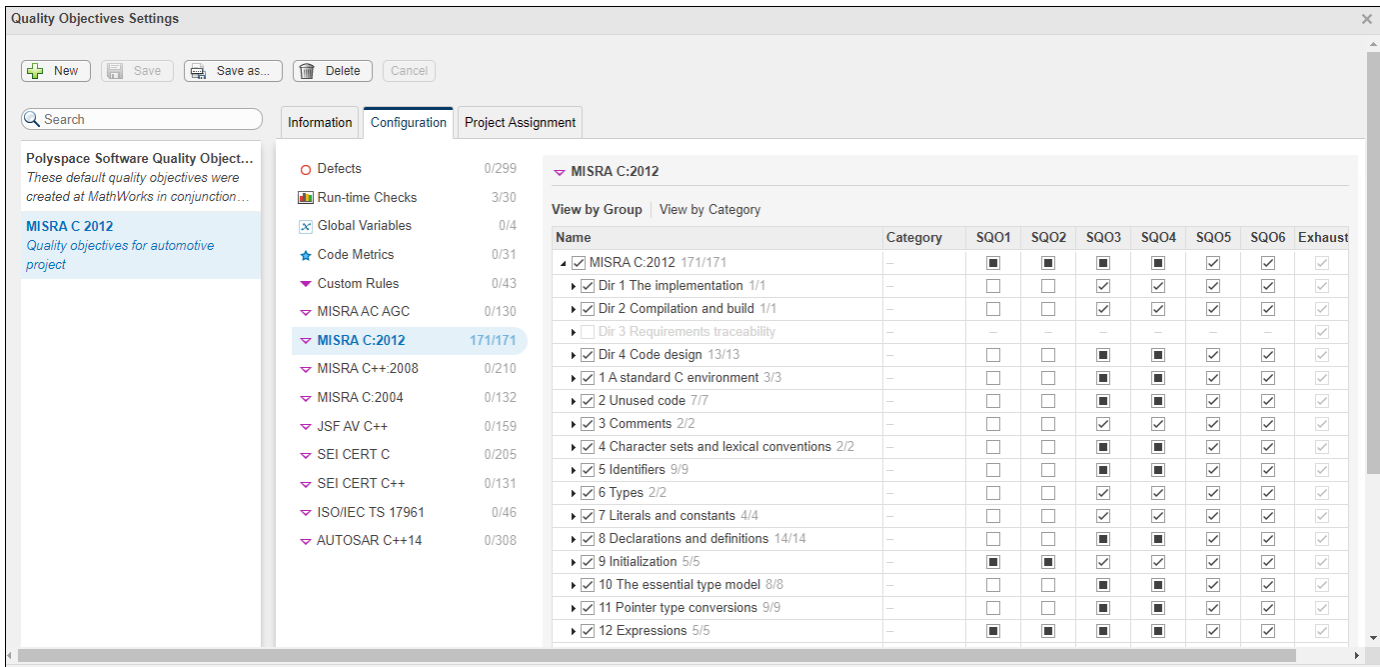


Customize Software Quality Objectives

To customize the thresholds that you use as pass/fail criteria to track the quality of your code, create or edit quality objective definitions and apply these definitions to specific projects. For instance, you might have a project where you want to check the quality of your code against only the MISRA C[®]:2012 coding standard.

On the **Quality Objectives** dashboard, click **Quality Objectives Settings**. You must have the role of **Administrator** or **Owner** to customize the quality objective settings. Users who have the role of **Contributor** have a read-only view of the quality objective settings. You cannot edit the **Polyspace Software Quality Objectives** definition.

1 Interpret Polyspace Bug Finder Results



Create or Edit Quality Objectives Definition

To create a quality objectives definition, click **New**, and enter a name for the new definition. After you assign this definition to a project, the name of the definition is displayed on the **Quality Objectives** card and the summary section of the **Quality Objectives** dashboard for the project. You can optionally provide a description for the quality objectives definition and for the different SQA levels of that definition. Go to the **Information** tab to view or make additional edits to the descriptions.

To edit the thresholds selection, on the **Configuration** tab, click a findings family, for instance MISRA C:2004, and then select a node or expand the node to select individual results. For each family of results, you can view the nodes by group or by category when available.

When you select nodes in the leftmost part of the table:

- indicates that all entries under the node are enabled.
- indicates that some entries under the node are not enabled.

For the quality objective thresholds under the SQA columns:

- indicates that all the entries that are enabled under the node on that row apply to this threshold.
- indicates that some of the entries that are enabled under the node on that row do not apply to this threshold.

	Category	SQO1	SQO2	SQO3	SQO4	SQO5	SQO6	Exhaus
▾ <input checked="" type="checkbox"/> MISRA C:2004 52/131	–	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
▸ <input type="checkbox"/> 1 Environment 0/1	–	–	–	–	–	–	–	<input checked="" type="checkbox"/>
▾ <input checked="" type="checkbox"/> 2 Language extensions 2/3	–	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="checkbox"/> 2.1 Assembly language shall be encaps...	Required	–	–	–	–	–	–	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> 2.2 source code shall only use /* ... */ st...	Required	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> 2.3 The character sequence /* shall not ...	Required	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

For example, in the previous figure, the **Language extensions** node is expanded. The check box next to the node is partially filled because rule 2.1 is not enabled. For the thresholds, all the rules that are enabled under the node apply to thresholds SQO5 and SQO6. Rule 2.2 does not apply to SQO4, which is why the check box for SQO4 is partially filled.

For **Run-time Checks**, customize the percentage of findings that you must address or justify for each threshold. Enter a value between 0 and 100. To disable the selection, leave the entry blank.

For **Code Metrics**, customize the value of the different metrics for each threshold. To disable the selection, leave the entry blank.

When you make a selection for a threshold, all higher thresholds inherit that selection. For instance, if you select a coding rule for SQO3, the rule is also selected for SQO4, SQO5, and SQO6. By default, when you first enable a node or result, it applies only to SQO6.

To save your changes, click **Save** or **Save as** to save your edits in a new quality objectives definition.

The quality objectives statistics for a project are recalculated when:

- You upload a new run for the project.
- You select a finding and make a change to any of the fields in the **Result Details** pane.

Tip When the **Quality Objectives** settings and the calculated statistics for a project are out of sync, the **Quality Objectives** dashboard displays a warning .

Assign Quality Objectives Definition

To assign a quality objectives definition or level to a project, right-click a project in the **Project Explorer** or click the gear icon on the **Quality Objectives** card or dashboard. Before making changes to the quality objectives level or definition for a project, make sure that you inform all Polyspace Access contributors to that project.

By default, the first time you upload results to a new project, Polyspace Access assigns the **Polyspace Software Quality Objectives** to that project. To view which projects a quality objectives definition is assigned to, go to the **Project Assignment** tab in the **Quality Objectives Settings**. If you delete a quality objectives definition, Polyspace Access assigns the **Polyspace Software Quality Objectives** to all the projects to which the deleted definition was assigned.



See Also

More About

- “Bug Finder Quality Objectives” on page 1-38
- “Code Metrics”


Call Hierarchy

The **Call Hierarchy** pane displays the call tree of functions in the source code.

For each function `foo`, the **Call Hierarchy** pane lists the functions and tasks that call `foo` (callers) and those called by `foo` (callees). The callers are indicated by . The callees are indicated by . The **Call Hierarchy** pane lists direct function calls and indirect calls through function pointers.

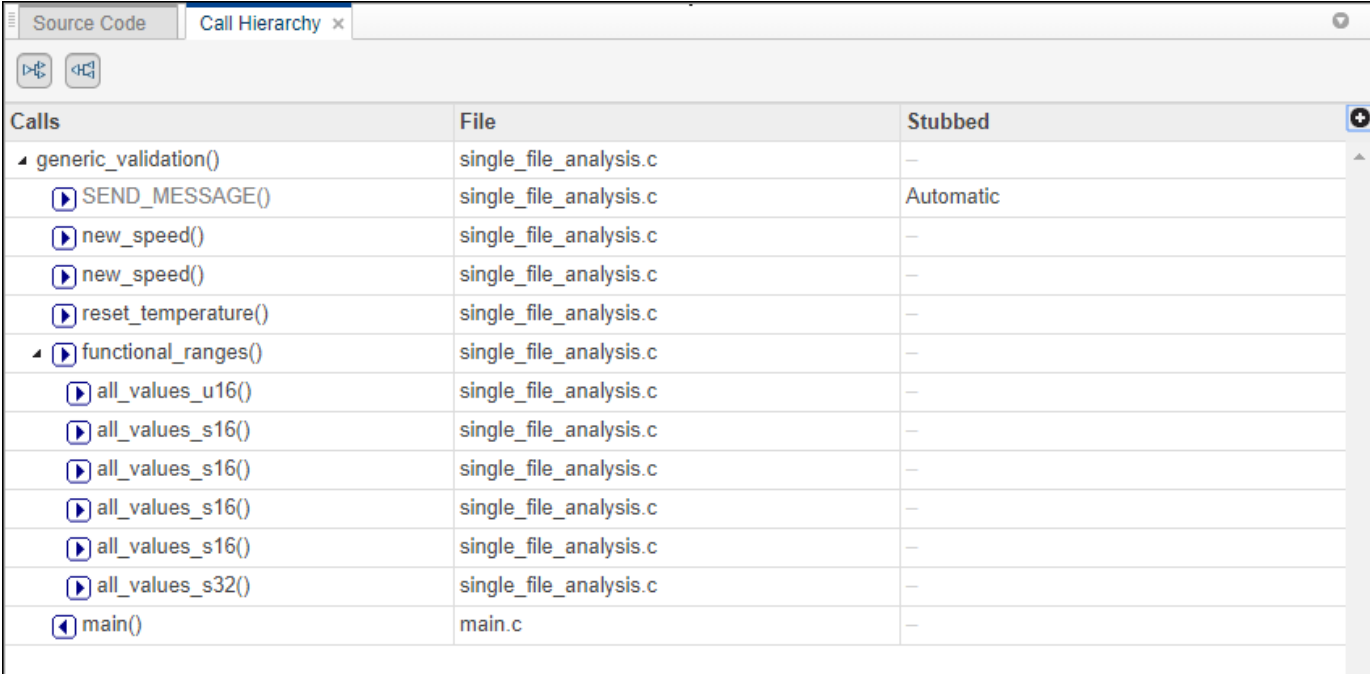
Note For Polyspace Bug Finder Access findings, you might not see all callers or callees of a function, especially for calls through function pointers and dead code.

For instance, Polyspace Bug Finder Access does not display the functions registered with `at_exit()` and `at_quick_exit()`, and called by `exit()` and `quick_exit()` respectively.

You open the **Call Hierarchy** pane by using the  icon in your **Results Details** pane, or by going to **Layout > Show/Hide View**.

To update the pane, click a defect on the **Results List** or CTRL-click a result in the **Source Code** pane. You see the function containing the defect with its callers and callees.

In this example, the **Call Hierarchy** pane displays the function `generic_validation`, and with its callers and callees.



Calls	File	Stubbed
<ul style="list-style-type: none"> generic_validation() <ul style="list-style-type: none"> SEND_MESSAGE() (Automatic) new_speed() new_speed() reset_temperature() functional_ranges() <ul style="list-style-type: none"> all_values_u16() all_values_s16() all_values_s16() all_values_s16() all_values_s16() all_values_s32() main() (main.c) 	<ul style="list-style-type: none"> single_file_analysis.c single_file_analysis.c single_file_analysis.c single_file_analysis.c single_file_analysis.c single_file_analysis.c single_file_analysis.c single_file_analysis.c single_file_analysis.c single_file_analysis.c single_file_analysis.c single_file_analysis.c single_file_analysis.c main.c 	<ul style="list-style-type: none"> - Automatic - - - - - - - - - - - -

Tip To navigate to the call location in the source code, select a caller or callee name

In the **Call Hierarchy** pane, you can perform these actions:

- **Show/Hide Callers and Callees**

Customize the view to display callers only or callees only. Show or hide callers and callees by clicking this button



- **Navigate Call Hierarchy**

You can navigate the call hierarchy in your source code. For a function, double-click a caller or callee name to navigate to the caller or callee definition in the source code.

- **Determine If Function Is Stubbed**

From the **Stubbed** column, you can determine if a function is stubbed. The entries in the column show why a function was stubbed.

- **Automatic:** Polyspace cannot find the function definition. For instance, you did not provide the file containing the definition.
- **Std library:** The function is a standard library function. You do not provide the function definition explicitly in your Polyspace project.
- **Mapped to std library:** You map the function to a standard library function by using the option `-code-behavior-specifications`. For more information on analysis options, see the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server.

Configuration Settings

The **Configuration Settings** pane displays all the analysis options that were passed to the Polyspace analysis engine to generate the currently selected findings. These options include the options that the user specifies and the options that are enabled by default.

You open the **Configuration Settings** pane by going to **Layout > Show/Hide View**.

Options	Value
-author	MathWorks
-checkers	BAD_PLAIN_CHAR_USE, BITWISE_NEG, FLOAT_ABSORPTION, FLOAT_CONV_OVFL, FLOAT_OVFL, FLOAT_STD_LIB, FLOAT_ZERO_DIV, INT_CONSTANT_OVFL, INT_CONV_OVFL, INT_OVFL, INT_PRECISION_EXCEEDED, INT_STD_LIB, INT_TO_FLOAT_PRECISION_LOSS, INT_ZERO_DIV, INVALID_OPERATION_ON_BOOLEAN, SHIFT_NEG, SHIFT_OVFL, SIGN_CHANGE, UINT_CONSTANT_OVFL, UINT_CONV_OVFL, UINT_OVFL
-compiler	gnu4.6
-critical-section-begin	BEGIN_CRITICAL_SECTION:Cs10, acquire_sensor:Cs11, acquire_printer:Cs12, acquire_sensor2:Cs13, acquire_printer2:Cs14
-critical-section-end	END_CRITICAL_SECTION:Cs10, release_sensor:Cs11, release_printer:Cs12, release_sensor2:Cs13, release_printer2:Cs14
-date	08/12/2019
-do-not-generate-results-for	all-headers
-dos	true
-entry-points	bug_datarace_task1, bug_datarace_task2, bug_datarace_task3, bug_datarace_task4, bug_deadlock_task1, bug_deadlock_task2, bug_doublelock_task, bug_doubleunlock_task, bug_badlock_task, bug_badunlock_task, bug_dataracestdlib_task1, bug_dataracestdlib_task2, bug_destroylocked_task, corrected_datarace_task1, corrected_datarace_task2, corrected_datarace_task3, corrected_datarace_task4, corrected_deadlock_task1, corrected_deadlock_task2, corrected_doublelock_task, corrected_doubleunlock_task, corrected_badlock_task, corrected_badunlock_task, corrected_dataracestdlib_task1, corrected_dataracestdlib_task2, corrected_destroylocked_task
-lang	C
-misra3	mandatory
-prog	Bug_Finder_Example
-results-dir	D:\Polyspace\Bug_Finder_Example\BF_Result_1
-target	x86_64
-verif-version	1.0

Click **Checkers configuration** to see which checkers are enabled for:



- “Defects”.
- “Coding Standards”, for instance MISRA C: 2012.
- “Custom Coding Rules”.

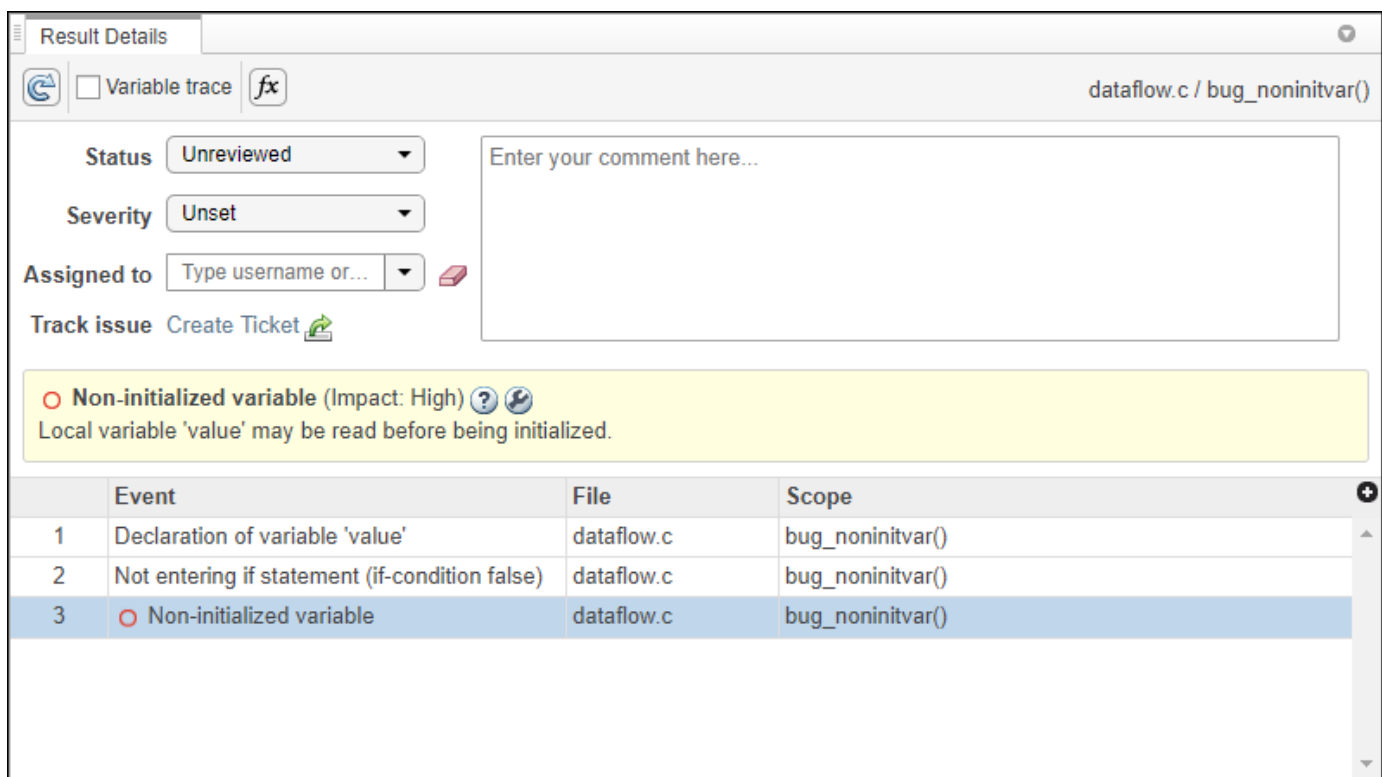
The **Checkers configuration** is not available for a Code Prover project if no coding standard or custom coding rules are enabled.

Result Details

The **Result Details** pane contains comprehensive information about a specific defect. To see this information, on the **Results List** pane, select the defect.

- The top right corner shows the file and function containing the defect, in the format *file_name/function_name*.
- The yellow box contains the name of the defect with an explanation of why the defect occurs.

The  button allows you to access documentation for the defect. When available, click the  icon to see fix suggestions for the defect.




Result Details dataflow.c / bug_noninitvar()



Variable trace fx

Status: Unreviewed Enter your comment here...

Severity: Unset

Assigned to: Type username or... 📄

Track issue [Create Ticket](#) 

○ **Non-initialized variable** (Impact: High)  
Local variable 'value' may be read before being initialized.



	Event	File	Scope
1	Declaration of variable 'value'	dataflow.c	bug_noninitvar()
2	Not entering if statement (if-condition false)	dataflow.c	bug_noninitvar()
3	○ Non-initialized variable	dataflow.c	bug_noninitvar()

On this pane, you can also:

- Assign a **Severity** and **Status** to each check, and enter comments to describe the results of your review.
- Assign a reviewer to the result. A reviewer can filter the **Results List** to only show results that are assigned to him or her.
- Create a ticket in a bug tracking tool such as JIRA. Once you create the ticket the **Results Details** for this defect shows a clickable link to the ticket you created.
- View the event traceback.

The **Event** column lists the sequence of code instructions causing the defect. The **Scope** column lists the function containing the instructions. If the instructions are not in a function, the column lists the file containing the instructions.

The **Variable trace** check box allows you to see an additional set of instructions that are related to the defect.

- Click the  icon to open the “Call Hierarchy” on page 1-19.
- Click the  icon to open the:
 - **Error Call Graph** if the selected finding is a **Run-time Check**.

The pane displays the call sequence that leads to the detected finding. Click a node on the graph to navigate back to the source code.

- **Variable Access Graph** if the selected finding is a **Global variable**.

The pane displays a graphical representation of the access operations on global variables. Click a node on the graph to navigate back to the source code at the location of calling and called functions.

See Also

More About

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 2-2
- “Review History” on page 1-28

Results List

The **Results List** pane lists all results along with their attributes.

For each result, the **Results List** pane contains the result attributes, listed in columns:


Attribute	Description
Family	Group to which the result belongs.
ID	Unique identification number of the result.
Type	Defect or coding rule violation.
Group	Category of the result, for instance: <ul style="list-style-type: none"> For defects: Groups such as static memory, numerical, control flow, concurrency, etc. For coding rule violations: Groups defined by the coding rule standard. <p>For instance, MISRA C: 2012 defines groups related to code constructs such as functions, pointers and arrays, etc.</p>
Check	Result name, for instance: <ul style="list-style-type: none"> For defects: Defect name For coding rule violations: Coding rule number
Information	Result sub-type when available. <ul style="list-style-type: none"> For defects: Impact classification. <p>For coding standards: required or mandatory, rule or recommendation.</p>
Detail	Additional information about a result. The column shows the first line of the Result Details pane. <p>For an example of how to use this column, see the result MISRA C:2012 Dir 1.1.</p>
File	File containing the instruction where the result occurs
Function	Function containing the instruction where the result occurs. If the function is a method of a class, it appears in the format <i>class_name::function_name</i> .
Status	Review status you have assigned to the result. The possible statuses are: <ul style="list-style-type: none"> Unreviewed (default status) To investigate To fix Justified No action planned Not a defect Other

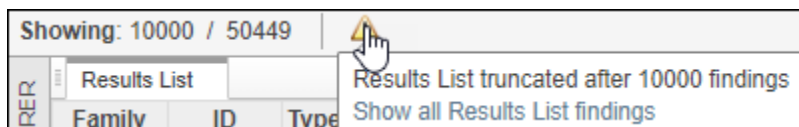
Attribute	Description
Severity	Level of severity you have assigned to the result. The possible levels are: <ul style="list-style-type: none"> • Unset • High • Medium • Low
Assigned to	User name of reviewer assigned to this result.
Ticket Key	When you create a bug tracking tool (BTT) ticket for a result, this field contains the ticket ID. Click the ticket ID in the Results Details to open the ticket in the BTT interface.
Comments	Comments you have entered about the result
Folder	Path to the folder that contains the source file with the result

To show or hide any of the columns, click the  icon in the upper-right of the **Results List** pane, then select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the results.
- Organize your result review using filters in the toolbar or in the context menu. For more information, see “Filter and Sort Results in Polyspace Access Web Interface” on page 3-2.
- Right-click a result to get the URL of the result. When you open this URL in a web browser you get see the **Results List** pane filtered to that one result.

If the **Results List** exceeds 10000 findings, Polyspace Access truncates the list and displays this icon  in the filters bar. To show all findings, see the contextual help of the icon.



The 10000 findings limit is preset and cannot be changed.

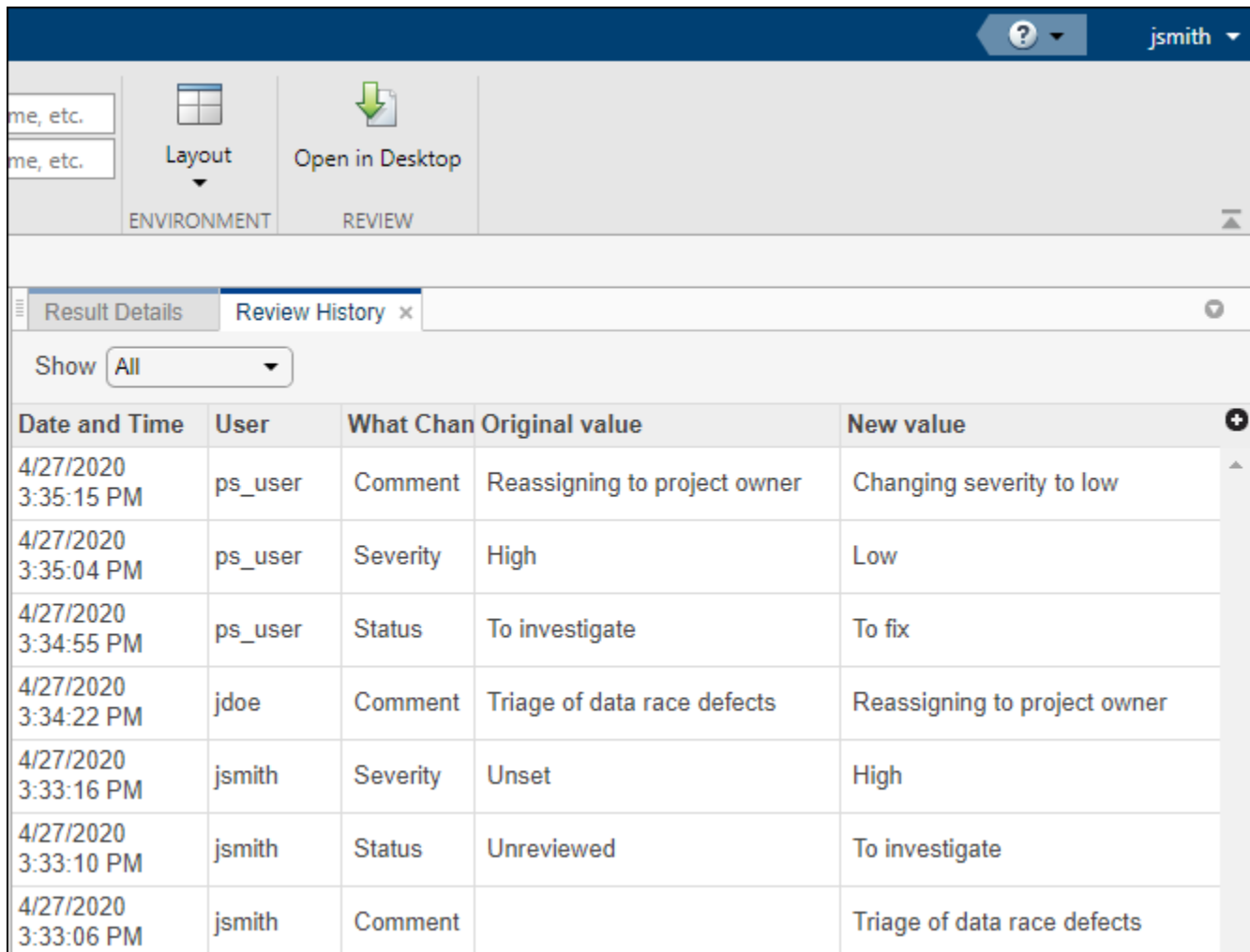
Review History

The **Review History** pane displays changes to the **Status**, **Severity**, or **Comment** for a finding. For each change to these review fields, you see a separate row with:

- The date and time of the change.
- The user name of the user who made the change.
- The review field that changed, for instance **Severity**.
- The original value of the review field.
- The new value of the review field.

All the changes that you make to the review fields of findings in the Polyspace desktop interface are shown in a single row after you upload these findings to Polyspace Access. The **Review History** pane does not display the user name of the user who made these changes.

You open the **Review History** pane by going to **Layout > Show/Hide View**.



Date and Time	User	What Chan	Original value	New value
4/27/2020 3:35:15 PM	ps_user	Comment	Reassigning to project owner	Changing severity to low
4/27/2020 3:35:04 PM	ps_user	Severity	High	Low
4/27/2020 3:34:55 PM	ps_user	Status	To investigate	To fix
4/27/2020 3:34:22 PM	jdoe	Comment	Triage of data race defects	Reassigning to project owner
4/27/2020 3:33:16 PM	jsmith	Severity	Unset	High
4/27/2020 3:33:10 PM	jsmith	Status	Unreviewed	To investigate
4/27/2020 3:33:06 PM	jsmith	Comment		Triage of data race defects

You can display changes for all the review fields, or you can filter for changes by **Status**, **Severity**, and **Comment**.

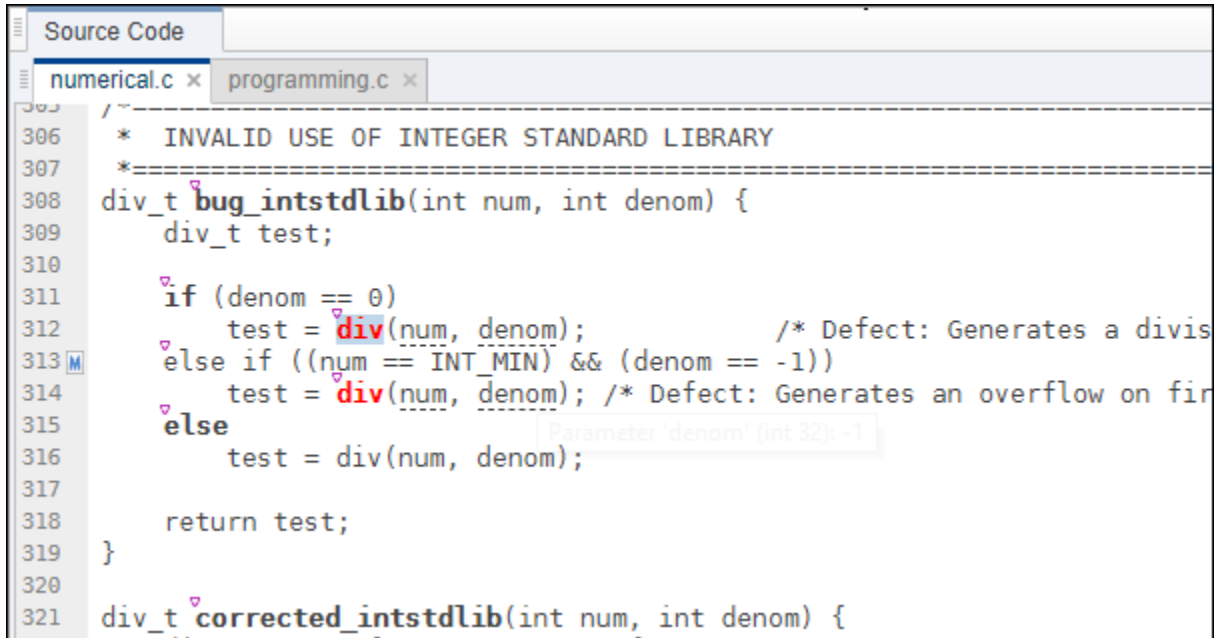
See Also

More About

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 2-2
- “Result Details” on page 1-24

Source Code

The **Source Code** pane shows the source code with the defects colored in red.

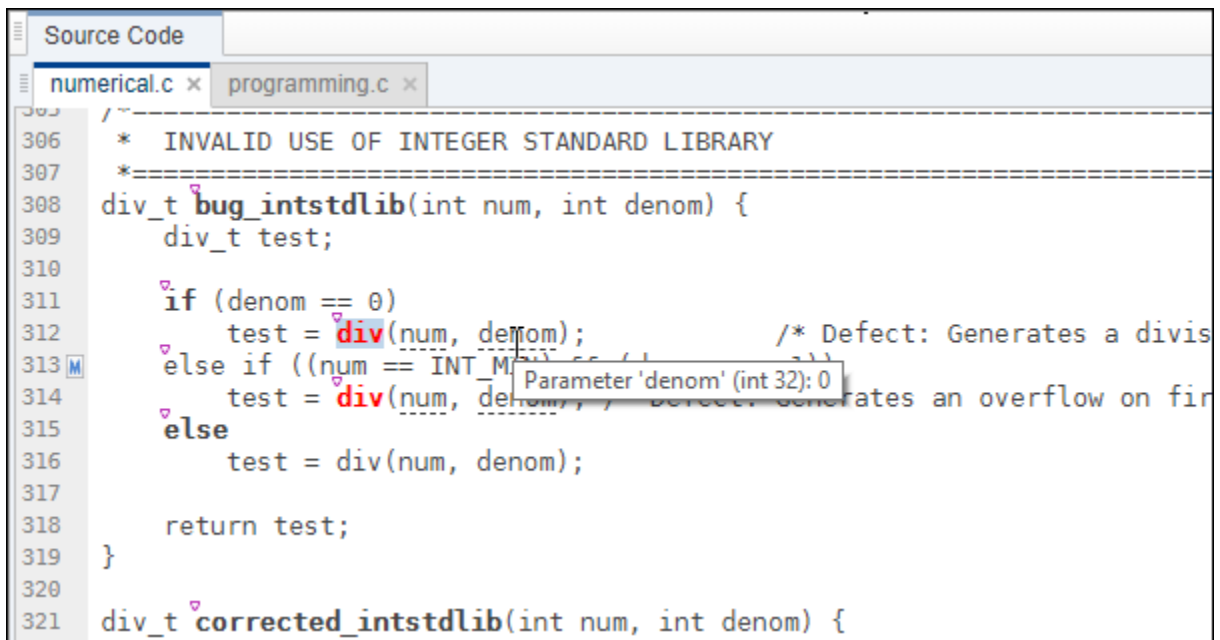


```

Source Code
numerical.c x programming.c x
306  * INVALID USE OF INTEGER STANDARD LIBRARY
307  *-----
308  div_t bug_intstdlib(int num, int denom) {
309      div_t test;
310
311      if (denom == 0)
312          test = div(num, denom);          /* Defect: Generates a divis
313  M else if ((num == INT_MIN) && (denom == -1))
314          test = div(num, denom); /* Defect: Generates an overflow on fir
315      else
316          test = div(num, denom);
317
318      return test;
319  }
320
321  div_t corrected_intstdlib(int num, int denom) {
  
```

Tooltips

Placing your cursor over a result displays a tooltip that provides range information for variables, operands, function parameters, and return values.

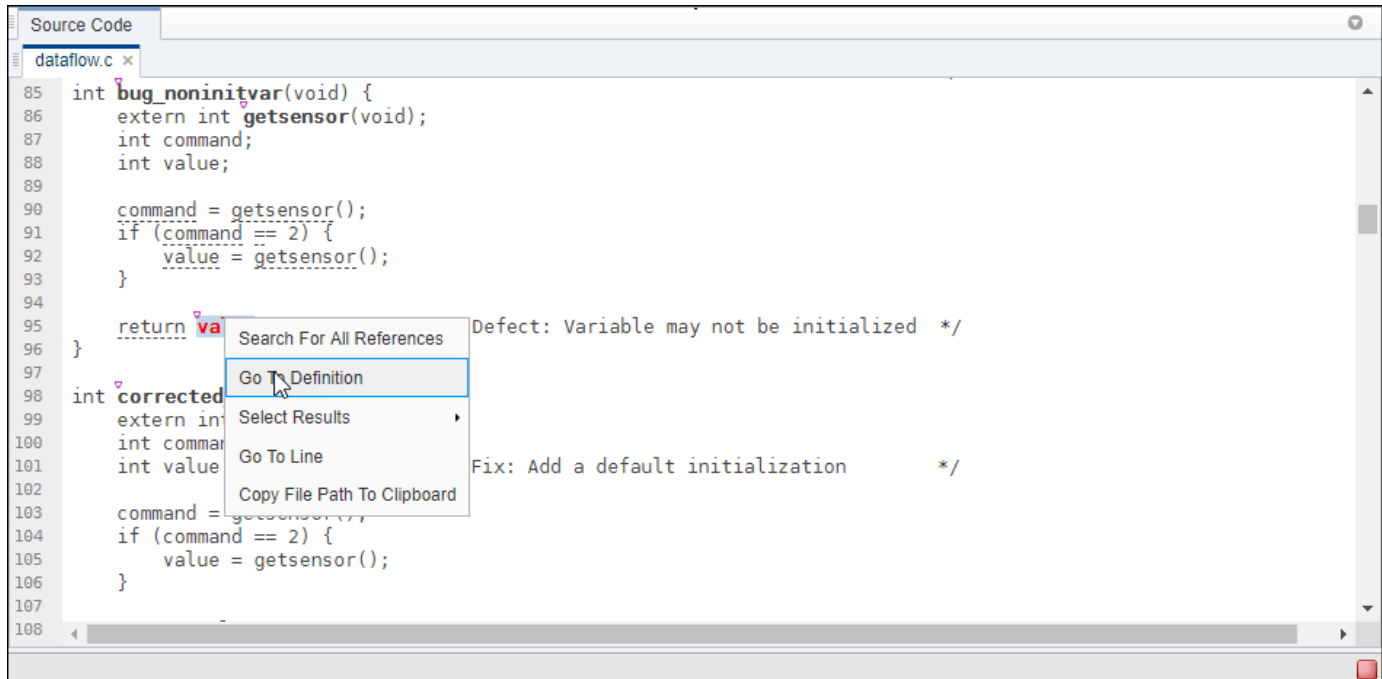


```

Source Code
numerical.c x programming.c x
306  * INVALID USE OF INTEGER STANDARD LIBRARY
307  *-----
308  div_t bug_intstdlib(int num, int denom) {
309      div_t test;
310
311      if (denom == 0)
312          test = div(num, denom);          /* Defect: Generates a divis
313  M else if ((num == INT_MIN) && (denom == -1))
314          test = div(num, denom); /* Defect: Generates an overflow on fir
315      else
316          test = div(num, denom);
317
318      return test;
319  }
320
321  div_t corrected_intstdlib(int num, int denom) {
  
```



Examine Source Code

On the **Source Code** pane, if you right-click a text string, the context menu provides options to examine your code:



For example, if you right-click the variable, you can use the following options to examine and navigate through your code:


- **Search For All References** — List all references in the **Code Search** pane. The software supports this feature for global and local variables, functions, types, and classes.
- **Go To Definition** — Go to the line of code that contains the definition of `i`. The software supports this feature for global and local variables, functions, types, and classes. If a definition is not available to Polyspace, selecting the option takes you to the declaration.
- **Select Results** — Show more information about the selected result in the **Results Details** pane and pin the result in the **Source Code** pane.

After you navigate away from the current result, use the  icon on the **Result Details** pane to come back.

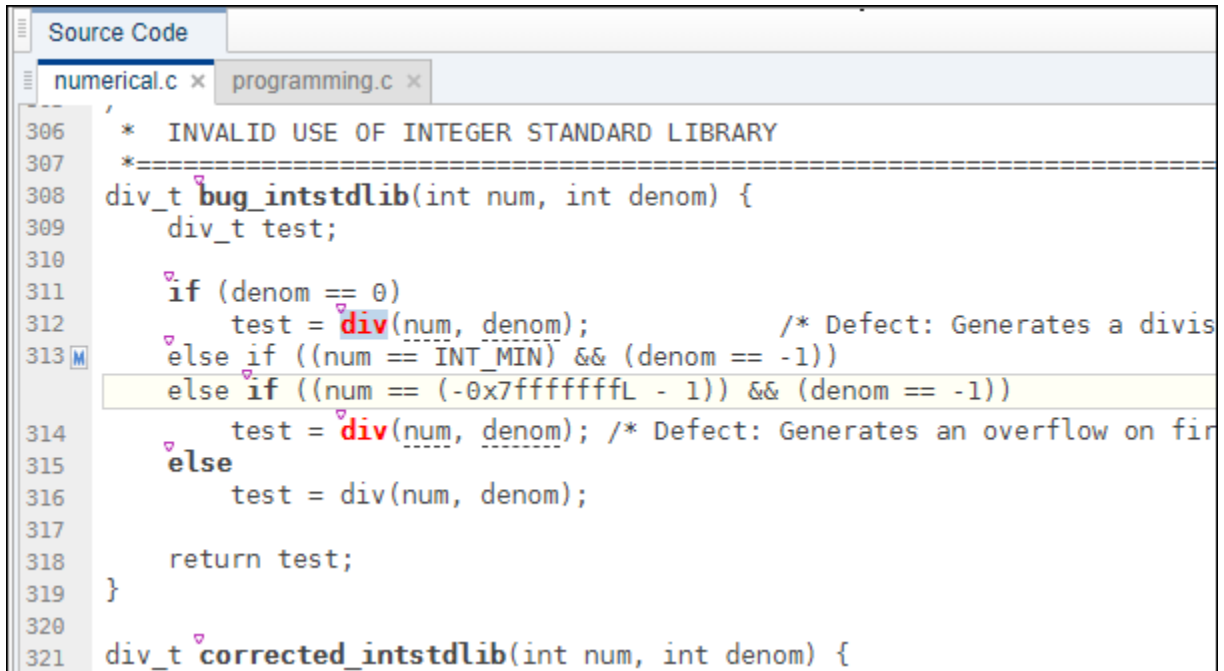
- **Go To Line** — Open the Go to line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.

To search for instances of your selection in the **Current Source File** or in **All Source Files**, double-click your selection before you right-click.

Expand Macros

You can view the contents of source code macros in the source code view. A code information bar displays  icons that identify source code lines with macros.

When you click this icon, the software displays the contents of macros on the next line.



```
Source Code
numerical.c x programming.c x
306 * INVALID USE OF INTEGER STANDARD LIBRARY
307 *=====
308 div_t bug_intstdlib(int num, int denom) {
309     div_t test;
310
311     if (denom == 0)
312         test = div(num, denom); /* Defect: Generates a divis
313 M else if ((num == INT_MIN) && (denom == -1))
314         else if ((num == (-0x7fffffffL - 1)) && (denom == -1))
315             test = div(num, denom); /* Defect: Generates an overflow on fir
316         else
317             test = div(num, denom);
318
319     return test;
320 }
321 div_t corrected_intstdlib(int num, int denom) {
```

To display the normal source code again, click the icon again.

Note

- 1 The **Result Details** pane also allows you to view the contents of a macro if the check you select lies within a macro.
 - 2 You cannot expand OSEK API macros in the **Source Code** pane.
-

View Code Block

On the **Source Code** pane, to highlight a block of code, click either its opening or closing brace. If the brace itself is highlighted, click the brace twice.

```

306  * INVALID USE OF INTEGER STANDARD LIBRARY
307  *-----
308  div_t bug_intstdlib(int num, int denom) {
309      div_t test;
310
311      if (denom == 0)
312          test = div(num, denom); /* Defect: Generates a divis
313  else if ((num == INT_MIN) && (denom == -1))
314          test = div(num, denom); /* Defect: Generates an overflow on fir
315  else
316          test = div(num, denom);
317
318      return test;
319  }
320
321  div_t corrected_intstdlib(int num, int denom) {

```

Navigate from Code to Model

If you run Polyspace on generated code in Simulink® and upload the results to Polyspace Access, you can navigate from the source code in Polyspace Access to blocks in the model.

On the **Source Code** pane in the Polyspace Access web interface, links in code comments show blocks that generate the subsequent lines of code. To see the block in the model:

- Right-click a link and select **Copy MATLAB Command to Highlight Block**.

```

28  /* Real-time model */
29  RT_MODEL_test20a_T test20a_M;
30  RT_MODEL_test20a_T *const test20a_M = &test20a_M;
31
32  /* Model step function */
33  void test20a_step(void)
34  {
35      /* Output: '<Root>/Out1' incorporates:
36       * Gain: '<Root>/Gain'
37       * Inport: '<Root>/In1'
38       */
39      test20a_Y.Out1 = 2;
40  }
41
42  /* Model initialize function */
43  void test20a_initialize(void)

```


This action copies the MATLAB® command required to highlight the block. The command uses the `Simulink.ID.hilite` function.

- In MATLAB editor, paste and run the copied command with the model open.

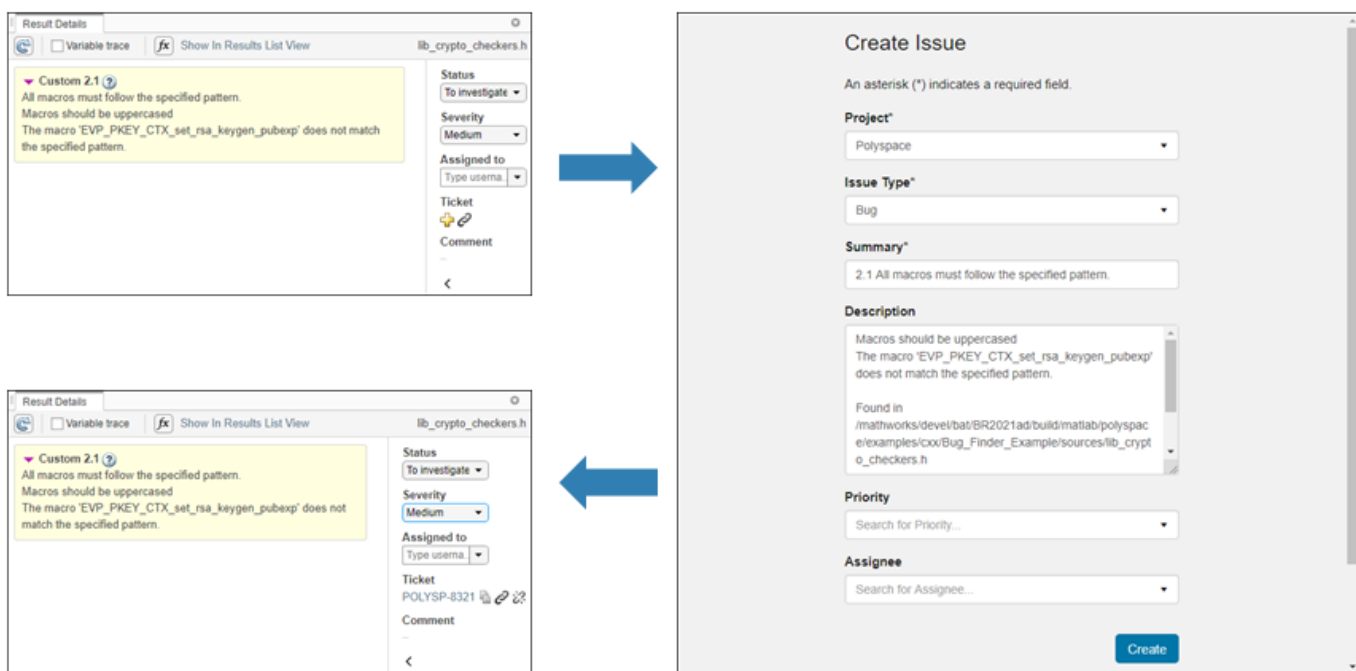
Track Issue in Bug Tracking Tool

If you use a bug tracking tool (BTT) such as Jira Software or Redmine as part of your software development process, you can configure Polyspace Access to create BTT tickets for Polyspace findings and add those tickets to the relevant project in your BTT software. See “Configure Issue Tracker”.

Create a Ticket

To create a BTT ticket, select one or more findings in the **Results list** and, from the **Results Details** pane, click  in Polyspace Access or **Create ticket** in the Polyspace desktop interface. To select multiple findings, press **CTRL** and click the findings.

Note In the desktop interface, you can create a BTT ticket only for results that you open from Polyspace Access.




If you use Jira, you may be prompted to enter your credentials. These credentials might be different from your Polyspace Access credentials.

After you create a BTT ticket, click the link in the **Results Details** pane to open the ticket in the BTT interface and track the progress in resolving the issue. For each finding that you selected when you created the ticket, the **Description** field of the ticket includes a URL to the Polyspace Access **Results List** filtered down to that finding.

Manage Existing Tickets

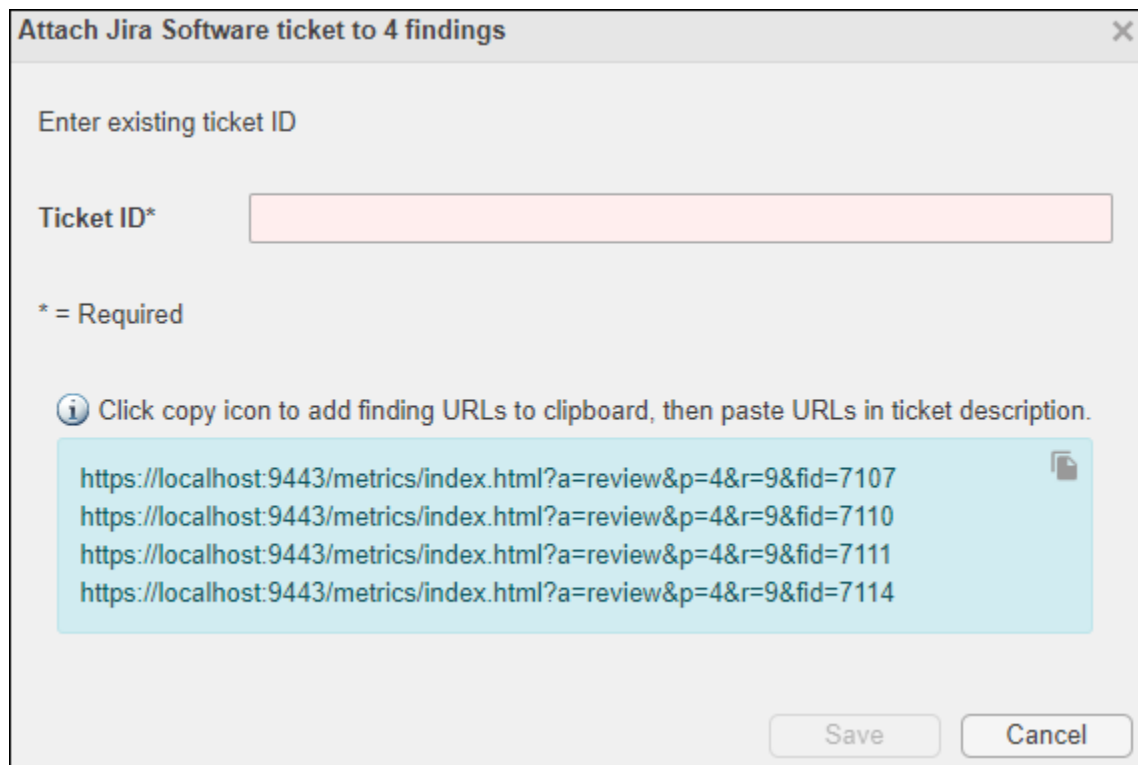
Once you create a BTT ticket, you can attach the ticket to additional findings or detach the ticket from findings associated with the ticket. To attach a ticket to additional findings:

- 1 Select findings in the **Results List** and then click  in the **Result Details**.
- 2 When prompted, enter the **ticket ID** in the dialogue window.

Click the copy icon in the **Result Details** pane of a finding already associated with the ticket to copy the **ticket ID**. The copy icon is not available when you select multiple findings with different ticket IDs. The **ticket ID** is also available in the **Ticket Key** column of the **Results List**.

- 3 Click the copy icon in the dialogue window to copy the findings URL, then click **Save**.
- 4 Click the ticket URL in the **Result Details** to open the ticket in the BTT interface and paste the findings URL you copied into the ticket description field.

You cannot attach more than one ticket to a finding. If a finding is already associated with a ticket, attaching a new ticket overwrites the existing **ticket ID**. This operation does not overwrite the ticket in your BTT. You can see all findings associated with a **ticket ID** by using the **Show only** text filter in the toolstrip.




Attach Jira Software ticket to 4 findings

Enter existing ticket ID


Ticket ID*

* = Required

 Click copy icon to add finding URLs to clipboard, then paste URLs in ticket description.

```
https://localhost:9443/metrics/index.html?a=review&p=4&r=9&fid=7107
https://localhost:9443/metrics/index.html?a=review&p=4&r=9&fid=7110
https://localhost:9443/metrics/index.html?a=review&p=4&r=9&fid=7111
https://localhost:9443/metrics/index.html?a=review&p=4&r=9&fid=7114
```

Save Cancel

To detach a ticket from a finding, select the finding in the **Results List**, then click  in the **Result Details**. The link to the ticket is removed from the **Result Details** pane. This operation does not remove the ticket in your BTT.

Note You cannot manage existing BBT tickets in the Polyspace desktop interface.

Bug Finder Quality Objectives

The Bug Finder Quality Objectives or BF-QOs are a set of thresholds against which you can compare your Bug Finder analysis results. These objectives are adapted from the Polyspace Code Prover™ “Software Quality Objectives” (Polyspace Code Prover Access). You can develop a review process based on the Quality Objectives.

You can use a predefined BF-QO level or define your own. To customize BF-QO levels, see “Customize Software Quality Objectives” on page 1-15.

Following are the predefined quality thresholds specified by each BF-QO.

BF-QO Level 1

Metric	Threshold Value
Comment density of a file	20
Number of paths through a function	80
Number of goto statements	0
Cyclomatic complexity	10
Number of calling functions	5
Number of calls	7
Number of parameters per function	5
Number of instructions per function	50
Number of call levels in a function	4
Number of return statements in a function	1
Language scope, an indicator of the cost of maintaining or changing functions. Calculated as follows: $(N1+N2) / (n1+n2)$ <ul style="list-style-type: none"> • <i>n1</i> — Number of different operators • <i>N1</i> — Total number of operators • <i>n2</i> — Number of different operands • <i>N2</i> — Total number of operands 	4
Number of recursions	0
Number of direct recursions	0

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2004 rules: <ul style="list-style-type: none"> • 5.2 • 8.11, 8.12 • 11.2, 11.3 • 12.12 • 13.3, 13.4, 13.5 • 14.4, 14.7 • 16.1, 16.2, 16.7 • 17.3, 17.4, 17.5, 17.6 • 18.4 • 20.4 	0
Number of unjustified violations of the following MISRA C:2012 rules: <ul style="list-style-type: none"> • 8.8, 8.11, and 8.13 • 11.1, 11.2, 11.4, 11.5, 11.6, and 11.7 • 14.1 and 14.2 • 15.1, 15.2, 15.3, and 15.5 • 17.1 and 17.2 • 18.3, 18.4, 18.5, and 18.6 • 19.2 • 21.3 	0
Number of unjustified violations of the following MISRA® C++ rules: <ul style="list-style-type: none"> • 2-10-2 • 3-1-3, 3-3-2, 3-9-3 • 5-0-15, 5-0-18, 5-0-19, 5-2-8, 5-2-9 • 6-2-2, 6-5-1, 6-5-2, 6-5-3, 6-5-4, 6-6-1, 6-6-2, 6-6-4, 6-6-5 • 7-5-1, 7-5-2, 7-5-4 • 8-4-1 • 9-5-1 • 10-1-2, 10-1-3, 10-3-1, 10-3-2, 10-3-3 • 15-0-3, 15-1-3, 15-3-3, 15-3-5, 15-3-6, 15-3-7, 15-4-1, 15-5-1, 15-5-2 • 18-4-1 	0

BF-QO Level 2 and 3

In addition to all the requirements of BF-QO Level 1, these levels includes the following thresholds:

Metric	Threshold Value
Number of "High Impact Defects" on page 3-11	0

BF-QO Level 4

In addition to all the requirements of BF-QO Level 2 and 3, this level includes the following thresholds:

Metric	Threshold Value
Number of "Medium Impact Defects" on page 3-14	0

BF-QO Level 5

In addition to all the requirements of BF-QO Level 4, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2004 rules: <ul style="list-style-type: none"> • 6.3 • 8.7 • 9.2, 9.3 • 10.3, 10.5 • 11.1, 11.5 • 12.1, 12.2, 12.5, 12.6, 12.9, 12.10 • 13.1, 13.2, 13.6 • 14.8, 14.10 • 15.3 • 16.3, 16.8, 16.9 • 19.4, 19.9, 19.10, 19.11, 19.12 • 20.3 	0
Number of unjustified violations of the following MISRA C:2012 rules: <ul style="list-style-type: none"> • 11.8 • 12.1 and 12.3 • 13.2 and 13.4 • 14.4 • 15.6 and 15.7 • 16.4 and 16.5 • 17.4 • 20.4, 20.6, 20.7, 20.9, and 20.11 	0

Metric	Threshold Value
Number of unjustified violations of the following MISRA C++ rules: <ul style="list-style-type: none"> • 3-4-1, 3-9-2 • 4-5-1 • 5-0-1, 5-0-2, 5-0-7, 5-0-8, 5-0-9, 5-0-10, 5-0-13, 5-2-1, 5-2-2, 5-2-7, 5-2-11, 5-3-3, 5-2-5, 5-2-6, 5-3-2, 5-18-1 • 6-2-1, 6-3-1, 6-4-2, 6-4-6, 6-5-3 • 8-4-3, 8-4-4, 8-5-2, 8-5-3 • 11-0-1 • 12-1-1, 12-8-2 • 16-0-5, 16-0-6, 16-0-7, 16-2-2, 16-3-1 	0

BF-QO Level 6

In addition to all the requirements of BF-QO Level 5, this level includes the following thresholds:

Metric	Threshold Value
Number of "Low Impact Defects" on page 3-18	0

BF-QO Exhaustive

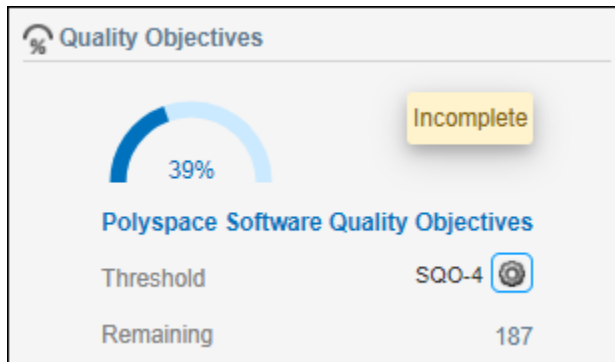
In addition to all the requirements of BF-QO Level 6, this level includes the following thresholds. The thresholds for coding rule violations apply only if you check for coding rule violations.

Metric	Threshold Value
Number of unjustified MISRA C and MISRA C++ coding rule violations	0
Number of unjustified defects	0

Comparing Analysis Results Against Quality Objectives

You can compare your analysis results against SQOs either in the Polyspace Access web interface or the Polyspace user interface.

- In the Polyspace Access web interface, you can first determine whether your project fails to attain a certain Quality Objective threshold by looking at the **Quality Objectives** card on the **Project Overview** dashboard.



The card shows the percentage of results that you have already fixed or justified in order to attain the threshold. Click the number of remaining findings to open those findings in the **Results List**. For a more detailed view of the quality of your code against all quality objectives thresholds, open the **Quality Objectives** dashboard. For more information, see the “Quality Objectives Dashboard” on page 1-14.

You can also generate reports that show the **PASS** or **FAIL** status using the templates `SoftwareQualityObjectives_Summary` and `SoftwareQualityObjectives`. See Bug Finder and Code Prover report (-report-template).

- In the Polyspace user interface, you can use the menu in the **Results List** toolbar to display only those results that you must fix or justify to attain a certain Software Quality Objective.

To activate the SQO options in this menu, select **Tools > Preferences**. On the **Review Scope** tab, select **Include Quality Objectives Scope**.

See Also

Related Examples

- “Filter and Sort Results in Polyspace Access Web Interface” on page 3-2
- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 2-2

Software Quality Objective Subsets (C:2004)

In this section...
“Rules in SQO-Subset1” on page 1-43
“Rules in SQO-Subset2” on page 1-44

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.
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.

Rule number	Description
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
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

Rule number	Description
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
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
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.

Rule number	Description
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

More About

- “Interpret Bug Finder Results in Polyspace Access Web Interface” on page 1-2

Software Quality Objective Subsets (AC AGC)

In this section...
“Rules in SQO-Subset1” on page 1-47
“Rules in SQO-Subset2” on page 1-47

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.

Rule number	Description
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
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 ##

Rule number	Description
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.
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

More About

- “Interpret Bug Finder Results in Polyspace Access Web Interface” on page 1-2

Software Quality Objective Subsets (C:2012)

In this section...
“Guidelines in SQO-Subset1” on page 1-50
“Guidelines in SQO-Subset2” on page 1-51

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
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

Rule	Description
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
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

Rule	Description
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
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

More About

- “Interpret Bug Finder Results in Polyspace Access Web Interface” on page 1-2

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++)

In this section...

“SQO Subset 1 - Direct Impact on Selectivity” on page 1-56

“SQO Subset 2 - Indirect Impact on Selectivity” on page 1-57

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.
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.
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.
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.

MISRA C++ Rule	Description
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.
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.
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.

MISRA C++ Rule	Description
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.
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.

MISRA C++ Rule	Description
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.
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

More About

- “Interpret Bug Finder Results in Polyspace Access Web Interface” on page 1-2

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). For more information on analysis options, see the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server.

Argument	Purpose
single-unit-rules	<p>Check rules that apply only to single translation units.</p> <p>If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the compilation phase.</p>
system-decidable-rules	<p>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.</p> <p>If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the linking phase.</p>

See also “Interpret Bug Finder Results in Polyspace Access Web Interface” on page 1-2.

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 <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.
8.4*	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	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: <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	The value of an expression of floating type shall not be implicitly converted to a different type if <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	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.
10.4	The value of a complex expression of float type may only be cast to narrower floating type.
10.5	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
10.6	The "U" suffix shall be applied to all constants of <code>unsigned</code> types.

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.
16.4*	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-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 <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.
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 <code>sizeof</code> 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 <stdarg.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 <code>union</code> 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

More About

- “Interpret Bug Finder Results in Polyspace Access Web Interface” on page 1-2

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.

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
Number of paths	80
Number of return statements	1

See Also

More About

- “Code Metrics”

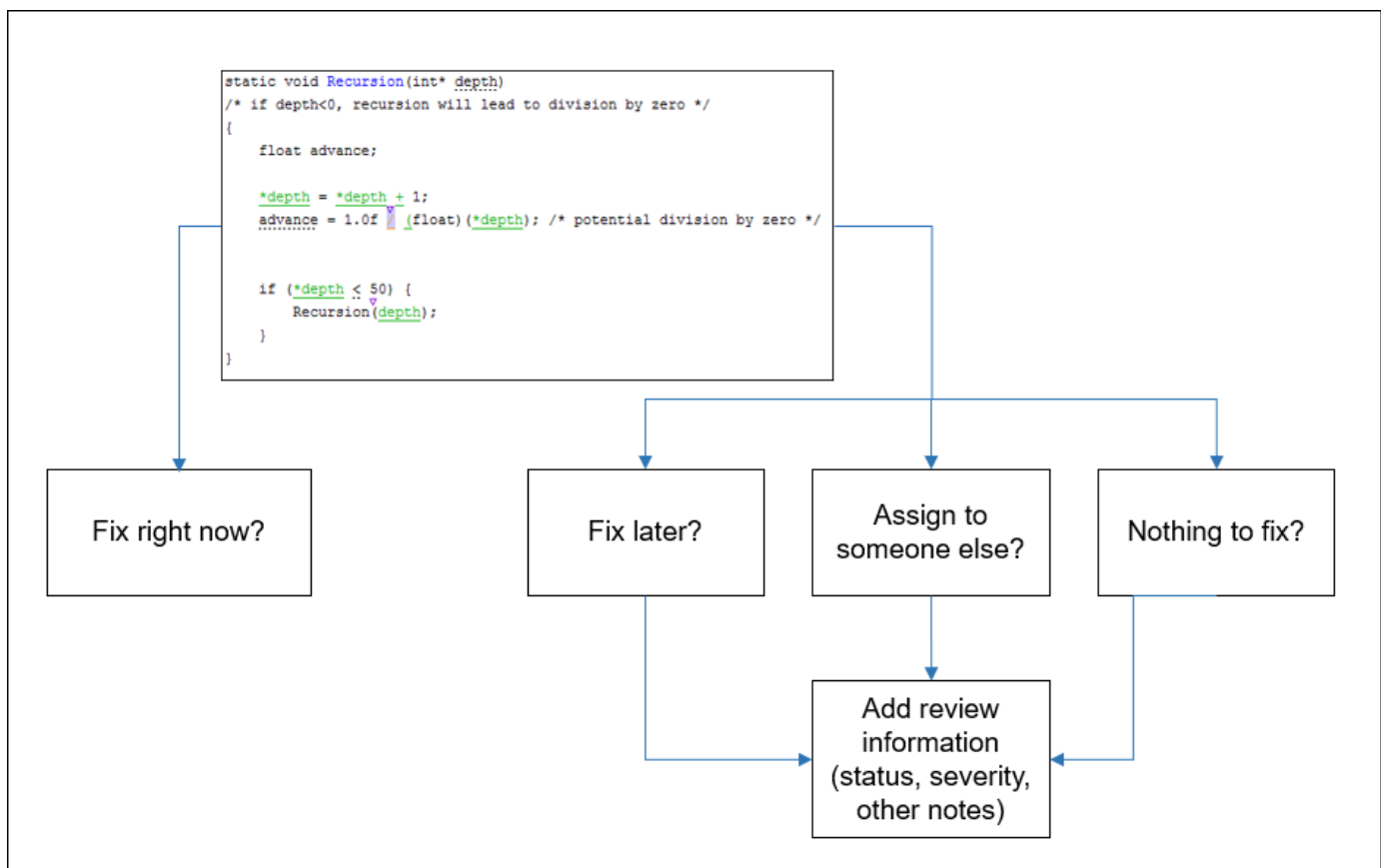
Fix or Comment Polyspace Results

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 2-2
- “Hide Known or Acceptable Polyspace Results” on page 2-5
- “Short Names of Bug Finder Defect Checkers” on page 2-12
- “Short Names of Code Complexity Metrics” on page 2-26
- “Define Custom Annotation Format” on page 2-28
- “Annotation Description Full XML Template” on page 2-36

Address Results in Polyspace Access Through Bug Fixes or Justifications

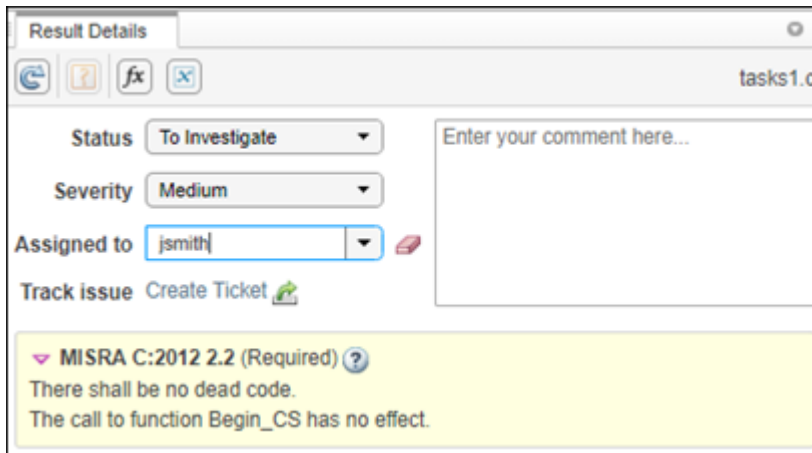
This topic describes how to add review information to Polyspace results in the Polyspace Access web interface. For a similar workflow in the user interface of the Polyspace desktop products, see “Address Polyspace Results Through Bug Fixes or Justifications” (Polyspace Bug Finder).

Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add review information to your Polyspace results to fix the code later or to justify the result. You can use the information to keep track of your review progress.



If you add review information to your results file, they carry over to the results of the next analysis on the same project. If you add the same information as comments to your code (annotate), they carry over to any subsequent analysis of the code, whether in the same project or not.

Add Review Information in Result Details pane



Set the **Status** and **Severity**, and optionally enter notes with more explanations in the **Result Details** pane. The status indicates your response to the Polyspace result. To create a custom **Status**, see “Open Polyspace Access Results in a Desktop Interface”.

If you do not plan to fix your code in response to a result, assign one of the following statuses:

- Justified
- No Action Planned
- Not a Defect

Based on the status, Polyspace considers that you have given due consideration and justified that result (retained the code despite the result).

To facilitate your review workflow, Polyspace Access also classifies analysis findings as:

- **To Do**, with a status of Unreviewed.
- **In Progress**, with a status of To fix, To investigate, or Other.
- **Done**, with a status of Justified, No action planned, or Not a defect.

In the **DASHBOARD** perspective, findings that are **To Do** or **In Progress** are considered as **Open Issues**. If a Polyspace analysis of your code finds known or acceptable defects or coding rule violations, you can remove the defects or violations from this list of **Open Issues** in subsequent analyses by assigning one of the justified statuses outlined above.

Comment or Annotate in Code

If you enter code comments or annotations in a specific syntax, the software can read them and populate the **Severity**, **Status**, and comment fields in the next analysis of the code. Open your source code in an editor and enter the annotation on the same line as the result.

For the annotation syntax, see “Hide Known or Acceptable Polyspace Results” on page 2-5.

If you do not specify a status in your annotation, Polyspace assumes that you have set a status of **No Action Planned**.

See Also

More About

- “Hide Known or Acceptable Polyspace Results” on page 2-5

Hide Known or Acceptable Polyspace Results

If a Polyspace analysis of your code finds known or acceptable defects or coding rule violations, you can suppress them in subsequent analyses. Add information to your *results* or *code* indicating that you have reviewed the issues and that you do not intend to fix them.

Adding Polyspace-specific code annotations to a file ensures that the review information carries over to all subsequent analysis of the file using a Polyspace product. The annotated line no longer shows the known result even if the file is analyzed via another Polyspace project or using another Polyspace product.

This topic focuses primarily on hiding results using code annotations. If you want to keep Polyspace review information outside your code, see “Alternatives to Code Annotations” on page 2-10.

Note that you cannot hide the run-time errors detected with Code Prover from your source code even with code annotations. However, like all other results, the review information associated with a run-time error is extracted from the corresponding code annotation and shown with the result.

Review Workflow Using Code Annotations

Code annotations can facilitate your review by suppressing known results.

Polyspace Access Web Interface

If you assign a status of *Justified*, *No action planned*, or *Not a defect* to a result, the Polyspace Access interface classifies the result as **Done**. Instead of assigning one of these statuses to a result in the Polyspace Access interface, you can assign the status on the relevant line of code through code annotations.

- If you assign a status and other review information in the Polyspace Access interface, the information is associated with the Polyspace Access project and carries over to the next upload to the project.
- If you assign a status and other review information through code annotations, the information is associated with the file analyzed and carries over even when the file containing the result is part of another project in Polyspace Access.

Add annotations by typing them directly in your code, in an editor or IDE for instance. See the annotation syntax below. If you annotate a result in your code, you cannot edit the status, severity, or comment fields associated with the result in the Polyspace Access interface.

- For the general review workflow in the Polyspace Access web interface, see “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 2-2.
- For information on how to filter results marked as **Done**, see “Filter and Sort Results in Polyspace Access Web Interface” on page 3-2.

Polyspace as You Code

In Polyspace as You Code, you can enter code annotations to suppress a result from subsequent runs. Enter the annotations in one of these ways:

- If you code in an IDE with a Polyspace as You Code extension or plugin, use a menu option on the line with the result to enter annotations. See options in:

- Visual Studio on page 6-4
- Visual Studio Code on page 6-8
- Eclipse on page 6-12

Note that the annotation entered in this way uses a minimal syntax and implicitly indicates a status of `No action planned`. If you analyze the annotated file with another Polyspace product such as Polyspace Bug Finder Server, a result annotated in Polyspace as You Code is displayed with the status `No action planned`.

- If you code in an IDE that is not supported with a Polyspace as You Code plugin or extension, directly type the annotation in your code. See the annotation syntax below.

You cannot enter review information such as status directly in a Polyspace as You Code result because the results are overwritten in each run. You can either enter the information as code annotations or use a Polyspace Access project with the review information as baseline for Polyspace as You Code runs. The review information is picked up from the code annotations or baseline. For more information on using a baseline, see “Baselining in Polyspace as You Code”.

Code Annotation Syntax

To add comments directly to your source file, use the Polyspace annotation syntax. The syntax is not case sensitive, and has the following format. Both C style comments within `/* */` and C++ style comments starting with `//` are supported. The following syntax shows the minimal information required in a code annotation.

- Annotation for current line of code (including within macros):

```
line of code; /* polyspace Family:Result_name */
```

- Annotation for current line of code and n following lines:

```
code; /* polyspace +n Family:Result_name */
```

- Annotation for block of code:

```
/* polyspace-begin Family:Result_name */  
code;  
/* polyspace-end Family:Result_name */
```

Annotations begin with the keyword `polyspace` and must include `Family` and `Result_name` field values. You can optionally specify `Status`, `Severity`, and `Comment` field values.

```
polyspace Family:Result_name [Status:Severity] "Comment"
```

When you annotate a block of code, if subsequent annotations nested within that block of code apply to the same `Family` and `Result_name`, the nested annotation is applied.

For example, in this code, the annotation on line 9 is applied instead of the block annotation, but the block annotation is applied to the violation on line 7.

```

1 /*polyspace-begin MISRA-C:14.9 [To fix:High] "Block annotation"*/
2 int main(void) /*polyspace MISRA-C:14.7 "Nested annotation applied"*/
3 {
4     int x = 1;
5     int y = x / 2;
6
7     if (y < 0) /* Block annotation is applied to this violation of MISRA-C:14.9*/
8         y++;
9     if (x > y) /*polyspace MISRA-C:14.9 [Justified:Low] "Nested annotation applied"*/
10        return x;
11    return x;
12 }
13 /*polyspace-end MISRA-C:14.9 [To fix:High] "Block annotation"*/

```

If you do not specify a status, Polyspace Access considers the result **Done**, and assigns the status **No action planned** to the result.

To replace the different annotation fields with their allowed values, use the values in this table or see the examples on page 2-9.

Field	Allowed Value
<i>Family</i>	<p>Type of analysis result:</p> <ul style="list-style-type: none"> • DEFECT (Polyspace Bug Finder) • RTE, for run-time checks (Polyspace Code Prover) • CODE-METRICS, for function-level code complexity metrics • VARIABLE, for global variables (Polyspace Code Prover) • MISRA-C or MISRA2004 for MISRA C: 2004 rule violations • MISRA-AC-AGC for violations of MISRA C:2004 rules applicable to generated code • MISRA-C3 or MISRA2012 for MISRA C: 2012 rule violations. The annotation works even for the rules applicable to generated code. • CERT-C for CERT® C coding standard violations • CERT-CPP for CERT C++ coding standard violations • ISO-17961 for ISO/IEC TS 17961 coding standard violations • MISRA-CPP for MISRA C++ rule violations • AUTOSAR-CPP14 for AUTOSAR C++14 rule violations • JSF for JSF®++ rule violations • CUSTOM for violations of custom coding rules <p>To specify all analysis results, use the asterisk character *:*</p>

Field	Allowed Value
<i>Result_name</i>	<p>For DEFECT, use short names of checkers. See “Short Names of Bug Finder Defect Checkers” on page 2-12.</p> <p>For RTE, use short names of run-time checks. See “Short Names of Code Prover Run-Time Checks” (Polyspace Code Prover Access).</p> <p>For CODE-METRICS, use short names of code complexity metrics. See “Short Names of Code Complexity Metrics” on page 2-26.</p> <p>For VARIABLE, the only allowed value is the asterisk character " * ".</p> <p>For coding standard violations, specify the rule number or numbers.</p> <p>To specify all parts of a result name [MISRA2012:17.*] or all result names in a family [DEFECT:*], use the asterisk character.</p>
<i>Status</i>	<p>Text to indicate how you intend to address the error in your code. This value populates the Status column in the Results List pane as:</p> <ul style="list-style-type: none"> • Unreviewed • To investigate • To fix • Justified • No action planned • Not a defect • Other <p>Polyspace Access removes results annotated with status Justified, No action planned, or Not a defect from the list of Open Issues in subsequent analyses.</p>
<i>Severity</i>	<p>Text to indicate how critical you consider the error in your code. This value populates the Severity column in the Results List pane as:</p> <ul style="list-style-type: none"> • Unset • High • Medium • Low
<i>Comment</i>	<p>Additional text, such as a keyword or an explanation for the status and severity. This value populates the Comment column in the Results List pane.</p> <p>The additional text can span more than one line in the code. When showing this text in reports, leading and trailing spaces on a line are merged into one space so that the entire text can be read as a single paragraph.</p>

Code Annotation Syntax Examples

Annotate a Single Defect

Enter an annotation on the same line as the defect and specify the *Family* (DEFECT) and the *Result_name* (INT_OVFL). When you do not specify a status, Polyspace assigns the status No action planned and the result is considered **Done** in subsequent analyses.

```
int var = INT_MAX;
var++;/* polyspace DEFECT:INT_OVFL */
```

Annotate a Single Coding Standard Violation

Justify a coding standard violation, for instance, a CERT-C violation.

Enter an annotation on the same line as the violation and specify the *Family* (CERT-C) and the *Result_name* (the rule number, for instance, STR31-C). Assign the status Justified, severity Low and a comment.

```
code; /* polyspace CERT-C:STR31-C [Justified:Low] "Overflow cannot happen
because of external constraints." */
```

Annotate All MISRA C: 2012 Violations Over Multiple Lines

Enter an annotation with +n between polyspace and the *Family:Result_name* entries. The annotation applies to the same line and the n following lines.

This annotation applies to lines 4-7. The line count includes code, comments, and blank lines.

```
4. code ; // polyspace +3 MISRA2012:*
5. //comment
6.
7. code;
8. code;
```

Annotate All Code Metrics on Function

To annotate function-level code complexity metrics, in the function definition, enter an annotation on the same line as the function name.

This annotation suppresses all code complexity metrics for function func:

```
char func(char param) { //polyspace CODE-METRICS:*
    ...
}
```

Specify Multiple Families in the Same Annotation

Enter each family separated by a space. This annotation applies to all MISRA C:2012 rules 17 and to all run-time checks.

```
some code; /* polyspace MISRA2012:17.* RTE:* */
```

Specify Multiple Result Names in the Same Annotation

After you specify the *Family* (DEFECT), enter each *Result_name* separated by a comma.

```
system("rm ~/.config"); /* polyspace DEFECT:UNSAFE_SYSTEM_CALL,RETURN_NOT_CHECKED */
```

Add Explanatory Comments to Annotation

After you specify a *Family* and a *Result_name*, you can add a *Comment* with additional information for your justification. You can provide a comment for all families and result names, or a comment for each family or result name.

```
//Single comment
code; /* polyspace DEFECT:BAD_FREE MISRA2004:* "OK Defect and MISRA" */
//Multiple comments incorrect syntax:
code; /* polyspace DEFECT:* "OK defect" MISRA2004:5.2 "OK MISRA" */
//Multiple comments correct syntax:
code; /* polyspace DEFECT:* "OK defect" polyspace MISRA2004:5.2 "OK MISRA" */
```

In annotations, Polyspace ignores all text following double quotes. To specify additional *Family:Result_name*, [*Status:Severity*] or *Comment* entries, you must reenter the keyword polyspace after text in double quotes.

Set Status and Severity

You can specify allowed values on page 2-6 or enter custom values for status and severity.

```
//Set Status only
code; /* polyspace DEFECT:* [To fix] "some comment" */

//Set Status 'To fix' and Severity 'High'
code; /* polyspace VARIABLE:* [To fix: High] "some comment"*/

//Set custom status 'Assigned' and Severity 'Medium'
code; /* polyspace MISRA2012:12.* [Assigned: Medium] */
```

Alternatives to Code Annotations

If you want to keep Polyspace-specific information separate from your code but still hide known or acceptable results, you can add review information to the Polyspace results, and import them into later analysis results. There are several ways to import this information.

- In the most common workflow involving Polyspace Access, the review information is automatically imported. If you upload an analysis result to a project in the Polyspace Access web server, review information from the last uploaded run is imported to the current upload.
- You can explicitly force an import using:
 - The option `-import-comments` with commands such as `polyspace-bug-finder-server` or `polyspace-code-prover-server`. See `-import-comments`.
 - The `polyspace-comments-import` command. See `polyspace-comments-import`.

Using the `polyspace-comments-import` command allows you to import from more than one set of results.

- You can use a Polyspace Access project as baseline for Polyspace as You Code runs. See “Baselining in Polyspace as You Code”.

See Also

More About

- “Define Custom Annotation Format” on page 2-28
- “Short Names of Bug Finder Defect Checkers” on page 2-12
- “Short Names of Code Complexity Metrics” on page 2-26

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 analysis option `Find defects (-checkers)` in the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server.

Defect	Command-line Name
*this not returned in copy assignment operator	RETURN_NOT_REF_TO_THIS
A move operation may throw	MOVE_OPERATION_MAY_THROW
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
Asynchronously cancellable thread	ASYNCHRONOUSLY_CANCELLABLE_THREAD
Atomic load and store sequence not atomic	ATOMIC_VAR_SEQUENCE_NOT_ATOMIC
Atomic variable accessed twice in an expression	ATOMIC_VAR_ACCESS_TWICE
Automatic or thread local variable escaping from a thread	LOCAL_ADDR_ESCAPE_THREAD
Bad file access mode or status	BAD_FILE_ACCESS_MODE_STATUS
Bad order of dropping privileges	BAD_PRIVILEGE_DROP_ORDER

Defect	Command-line Name
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
Bytewise operations on nontrivial class object	MEMOP_ON_NONTRIVIAL_OBJ
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
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
Const parameter values may cause unnecessary data copies	CONST_PARAMETER_VALUE
Const return values may cause unnecessary data copies	CONST_RETURN_VALUE
Const rvalue reference parameter may cause unnecessary data copies	CONST_RVALUE_REFERENCE_PARAMETER

Defect	Command-line Name
Const std::move input may cause a more expensive object copy	EXPENSIVE_STD_MOVE_CONST_OBJECT
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
Data race on adjacent bit fields	DATA_RACE_BIT_FIELDS
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

Defect	Command-line Name
Empty destructors may cause unnecessary data copies	EMPTY_DESTRUCTOR_DEFINED
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
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
Expensive <code>c_str()</code> to <code>std::string</code> construction	EXPENSIVE_C_STR_STD_STRING_CONSTRUCTION
Expensive constant <code>std::string</code> construction	EXPENSIVE_CONSTANT_STD_STRING
Expensive copy in a range-based for loop iteration	EXPENSIVE_RANGE_BASED_FOR_LOOP_ITERATION
Expensive local variable copy	EXPENSIVE_LOCAL_VARIABLE
Expensive logical operation	EXPENSIVE_LOGICAL_OPERATION
Expensive pass by value	EXPENSIVE_PASS_BY_VALUE
Expensive return by value	EXPENSIVE_RETURN_BY_VALUE
Expensive use of non-member <code>std::string</code> operator+() instead of a simple append	EXPENSIVE_STD_STRING_APPEND
Expensive use of <code>std::string</code> methods instead of more efficient overload	EXPENSIVE_USE_OF_STD_STRING_METHODS
Expensive use of <code>std::string</code> with empty string literal	UNNECESSARY_EMPTY_STRING_LITERAL
File access between time of check and use (TOCTOU)	TOCTOU
File descriptor exposure to child process	FILE_EXPOSURE_TO_CHILD
File does not compile	file_does_not_compile
File manipulation after <code>chroot</code> without <code>chdir</code>	CHROOT_MISUSE

Defect	Command-line Name
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
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
Hard-coded sensitive data	HARD_CODED_SENSITIVE_DATA
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
Incorrect value forwarding	INCORRECT_VALUE_FORWARDING
Incorrectly indented statement	INCORRECT_INDENTATION
Inefficient string length computation	INEFFICIENT_BASIC_STRING_LENGTH
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
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

Defect	Command-line Name
Invalid use of standard library string routine	STR_STD_LIB
Invalid va_list argument	INVALID_VA_LIST_ARG
Join or detach of a joined or detached thread	DOUBLE_JOIN_OR_DETACH
Lambda used as decltype or typeid operand	LAMBDA_TYPE_MISUSE
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
Macro terminated with a semicolon	SEMICOLON_TERMINATED_MACRO
Macro with multiple statements	MULTI_STMT_MACRO
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
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

Defect	Command-line Name
Missing case for switch condition	MISSING_SWITCH_CASE
Missing certification authority list	CRYPTO_SSL_NO_CA
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 constexpr specifier	MISSING_CONSTEXPR
Missing data for encryption, decryption or signing operation	CRYPTO_PKEY_NO_DATA
Missing explicit keyword	MISSING_EXPLICIT_KEYWORD
Missing final step after hashing update operation	CRYPTO_MD_NO_FINAL
Missing hash algorithm	CRYPTO_MD_NO_ALGORITHM
Missing lock	BAD_UNLOCK
Missing null in string array	MISSING_NULL_CHAR
Missing or double initialization of thread attribute	BAD_THREAD_ATTRIBUTE
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 private key for X.509 certificate	CRYPTO_SSL_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 salt for hashing operation	CRYPTO_MD_NO_SALT
Missing unlock	BAD_LOCK
Missing virtual inheritance	MISSING_VIRTUAL_INHERITANCE

Defect	Command-line Name
Missing X.509 certificate	CRYPTO_SSL_NO_CERTIFICATE
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
Modification of internal buffer returned from nonreentrant standard function	WRITE_INTERNAL_BUFFER_RETURNED_FROM_STD_FUNC
Move operation on const object	MOVE_CONST_OBJECT
Multiple mutexes used with same conditional variable	MULTI_MUTEX_WITH_ONE_COND_VAR
Multiple threads waiting on same condition variable	SIGNALED_COND_VAR_NOT_UNIQUE
No data added into context	CRYPTO_MD_NO_DATA
Noexcept function exits with exception	NOEXCEPT_FUNCTION_THROWS
Non-compliance with AUTOSAR specification	autosar_lib_non_compliance
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

Defect	Command-line Name
Opening previously opened resource	DOUBLE_RESOURCE_OPEN
Operator new not overloaded for possibly overaligned class	MISSING_OVERLOAD_NEW_FOR_ALIGNED_OBJ
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	TAINTED_PTR_OFFSET
Pointer or reference to stack variable leaving scope	LOCAL_ADDR_ESCAPE
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 inappropriate data type for switch expression	INAPPROPRIATE_TYPE_IN_SWITCH
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
Redundant expression in sizeof operand	sizeof_USELESS_OP

Defect	Command-line Name
Resource leak	RESOURCE_LEAK
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
Semicolon on same line as if, for or while statement	SEMICOLON_CTRL_STMT_SAME_LINE
Sensitive data printed out	SENSITIVE_DATA_PRINT
Sensitive heap memory not cleared before release	SENSITIVE_HEAP_NOT_CLEARED
Server certificate common name not checked	CRYPTO_SSL_HOSTNAME_NOT_CHECKED
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
Static uncalled function	UNCALLED_FUNC
std::endl may cause an unnecessary flush	STD_ENDL_USE
std::move called on an unmovable type	STD_MOVE_UNMOVABLE_TYPE
Stream argument with possibly unintended side effects	STREAM_WITH_SIDE_EFFECT

Defect	Command-line Name
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
Throw argument raises unexpected exception	THROW_ARGUMENT_EXPRESSION_THROWS
TLS/SSL connection method not set	CRYPTO_SSL_NO_ROLE
TLS/SSL connection method set incorrectly	CRYPTO_SSL_BAD_ROLE
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
Unmodified variable not const-qualified	UNMODIFIED_VAR_NOT_CONST
Unnamed namespace in header file	UNNAMED_NAMESPACE_IN_HEADER
Unprotected dynamic memory allocation	UNPROTECTED_MEMORY_ALLOCATION
Unreachable code	UNREACHABLE
Unreliable cast of function pointer	FUNC_CAST
Unreliable cast of pointer	PTR_CAST
Unsafe call to a system function	UNSAFE_SYSTEM_CALL

Defect	Command-line Name
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
Use of a forbidden function	FORBIDDEN_FUNC
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
Use of indeterminate string	INDETERMINATE_STRING
Use of new or make_unique instead of more efficient make_shared	MISSING_MAKE_SHARED
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 tainted pointer	TAINTED_PTR

Defect	Command-line Name
Use of setjmp/longjmp	SETJMP_LONGJMP_USE
Use of undefined thread ID	UNDEFINED_THREAD_ID
Use of signal to kill thread	THREAD_KILLED_WITH_SIGNAL
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
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
X.509 peer certificate not checked	CRYPTO_SSL_CERT_NOT_CHECKED

See Also

More About

- “Hide Known or Acceptable Polyspace Results” on page 2-5

Short Names of Code Complexity Metrics

When annotating your code to justify metrics or creating custom software quality objectives, you use short names of code complexity metrics instead of the full names. The following table lists the short names for code complexity metrics.

Note that you can only annotate your code for function level code complexity metrics only.

Project Metrics

Code Metric	Acronym
Number of Direct Recursions	AP_CG_DIRECT_CYCLE
Number of Header Files	INCLUDES
Number of Files	FILES
Number of Protected Shared Variables (Code Prover only)	PSHV
Number of Recursions	AP_CG_CYCLE
Number of Potentially Unprotected Shared Variables (Code Prover only)	UNPSHV
Program Maximum Stack Usage (Code Prover only)	PROG_MAX_STACK
Program Minimum Stack Usage (Code Prover only)	PROG_MIN_STACK

File Metrics

Code Metric	Acronym
Comment Density	COMF
Estimated Function Coupling	FCO
Number of Lines	TOTAL_LINES
Number of Lines Without Comment	LINES_WITHOUT_CMT

Function Metrics

Code Metric	Acronym
Cyclomatic Complexity	VG
Higher Estimate of Local Variable Size	LOCAL_VARS_MAX
Language Scope	VOCF
Language Scope	LOCAL_VARS_MIN
Minimum Stack Usage (Code Prover only)	MIN_STACK
Maximum Stack Usage (Code Prover only)	MAX_STACK
Number of Call Levels	LEVEL

Code Metric	Acronym
Number of Call Occurrences	NCALLS
Number of Called Functions	CALLS
Number of Calling Functions	CALLING
Number of Executable Lines	FXLN
Number of Function Parameters	PARAM
Number of Goto Statements	GOTO
Number of Instructions	STMT
Number of Lines Within Body	FLIN
Number of Local Non-Static Variables	LOCAL_VARS
Number of Local Static Variables	LOCAL_STATIC_VARS
Number of Paths	PATH
Number of Return Statements	RETURN

See Also

More About

- “Hide Known or Acceptable Polyspace Results” on page 2-5

Define Custom Annotation Format

This example shows how to create and edit an XML file to define an annotation format and map it to the Polyspace annotation syntax. Once you create and edit the XML file, pass the file to Polyspace by using option `-xml-annotations-description`.

To define multiple custom annotation formats, see “Define Multiple Custom Annotation Syntaxes” on page 2-34.

To get started, copy the following code to a text editor and save it on your machine as `annotations_description.xml`.

```

<?xml version="1.0" encoding="UTF-8"?>
<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
  Group="example XML">

  <Expressions Search_For_Keywords="myKeyword"
    Separator_Result_Name="," >
    <!-- Define annotation format in this
      section by adding <Expression/> elements -->

    <Expression Mode="SAME_LINE"
      Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"
      Rule_Identifier_Position="1"
    />

    <Expression Mode="GOTO_INCREMENT"
      Regex="myKeyword\s+(\d+\s)(\w+(\s*,\s*\w+)*)"
      Increment_Position="1"
      Rule_Identifier_Position="2"
    />

    <Expression Mode="BEGIN"
      Regex="myKeyword\s*(\w+(\s*,\s*\w+)*)\s*Block_on"
      Rule_Identifier_Position="1"
      Case_Insensitive="true"
    />

    <Expression Mode="END"
      Regex="myKeyword\s*(\w+(\s*,\s*\w+)*)\s*Block_off"
      Rule_Identifier_Position="1"
    />

    <Expression Mode="END_ALL"
      Regex="myKeyword\sBlock_off_all"
    />

    <Expression Mode="SAME_LINE"

    Regex="myKeywords\s+(\w+(\s*,\s*\w+)*)
    (\s*\[(\w+\s*)*([:\s*(\w+\s*)+)*\])*(\s*-\s*)*([^\s]*)\s*-.*"
    Rule_Identifier_Position="1"
    Status_Position="4"
    Severity_Position="6"
    Comment_Position="8"
    />
    <!-- Put the regular expression on a single line instead of two line
    when you copy it to a text editor -->

    <!-- SAME_LINE example with more complex regular expression.
      Matches the following annotations:
      //myKeywords 50 [my_status:my_severity] -Additional comment-
      //myKeywords 50 [my_status]
      //myKeywords 50 [:my_severity]
      //myKeywords 50 -Additional comment-
    -->

  </Expressions>

  <Mapping>
    <!-- Map your annotation syntax to the Polyspace annotation
      syntax by adding <Result_Name_Mapping /> elements in this section -->

    <Result_Name_Mapping Rule_Identifier="100" Family="DEFECT" Result_Name="INT_ZERO_DIV"/>

    <Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
    <Result_Name_Mapping Rule_Identifier="51" Family="MISRA-C3" Result_Name="8.7"/>
    <Result_Name_Mapping Rule_Identifier="ALL_MISRA" Family="MISRA-C3" Result_Name="*" />
  </Mapping>
</Annotations>

```

The XML file consists of two parts:

- <Expressions> . . . </Expressions> where you define the format of your annotation syntax.

- `<Mapping>...</Mapping>` where you map your syntax to the Polyspace annotation syntax.

After you edit this file, Polyspace can interpret your custom code annotation when you invoke the option `-xml-annotations-description`.

Define Annotation Syntax Format

To define an annotation syntax in Polyspace, your syntax must follow a pattern that you can represent with a regular expression. See “Regular Expressions” (MATLAB). It is recommended that you include a keyword in the pattern of your annotation syntax to help identify it. In this example, the keyword is `myKeyword`. Set the attribute `Search_For_Keywords` equal to this keyword.

Once you know the pattern of your annotation, you can define it in the XML by adding an `<Expression/>` element and specifying at least the attributes `Mode`, `Regex`, and `Rule_Identifier_Position`. For instance, the first `<Expression/>` element in `annotations_description.xml` defines an annotation with these attributes:

- `Mode="SAME_LINE"`. The annotation applies to code on the same line.
- `Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"`. Polyspace uses the regular expression to search for a string that begins with `myKeyword`, followed by a space `\s+`. Polyspace then searches for a capturing group `(\w+(\s*,\s*\w+)*)` that includes an alphanumeric rule identifier `\w+` and, optionally, additional comma-separated rule identifiers `(\s*,\s*\w+)*`.
- `Rule_Identifier_Position="1"`. The integer value of this attribute corresponds to the number of opening parentheses preceding the relevant capturing group in the regular expression. In `myKeyword\s+(\w+(\s*,\s*\w+)*)`, one opening parenthesis precedes the capturing group of the rule identifier `(\w+(\s*,\s*\w+)*)`. If you want to match rule identifiers captured by `(\s*,\s*\w+)`, then you set `Rule_Identifier_Position="2"` because two opening parentheses precede this capturing group.

The list of attributes and their values are listed in this table. The example column refers to the format defined in `annotations_description.xml`.

Attribute	Use	Value	Example
Mode	Required	SAME_LINE	Applies only on the same line as the annotation. code; //myKeyword 100
		GOTO_INCREMENT	Applies on the same line as the annotation and the following n lines: 3. code; // myKeyword +3 ALL_MISRA 4. /*comments */ 5. 6. code; 7. code; The preceding annotation applies to lines 3-6 only.

Attribute	Use	Value	Example
		BEGIN	<p>Applies to the same line and all following lines until a corresponding expression with attribute Mode="END" or "END_ALL", or until the end of the file.</p> <pre>//myKeyword 50, 51 Block_on Code block 1; ...</pre>
		END	<p>Stops the application of a rule identifier declared by a corresponding expression with attribute Mode="BEGIN".</p> <pre>//myKeyword 50, 51 Block_on Code block 1; ... More code; //myKeyword 50 Block_off</pre> <p>Only rule identifier 50 is turned off. Rule identifier 51 still applies.</p>
		END_ALL	<p>Stops all rule identifiers declared by an expression with attribute Mode="BEGIN".</p> <pre>//myKeyword 50, 51 Block_on Code block 1; ... More code; //myKeyword Block_off_all</pre> <p>Rule identifiers 50 and 51 are turned off.</p>
Regex	Required	Regular expression search string	<p>See "Regular Expressions" (MATLAB). Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)" matches these expressions:</p> <pre>// myKeyword 50, 51 /* myKeyword ALL_MISRA, 100 */</pre>

Attribute	Use	Value	Example
Rule_Identifier_Position	Required, except when you set Mode="END_ALL"	Integer	<p>The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.</p> <pre><Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s) (\w+(\s*,\s*\w+)*)" Increment_Position="1" Rule_Identifier_Position="2"/></pre> <p>Note Enter the regex expression on a single line when you edit your XML file.</p> <p>The search expression for the rule identifier <code>\w+(\s*,\s*\w+)*</code> is after the second opening parenthesis of the regular expression.</p>
Increment_Position	Required only when you set Mode="GOTO_INCREMENT"	Integer	<p>The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.</p> <pre><Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s) (\w+(\s*,\s*\w+)*)" Increment_Position="1" Rule_Identifier_Position="2"/></pre> <p>Note Enter the regex expression on a single line when you edit your XML file.</p> <p>The search expression for the increment <code>\+\d+\s</code> is after the first opening parenthesis of the regular expression.</p>
Status_Position	Optional	Integer	<p>See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the Status column on the Results List pane of the user interface.</p>
Severity_Position	Optional	Integer	<p>See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the Severity column on the Results List pane of the user interface.</p>

Attribute	Use	Value	Example
Comment_Position	Optional	Integer	See Increment_Position example. When you use this attribute, the entry in your annotation is displayed in the Comment column on the Results List pane of the user interface. Your comment is appended to the string Justified by annotation in source :
Case_Insensitive	Optional	True or false	When you set this attribute to "true", the regular expression is case insensitive, otherwise it is case sensitive. If you do not declare this attribute in your expression, the regular expression is case sensitive. For Case_Insensitive="true", these annotations are equivalent: //MYKEYWORD ALL_MISRA BLOCK_ON //mykeyword all_misra block_on

Map Your Annotation to the Polyspace Annotation Syntax

After you define your annotation format, you can map the rule identifiers you are using to their corresponding Polyspace annotation syntax. You can do this mapping by adding an `<Result_Name_Mapping/>` element and specifying attributes `Rule_Identifier`, `Family`, and `Result_Name`. For instance, if rule identifier 50 corresponds to MISRA C: 2012 rule 8.4, map it to the Polyspace syntax MISRA-C3:8.4 by using this element:

```
<Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
```

The list of attributes and their values are listed in this table. The example column refers to the format defined in `annotations_description.xml`.

Attribute	Use	Value	Example
Rule_Identifier	Required	User defined. Each value must be unique.	See the mapping section of <code>annotations_description.xml</code>
Family	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 2-6.	See the mapping section of <code>annotations_description.xml</code>
Result_Name	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 2-6.	See the mapping section of <code>annotations_description.xml</code>

Define Multiple Custom Annotation Syntaxes

To define more than one annotation syntax, in your XML file, specify a comma separated list of keywords associated with each syntax for the `Search_For_Keywords` attribute.

For example, if you use custom annotations that follow these patterns to annotate violations of MISRA C: 2012 rules:

```
int func(int p) //customSyntax M123 $ customSyntax M124
{
    int i;
    int j = 1;

    i = 1024 / (j - p);
    return i;
}

int func2(void){ //otherCustomSyntax 50
    int x=func(2);
    return x;
}
```

Enter the following in the XML file where you define the custom annotation syntax.

```
<?xml version="1.0" encoding="UTF-8"?>
<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
  Group="multipleCustomSyntax">
  <!-- Enter comma separated list of keywords -->
  <Expressions Search_For_Keywords="customSyntax,otherCustomSyntax"
    Separator_Result_Name="$" >

    <!-- This section defines the annotation syntax format -->
    <Expression Mode="SAME_LINE"
      Regex="customSyntax\s(\w+(\s*,\s*\w+)*)"
      Rule_Identifier_Position="1"
    />
    <Expression Mode="SAME_LINE"
      Regex="otherCustomSyntax\s(\w+(\s*,\s*\w+)*)"
      Rule_Identifier_Position="1"
    />
  </Expressions>
  <!-- This section maps the user annotation to the Polyspace
  annotation syntax -->
  <Mapping>
    <!-- Mapping for customSyntax rules -->
    <Result_Name_Mapping Rule_Identifier="M123" Family="MISRA-C3" Result_Name="8.7"/>
    <Result_Name_Mapping Rule_Identifier="M124" Family="MISRA-C3" Result_Name="D4.6"/>
    <!-- Mapping for otherCustomSyntax rules -->
    <Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
  </Mapping>
</Annotations>
```

When you use multiple custom annotations, each rule identifier must be unique. For instance, in the preceding example, you cannot reuse rule identifier M123 with `otherCustomSyntax`.

See Also

-xml-annotations-description

More About

- “Annotation Description Full XML Template” on page 2-36
- “Hide Known or Acceptable Polyspace Results” on page 2-5

- “Resolve -xml-annotations-description Errors” on page 4-7

Annotation Description Full XML Template

This table lists all the elements, attributes, and values of the XML that you can use to define an annotation format and map it to the Polyspace annotation syntax. For an example of how to edit an XML to define annotation syntax, see “Define Custom Annotation Format” on page 2-28.

Element	Attribute	Use	Value
Annotations	Group	Required	User defined string. For example, "Custom Annotations"
Expressions	Search_For_Keyword s	Required	User defined string. This string is a keyword you include in the pattern of your annotation syntax to help identify it. For example, "myKeyword". To use multiple custom annotations, enter a comma separated list of keyword. See “Define Multiple Custom Annotation Syntaxes” on page 2-34.
	Separator_Result_Name	Required	User defined string. This string is a separator when you list multiple Polyspace result names in the same annotation. For example ","
	Separator_Family_And_Result_Name	Optional	User defined string. This string is a separator when you list multiple Polyspace results families in the same annotation. For example, " "
	Separator_Family	Optional	User defined string. This string is a separator when you list a Polyspace results family and results name in the same annotation. For example, ":"
Expression	Mode	Required	SAME_LINE
			GOTO_INCREMENT
			BEGIN

Element	Attribute	Use	Value
			END
			END_ALL
			NEXT_CODE_LINE
			The annotation applies to the next line of code. Comments and blank lines are ignored.
			GOTO_LABEL
			LABEL
			XML_START
			XML_CONTENT
			The annotation for this expression must be on a single line.
	XML_END		
	Regex	Required	Regular expression search string that matches the pattern of your annotation.
	Rule_Identifier_Position	Required, except when you set Mode="END_ALL" or "LABEL"	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Increment_Position	Required only when you set Mode="GOTO_INCREMENT"	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Status_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.

Element	Attribute	Use	Value
	Severity_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Comment_Position	Optional	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Label_Position	Required only when you set Mode="GOTO_LABEL" or "LABEL"	Integer. The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression.
	Case_Insensitive	Optional	True or false. When you do not declare this attribute, the default value is false.
	Is_Pragma	Optional	True or false. When you do not declare this attribute, the default value is false. Set this attribute to true if you want to declare your annotation using a pragma instead of a comment.
	Applies_Also_On_Same_Line	Optional	True or false. When you do not declare this attribute, the default value is true. Use this attribute to enable annotations with the old Polyspace syntax to apply on the same line of code.

Element	Attribute	Use	Value
Mapping	None	None	None
Result_Name_Mapping	Rule_Identifier	Required	User defined
	Family	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 2-6.
	Result_Name	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 2-6.

Example

This example code covers some of the less commonly used attributes for defining annotations in XML.

```

<?xml version="1.0" encoding="UTF-8"?>

<Annotations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="annotations_xml_schema.xsd"
  Group="XML Template">

  <Expressions Separator_Result_Name="."
    Search_For_Keywords="myKeyword">

    <Expression Mode="GOTO_LABEL"
      Regex="(\\A|\\W)myKeyword\\s+S\\s+(\\d+(\\s*,\\s*\\d+)*\\s+([a-zA-Z_-]\\w+)"
      Rule_Identifier_Position="2"
      Label_Position="4"

      />

    <Expression Mode="LABEL"
      Regex="(\\A|\\W)myKeyword\\s+L:(\\w+)"
      Label_Position="2"

      />

    <!-- Annotation applies starting current line until
      next declaration of label word "myLabel"
      Example:

      code; // myKeyword S 100 myLabel
      ...
      more code;
      // myKeyword L myLabel
    -->

    <Expression Mode="BEGIN"
      Regex="#\\s*pragma\\s+myKeyword_MESSAGES_ON\\s+(\\w+)"
      Rule_Identifier_Position="1"
      Is_Pragma="true"
      />

    <!-- Annotation declared with pragma instead of comment
      Example:#pragma myKeyword_MESSAGES_ON 100 -->

    <!-- Comment declaration with XML format-->

    <!-- XML_START must be declared before XML_CONTENT -->
    <Expression Mode="XML_START"
      Regex="<\\s*myKeyword_COMMENT\\s*>"

      />

    <!-- Example: <myKeyword_COMMENT> -->

    <Expression Mode="XML_CONTENT"
      Regex="<\\s*(\\d*)\\s*>(((?![*]/)(?!<).)*)</\\s*(\\d*)\\s*>"
      Rule_Identifier_Position="1"
      Comment_Position="2"

      />

    <!-- Example: <100>This is my comment</100>
      XML_CONTENT must be declare on a single line.

      <100>This is my comment
      </100>
      is incorrect.
    -->

    <Expression Mode="XML_END"
      Regex="</\\s*myKeyword_COMMENT\\s*>"

      />

    <!-- Example: </myKeyword_COMMENT> -->
  </Expressions>

  <Mapping>

  <Result_Name_Mapping Rule_Identifier="100" Family="MISRA-C" Result_Name="4.1"/>
  </Mapping>
</Annotations>

```


See Also

More About

- “Hide Known or Acceptable Polyspace Results” on page 2-5

Manage Results

- “Filter and Sort Results in Polyspace Access Web Interface” on page 3-2
- “Create Custom Filter Groups in Polyspace Access Web Interface” on page 3-6
- “Compare Analysis Results to Previous Runs” on page 3-8
- “Classification of Defects by Impact” on page 3-11
- “Bug Finder Defect Groups” on page 3-22

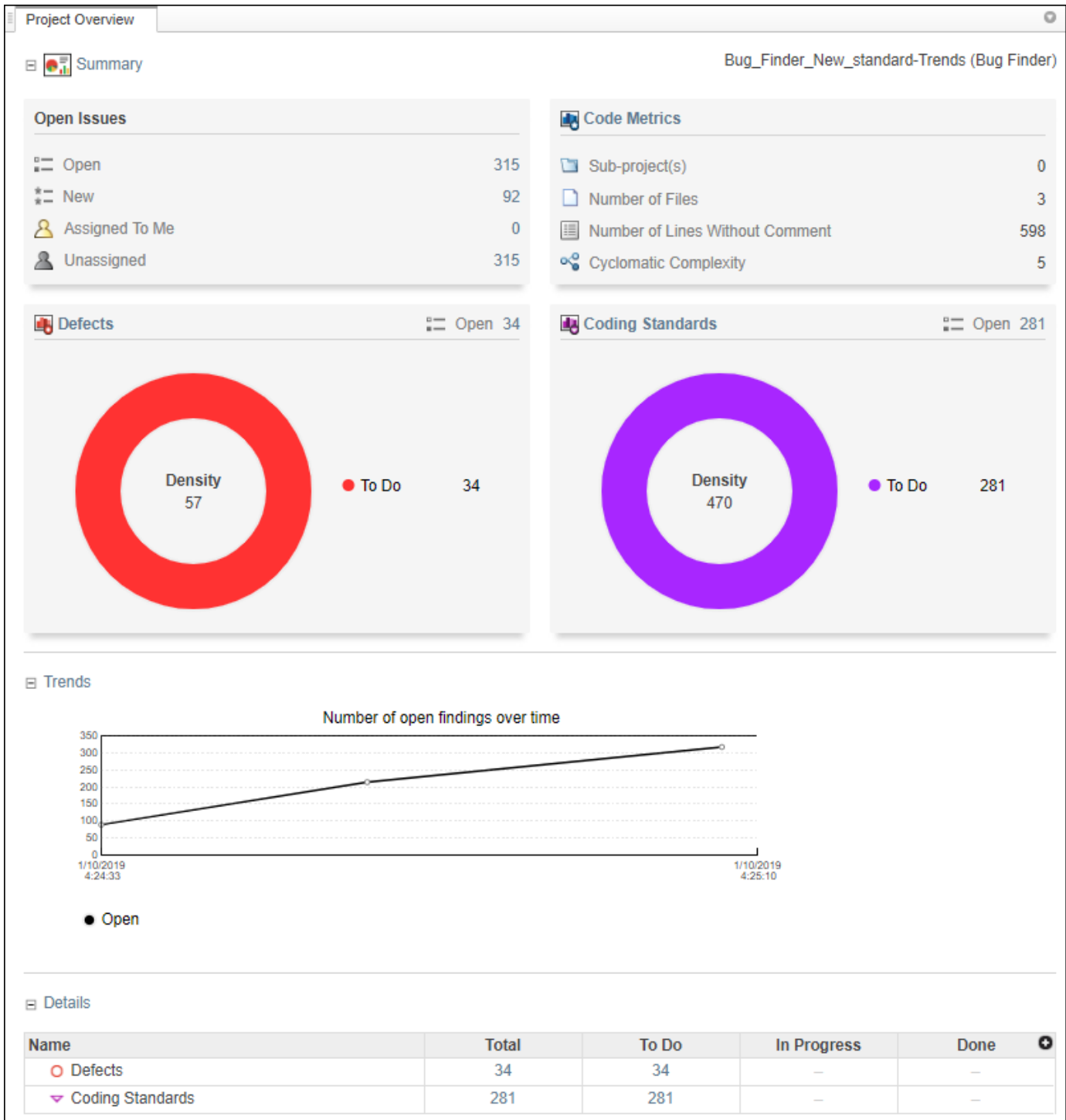
Filter and Sort Results in Polyspace Access Web Interface

This topic describes how to filter, sort, and otherwise manage results in the Polyspace Access web interface. For a similar workflow in the user interface of the Polyspace desktop products, see “Filter and Group Results in Polyspace Desktop User Interface” (Polyspace Bug Finder).

When you open the results of a Polyspace analysis in the **DASHBOARD** view of Polyspace Access, you see statistics about your project in the **Project Overview** dashboard. The statistics cover findings for:

- Bug Finder “Defects”.
- Code Prover “Run-Time Checks” (Polyspace Code Prover Access).
- “Coding Standards” violations.
- “Code Metrics” and “Bug Finder Quality Objectives” on page 1-38 compliance.

To organize your review, you can narrow down the list or group results by file or result type.



Some of the ways you can use filtering are:

- You can display only certain types of defects or run-time checks.

For instance, for a Bug Finder analysis, you can display only high-impact defects. See “Classification of Defects by Impact” on page 3-11.

- You can display only new results found since the last analysis or since a previous analysis. See “Compare Analysis Results to Previous Runs” on page 3-8.
- You can display only the results that you have not justified. Results that are not justified are considered **Open**. They are results with status Unreviewed, To Investigate, To Fix, or Other.

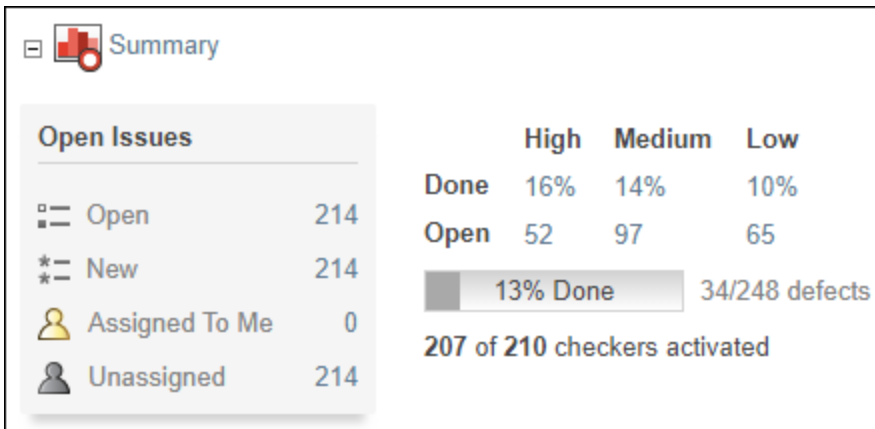
For information on justification, see “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 2-2.

- You can display only results that you still need to address to reach a **Quality Objectives** threshold.

Filter Results

You can filter results by drilling down on a set of results in a dashboard, or directly in the **Results List** pane by using the **REVIEW** toolstrip filters.

Filter Using Dashboards



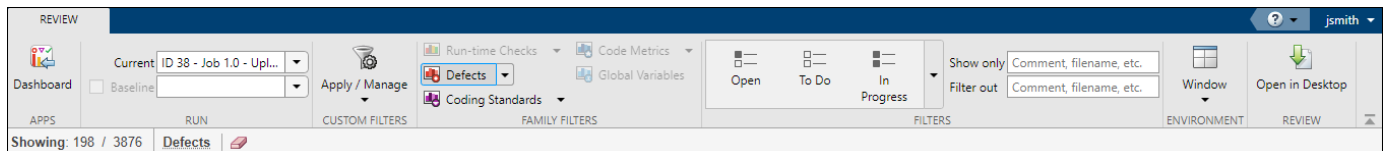
In the **DASHBOARD** view, you can:

- Click a section of a pie chart or a pie chart legend on the **Project Overview** dashboard to see the corresponding set of results.
- Open dashboards for different families of results, then click a number to open a list filtered to the corresponding set of results. For instance:
 - To see only high-impact defects that are still **Open** in a Bug Finder analysis, click the corresponding number in the **Summary** section of the **Defects** dashboard. **Open** results have status Unreviewed, To Investigate, To Fix, or Other.
 - To see only red checks that are **Done** in a Code Prover analysis, click the corresponding number in the **Summary** section of the **Run-time Checks** dashboard. **Done** results have status Justified, No Action Planned, or Not A Defect.
 - To see violations of the MISRAC C:2012 coding standards in a particular file, use the table in the **Details** section of the **MISRA C:2012** dashboard.

- Compare the **Current** run to an earlier **Baseline** run and review **New** or **Unresolved** findings. See “Compare Analysis Results to Previous Runs” on page 3-8.

If you select a folder that contains multiple projects in the **Project Explorer**, the dashboards display an aggregate of results for all the projects. Most of the fields in the dashboard are not clickable when you look at the statistics for multiple projects.

Filter Using REVIEW Toolstrip



In the **REVIEW** view, you can filter results by families of Polyspace results (**FAMILY FILTERS**), or by result review progress (**FILTERS**).

The filter bar underneath the toolstrip shows how many findings are displayed out of the total findings, along with which filters are currently applied.

The buttons in the **FILTERS** section of the toolstrip are global. They apply to all families of findings.

To filter results by specific content, such as a function name, use the **Show only** or **Filter out** text filters. These filters match the text you enter against the content of all the columns in the “Results List” on page 1-26. For instance, if you enter `foo` in the **Filter out** filter, the **Results List** hides all the results that contain `foo` in any of the **Results List** columns.

You can also filter results by right-clicking the content of a column in the **Results List**. This action is equivalent to entering the content directly in the **Show only** or **Filter out** text filters. For instance, if you right-click `foo` in the **Function** column, the filter applies to all results that contain `foo` in any of the **Results List** columns.

Filters you apply do not carry over to the next analysis.

See Also

More About

- “Classification of Defects by Impact” on page 3-11

Create Custom Filter Groups in Polyspace Access Web Interface

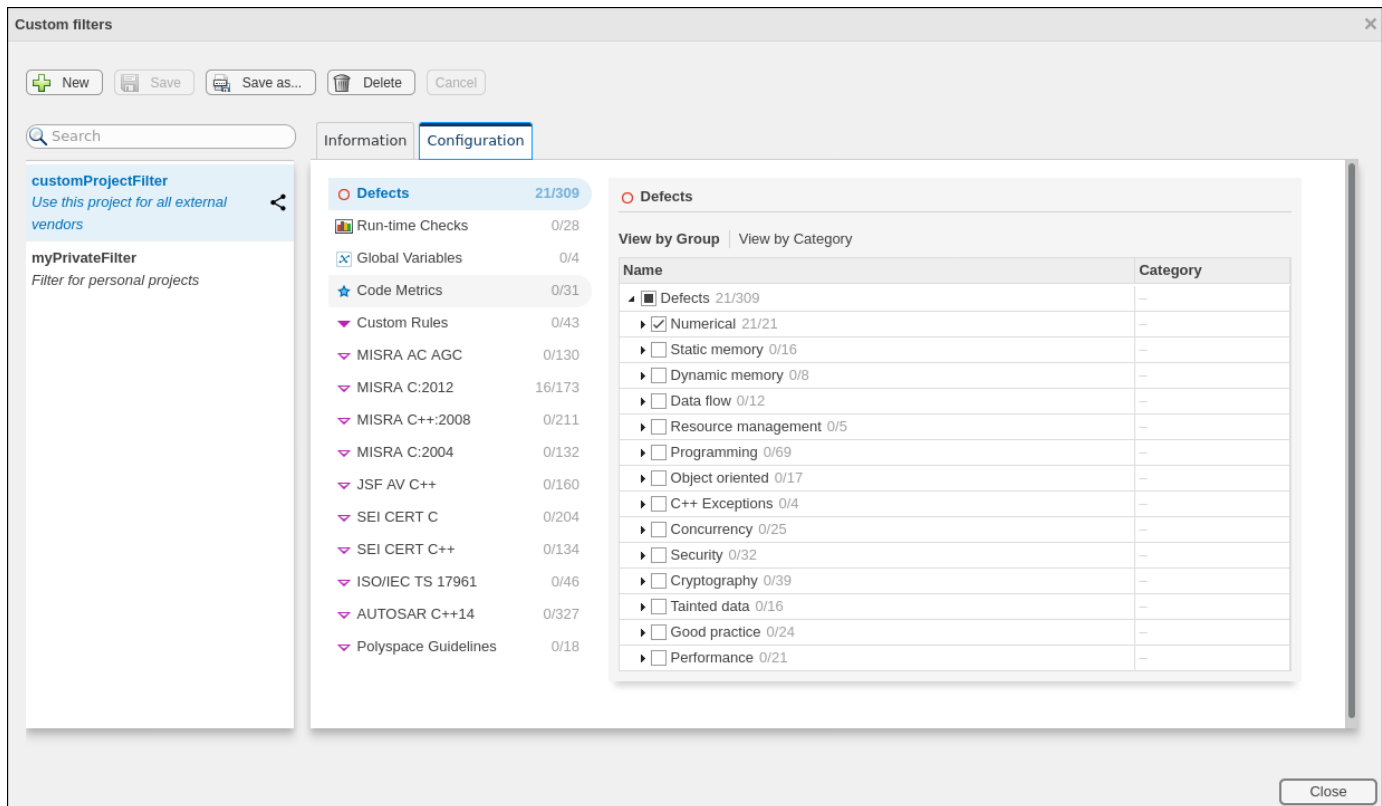
When you review results in the **Results List**, you can apply filters from the **FAMILY FILTERS** section of the toolstrip to focus your review on specific Polyspace families of results, such as:

- Bug Finder “Defects”.
- Code Prover “Run-Time Checks” (Polyspace Code Prover Access) and “Global Variables” (Polyspace Code Prover Access).
- “Coding Standards”.
- “Code Metrics”.

Define custom filters to narrow the scope of your review to only findings that are relevant to your project or organization. For instance, you might be interested in reviewing only **Numerical** Bug Finder defects and violations of **Mandatory MISRA C:2012** rules.

Once you define custom filters, you can share those filters with other Polyspace Access users to ensure consistent review scopes across your projects or organization.

To create or edit a custom filter, click **Apply/Manage > Manage filters**.



To create a new filter, in the **Custom filters** window, click **New** and then enter the filter name in the **New Custom Filter** pop-up window. You can optionally provide a description and enable the **Shared filter** checkbox to share the filter with other Polyspace Access users.

By default, custom filters are private and can be viewed only by the user who creates the filter. A private filter can be edited only by the user who creates that filter. A shared filter can be edited by the user who creates the filter or by a user with the role of **Administrator**.

To make changes to a filter name, description, or to enable or disable filter sharing, go to the **Information** tab.

To edit the filter selection, on the **Configuration** tab, click a Polyspace results family, for instance MISRA C:2012, and then select a node or expand the node to select individual results. For each family of results, you can view the nodes by group or by category when available.

To save your changes, click **Save** or **Save as** to save your edits in as new custom filter.

Apply custom filters by selecting the appropriate filter from **Apply/Manage > Private filters** or **Apply/Manage > Shared filters**. You can apply more than one custom filter, including combinations of private and shared filters.

Custom filters do not apply to the **DASHBOARD** view.

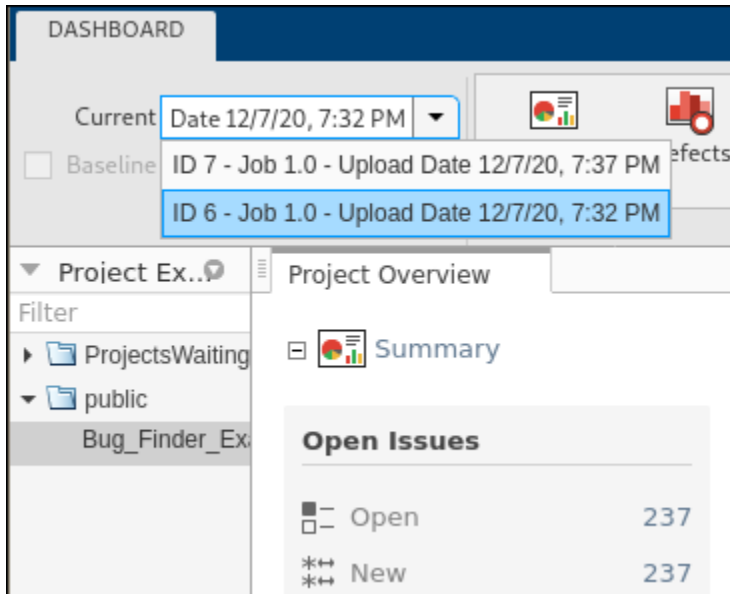
See Also

Related Examples

- “Filter Results” on page 3-4
- “Customize Software Quality Objectives” on page 1-15

Compare Analysis Results to Previous Runs

When you open Polyspace analysis results in the Polyspace Access **DASHBOARD** or **REVIEW**, you see a snapshot of the most recent run that was uploaded to the project. To view a snapshot from an earlier run, select that run from the **Current** run drop-down list.



Select a previous run to see the state of your project from a few submissions ago. For instance, you might want to investigate a spike in findings in a previous version of your project. When you view an older project run in the **DASHBOARD** or **REVIEW** views, all the information for the currently selected run is displayed, except:

- The **Quality Objectives** settings and the **Review History** pane show the same information no matter which run you select.
- You cannot edit the **Result Details** fields if the selected run is not the latest run.

If you share a finding URL from an older run, the Polyspace Access interface opens that finding in the most recent version of the project. If the finding is not present in the most recent run, through the interface, you can open the finding in the older run.

Comparison Mode

To compare two runs in a project, on the toolbar, select a **Current** run, and then select a **Baseline** run. Ensure that the **Baseline** checkbox is enabled. You can compare current runs to only older baseline runs.

The screenshot displays the 'DASHBOARD' view of a software tool. The top navigation bar includes 'Current' and 'Baseline' dropdown menus, both set to 'ID 6 - Job 1.0 - Uploac'. The main content area is divided into several sections:

- Project Overview:** Contains a 'Summary' table comparing 'Baseline Run' and 'Current Run' across various metrics.
- Details:** A table showing the status of findings (Resolved, New, Unresolved) for Defects, Custom Rules, and Polyspace Guidelines.
- Project Explorer:** Shows the project structure, including 'Bug_Finder_Example (Bug Finder)'.

Comparison	Baseline Run	Current Run
Number of Files	14	14
Number of Lines Without Comm...	5201	5201
Defects - Total	242	-
Defects - Density	36	0
Coding Standards - Total	49	-
Coding Standards - Density	9	0

Name	Resolved	New	Unre
Defects	188	-	
Custom Rules	45	-	
Polyspace Guidelines	4	-	

In the **DASHBOARD** view, the comparison shows a summary of statistics for each run and details of the number of findings that are:

- **Resolved:** Findings from the baseline run that are **Done** in the current run, or findings that are not in the current run because they are **Fixed**. Findings are **Done** if they have a status of Justified, No Action Planned, or Not A Defect. Findings are **Fixed** if they are fixed in the source code or the source code containing the finding is deleted or no longer part of the analysis.
- **New:** Findings that are in the current run but not in the baseline run.
- **Unresolved:** Findings that are in the baseline run and the current run.

The comparison mode is not available for the **Code Metrics** and **Quality Objectives** dashboards.

Click a cell in the **Details** table to open the corresponding results in the **Results List**. If a finding is **Resolved**, the interface displays the **Source Code** and **Result Details** information from the **Baseline** run.

In the **REVIEW** view, in addition to **Resolved**, **New**, and **Unresolved**, you can filter findings by **Fixed**. These findings are no longer in the current run because they are fixed, or the source code containing the findings is deleted or no longer part of the analysis.

The total number of findings displayed in the **Results List** corresponds to the findings from the **Current** run and the findings from the **Baseline** run that are **Fixed** in the **Current** run.

To turn off the comparison mode, deselect the **Baseline** checkbox or select **None** in the **Baseline** drop-down list.

See Also

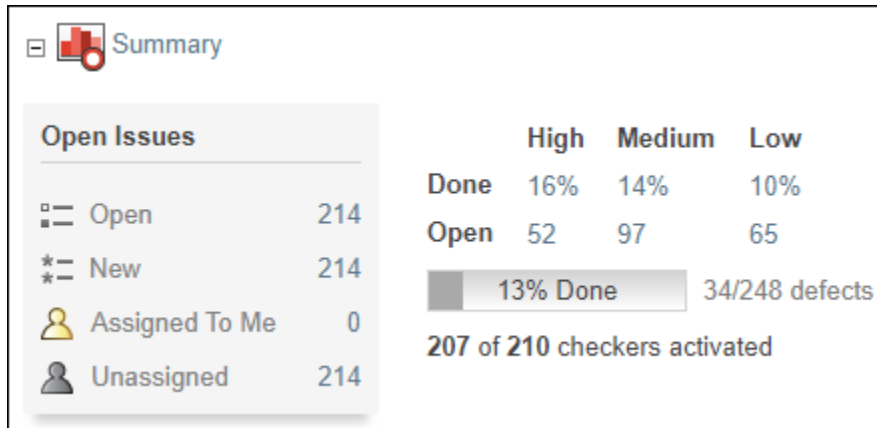
Related Examples

- “Address Results in Polyspace Access Through Bug Fixes or Justifications” on page 2-2
- “Filter Results” on page 3-4

Classification of Defects by Impact

To prioritize your review of Polyspace Bug Finder defects, you can use the **Impact** attribute assigned to the defect. This attribute appears on:

- The **Summary** section of the **Defects** dashboard.



You can view at a glance whether you have many high impact defects, and how many defects are still open. Open defects are defects that have a status **Unreviewed**, **To Investigate**, **To Fix**, or **Other**. You can click a number to open the corresponding set of results in the **Results List** pane. See “Filter and Sort Results in Polyspace Access Web Interface” on page 3-2.

- The **Results List** pane, in the **REVIEW** view. Use the drop-down selection under the **Defects** button in the toolstrip.

You can filter out low and/or medium impact defects using this button. See “Filter and Sort Results in Polyspace Access Web Interface” on page 3-2.

- The **Result Details** pane, beside the defect name.

The impact is assigned to a defect based on the following considerations:

- Criticality, or whether the defect is likely to cause a code failure.

If a defect is likely to cause a code to fail, it is treated as a high impact defect. If the defect currently does not cause code failure but can cause problems with code maintenance in the future, it is a low impact defect.

- Certainty, or the rate of false positives.

For instance, the defect **Integer division by zero** is a high-impact defect because it is almost certain to cause a code crash. On the other hand, the defect **Dead code** has low impact because by itself, presence of dead code does not cause code failure. However, the dead code can hide other high-impact defects.

You cannot change the impact assigned to a defect.

High Impact Defects

The following list shows the high-impact defects.

C++ Exception

- Noexcept function exits with exception
- Throw argument raises unexpected exception

Concurrency

- Data race on adjacent bit fields
- Data race
- Data race through standard library function call
- Deadlock
- Double lock
- Double unlock
- Missing unlock

Data Flow

- Non-initialized pointer
- Non-initialized variable

Dynamic Memory

- Deallocation of previously deallocated pointer
- Invalid deletion of pointer
- Invalid free of pointer
- Use of previously freed pointer

Numerical

- Absorption of float operand
- Float conversion overflow
- Float division by zero
- Integer conversion overflow
- Integer division by zero
- Invalid use of standard library floating point routine
- Invalid use of standard library integer routine

Object Oriented

- Base class assignment operator not called
- Copy constructor not called in initialization list
- Object slicing

Performance

- Empty destructors may cause unnecessary data copies
- Inefficient string length computation

- `std::endl` may cause an unnecessary flush

Programming

- Assertion
- Character value absorbed into EOF
- Declaration mismatch
- Errno not reset
- Incorrect value forwarding
- Invalid use of `==` (equality) operator
- Invalid use of standard library routine
- Invalid `va_list` argument
- Misuse of `errno`
- Misuse of narrow or wide character string
- Misuse of return value from nonreentrant standard function
- Move operation on `const` object
- Non-compliance with AUTOSAR specification
- Possible misuse of `sizeof`
- Possibly unintended evaluation of expression because of operator precedence rules
- Typedef mismatch
- Variable length array with nonpositive size
- Writing to `const` qualified object
- Wrong type used in `sizeof`

Resource Management

- Closing a previously closed resource
- Resource leak
- Use of previously closed resource
- Writing to read-only resource

Security

- Bad order of dropping privileges
- Privilege drop not verified
- Returned value of a sensitive function not checked
- Unsafe call to a system function
- Use of non-secure temporary file

Static Memory

- Array access out of bounds
- Buffer overflow from incorrect string format specifier

- Destination buffer overflow in string manipulation
- Destination buffer underflow in string manipulation
- Invalid use of standard library memory routine
- Invalid use of standard library string routine
- Null pointer
- Pointer access out of bounds
- Pointer or reference to stack variable leaving scope
- Subtraction or comparison between pointers to different arrays
- Use of automatic variable as putenv-family function argument
- Use of path manipulation function without maximum sized buffer checking
- Wrong allocated object size for cast

Medium Impact Defects

The following list shows the medium-impact defects.

C++ Exception

- Exception caught by value
- Exception handler hidden by previous handler

Concurrency

- Asynchronously cancellable thread
- Atomic load and store sequence not atomic
- Atomic variable accessed twice in an expression
- Automatic or thread local variable escaping from a thread
- Data race including atomic operations
- Destruction of locked mutex
- Join or detach of a joined or detached thread
- Missing or double initialization of thread attribute
- Missing lock
- Multiple mutexes used with same conditional variable
- Thread-specific memory leak
- Use of undefined thread ID

Cryptography

- Constant block cipher initialization vector
- Constant cipher key
- Context initialized incorrectly for cryptographic operation
- Context initialized incorrectly for digest operation
- Incompatible padding for RSA algorithm operation
- Inconsistent cipher operations

- Incorrect key for cryptographic algorithm
- Missing blinding for RSA algorithm
- Missing block cipher initialization vector
- Missing certification authority list
- Missing cipher algorithm
- Missing cipher data to process
- Missing cipher final step
- Missing cipher key
- Missing data for encryption, decryption or signing operation
- Missing final step after hashing update operation
- Missing hash algorithm
- Missing padding for RSA algorithm
- Missing parameters for key generation
- Missing peer key
- Missing private key for X.509 certificate
- Missing private key
- Missing public key
- Missing salt for hashing operation
- Missing X.509 certificate
- No data added into context
- Nonsecure hash algorithm
- Nonsecure parameters for key generation
- Nonsecure RSA public exponent
- Nonsecure SSL/TLS protocol
- Predictable block cipher initialization vector
- Predictable cipher key
- Server certificate common name not checked
- TLS/SSL connection method not set
- TLS/SSL connection method set incorrectly
- Weak cipher algorithm
- Weak cipher mode
- Weak padding for RSA algorithm
- X.509 peer certificate not checked

Data Flow

- Pointer to non-initialized value converted to const pointer
- Unreachable code
- Useless if

Dynamic Memory

- Memory leak

Numerical

- Bitwise operation on negative value
- Integer constant overflow
- Integer overflow
- Sign change integer conversion overflow
- Use of plain char type for numerical value

Object Oriented

- Base class destructor not virtual
- Byte-wise operations on nontrivial class object
- Conversion or deletion of incomplete class pointer
- Copy operation modifying source operand
- Incompatible types prevent overriding
- Member not initialized in constructor
- Missing virtual inheritance
- Operator new not overloaded for possibly overaligned class
- Partial override of overloaded virtual functions
- Return of non const handle to encapsulated data member
- Self assignment not tested in operator

Performance

- Const `std::move` input may cause a more expensive object copy
- Expensive `c_str()` to `std::string` construction
- Expensive constant `std::string` construction
- Expensive copy in a range-based for loop iteration
- Expensive local variable copy
- Expensive logical operation
- Expensive pass by value
- Expensive return by value
- Inefficient string length computation
- Missing `constexpr` specifier
- `std::move` called on an unmovable type

Programming

- Abnormal termination of exit handler
- Bad file access mode or status
- Call through non-prototyped function pointer

- Copy of overlapping memory
- Environment pointer invalidated by previous operation
- Exception caught by value
- Exception handler hidden by previous handler
- Floating point comparison with equality operators
- Function called from signal handler not asynchronous-safe
- Function called from signal handler not asynchronous-safe (strict)
- Improper array initialization
- Incorrect data type passed to `va_arg`
- Incorrect pointer scaling
- Incorrect type data passed to `va_start`
- Incorrect use of `offsetof` in C++
- Incorrect use of `va_start`
- Inline constraint not respected
- Invalid assumptions about memory organization
- Invalid file position
- Invalid use of `=` (assignment) operator
- Memory comparison of padding data
- Memory comparison of strings
- Missing byte reordering when transferring data
- Misuse of `errno` in a signal handler
- Misuse of sign-extended character value
- Shared data access within signal handler
- Side effect in arguments to unsafe macro
- Signal call from within signal handler
- Standard function call with incorrect arguments
- Too many `va_arg` calls for current argument list
- Unnamed namespace in header file
- Unsafe conversion between pointer and integer
- Use of indeterminate string
- Use of `memset` with size argument zero

Resource Management

- Opening previously opened resource

Security

- Deterministic random output from constant seed
- `Errno` not checked
- Execution of a binary from a relative path can be controlled by an external actor

- File access between time of check and use (TOCTOU)
- File descriptor exposure to child process
- File manipulation after chroot without chdir
- Hard-coded sensitive data
- Inappropriate I/O operation on device files
- Incorrect order of network connection operations
- Load of library from a relative path can be controlled by an external actor
- Mismatch between data length and size
- Misuse of readlink()
- Predictable random output from predictable seed
- Sensitive data printed out
- Sensitive heap memory not cleared before release
- Uncleared sensitive data in stack
- Unsafe standard encryption function
- Unsafe standard function
- Vulnerable permission assignments
- Vulnerable pseudo-random number generator

Static Memory

- Unreliable cast of function pointer
- Unreliable cast of pointer

Tainted Data

- Array access with tainted index
- Command executed from externally controlled path
- Execution of externally controlled command
- Host change using externally controlled elements
- Library loaded from externally controlled path
- Loop bounded with tainted value
- Memory allocation with tainted size
- Tainted sign change conversion
- Tainted size of variable length array
- Use of externally controlled environment variable

Low Impact Defects

The following list shows the low-impact defects.

Concurrency

- Blocking operation while holding lock
- Function that can spuriously fail not wrapped in loop

- Function that can spuriously wake up not wrapped in loop
- Multiple threads waiting on same condition variable
- Signal call in multithreaded program
- Use of signal to kill thread

Data Flow

- Code deactivated by constant false condition
- Dead code
- Missing return statement
- Partially accessed array
- Static uncalled function
- Variable shadowing
- Write without a further read

Dynamic Memory

- Alignment changed after memory reallocation
- Mismatched alloc/dealloc functions on Windows
- Unprotected dynamic memory allocation

Good Practice

- Ambiguous declaration syntax
- Bitwise and arithmetic operation on a same data
- C++ reference to const-qualified type with subsequent modification
- C++ reference type qualified with const or volatile
- Delete of void pointer
- File does not compile
- Hard coded buffer size
- Hard coded loop boundary
- Hard-coded object size used to manipulate memory
- Incorrect syntax of flexible array member size
- Incorrectly indented statement
- Line with more than one statement
- Macro terminated with a semicolon
- Macro with multiple statements
- Missing break of switch case
- Missing overload of allocation or deallocation function
- Missing reset of a freed pointer
- Possibly inappropriate data type for switch expression
- Redundant expression in sizeof operand
- Semicolon on same line as if, for or while statement

- Unmodified variable not const-qualified
- Unused parameter
- Use of a forbidden function
- Use of setjmp/longjmp

Numerical

- Float overflow
- Integer precision exceeded
- Possible invalid operation on boolean operand
- Precision loss from integer to float conversion
- Shift of a negative value
- Shift operation overflow
- Unsigned integer constant overflow
- Unsigned integer conversion overflow
- Unsigned integer overflow

Object Oriented

- `*this` not returned in copy assignment operator
- Lambda used as typeid operand
- Missing explicit keyword

Performance

- A move operation may throw
- Const parameter values may cause unnecessary data copies
- Const return values may cause unnecessary data copies
- Const rvalue reference parameter may cause unnecessary data copies
- Empty destructors may cause unnecessary data copies
- Expensive use of non-member `std::string operator+()` instead of a simple `append`
- Expensive use of `std::string` methods instead of more efficient overload
- Expensive use of `std::string` with empty string literal
- `std::endl` may cause an unnecessary flush
- Use of `new` or `make_unique` instead of more efficient `make_shared`

Programming

- Accessing object with temporary lifetime
- Alternating input and output from a stream without flush or positioning call
- Call to `memset` with unintended value
- Format string specifiers and arguments mismatch
- Memory comparison of float-point values

- Missing null in string array
- Misuse of a FILE object
- Misuse of structure with flexible array member
- Modification of internal buffer returned from nonreentrant standard function
- Overlapping assignment
- Predefined macro used as an object
- Preprocessor directive in macro argument
- Qualifier removed in conversion
- Return from computational exception signal handler
- Side effect of expression ignored
- Stream argument with possibly unintended side effects
- Universal character name from token concatenation
- Unsafe string to numeric value conversion

Security

- Function pointer assigned with absolute address
- Information leak via structure padding
- Missing case for switch condition
- Umask used with chmod-style arguments
- Use of dangerous standard function
- Use of obsolete standard function
- Vulnerable path manipulation

Static Memory

- Arithmetic operation with NULL pointer

Tainted Data

- Pointer dereference with tainted offset
- Tainted division operand
- Tainted modulo operand
- Tainted NULL or non-null-terminated string
- Tainted string format
- Use of tainted pointer

See Also

More About

- “Filter and Sort Results in Polyspace Access Web Interface” on page 3-2

Bug Finder Defect Groups

In this section...

“Concurrency” on page 3-22
“Cryptography” on page 3-23
“Data flow” on page 3-23
“Dynamic Memory” on page 3-23
“Good Practice” on page 3-23
“Numerical” on page 3-24
“Object Oriented” on page 3-24
“Programming” on page 3-24
“Resource Management” on page 3-25
“Static Memory” on page 3-25
“Security” on page 3-25
“Tainted data” on page 3-25

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)**. See the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server.
- When reviewing results, you can review all results of a certain group together. Filter out other results during review. See “Manage Results”.

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”.

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”.

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”.

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”.

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”.

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”.

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”.

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”.

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”.

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”.

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”.

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”.

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”.

Command-Line Parameter: `tainted_data`

See Also

Troubleshooting Polyspace Access

Polyspace Access ETL and Web Server services do not start

Issue

You start the Polyspace Access services but after a moment, the **ETL** and **Web Server** services stop. You might see a HTTP 403 error message in your web browser when you try to connect to Polyspace Access.

Possible Cause: Hyper-V Network Configuration Cannot Resolve Local Host Names

On Windows®, if you installed Polyspace Access inside a virtual machine (VM), that VM is managed by Hyper-V. Depending on your network configuration, Hyper-V might not resolve local host names. The **Polyspace Access ETL** and **Polyspace Access Web Server** services cannot connect to the host that you specify with these host names.

To test whether Hyper-V can resolve host name `myHostname` on a machine that is connected to the Internet, at the command line, enter:

```
docker run --rm -it alpine ping myHostname
```

If Hyper-V cannot resolve the host name, you get an error message.

Solution

Stop and restart the `admin-docker-agent` binary without using the `--hostname` option.

- If you are on a trusted network or you do not want to use the HTTPS protocol:
 - 1 At the command-line, enter:

```
docker stop admin  
  
admin-docker-agent --restart-gateway
```
 - 2 In the **Cluster Admin** web interface, click **Restart Apps**.
- If you want to use the HTTPS protocol, generate certificates with a subject alternative name (SAN) that includes the IP address of the cluster operator node on which the services are running.
 - 1 Copy this configuration file to a text editor and save it on your machine as `openssl.cnf`.

Configuration file

```
[ req ]  
req_extensions = v3_req  
distinguished_name = req_distinguished_name  
prompt = no  
  
[ req_distinguished_name ]  
countryName = US  
stateOrProvinceName = yourState  
organizationName = myCompany  
organizationalUnitName = myOrganization  
emailAddress = user@email.com  
commonName = hostName
```

```
[ v3_req ]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = hostName
DNS.2 = fullyQualifiedDomainName
IP.1 = nodeIPAddress
```

hostName is the host name of the server that hosts Polyspace Access.
fullyQualifiedDomainName is the corresponding fully qualified domain name.
nodeIPAddress is the IP address of the node on which you run the `admin-docker-agent` binary.

You do not need to edit the value of the other fields in the `[req_distinguished_name]` section of `openssl.cnf`. Updating the value of these fields does not affect the configuration.

- 2 In the **Cluster Dashboard**, click **Configure Nodes**. The IP address listed in the **Hostname** field corresponds to *nodeIPAddress* in the `openssl.cnf` file. If there is more than one node listed, add an additional line in the `[alt_names]` section of `openssl.cnf` for each IP address. For example:

```
[ alt_names ]
DNS.1 = hostName
DNS.2 = fullyQualifiedDomainName
IP.1 = nodeIPAddress
IP.2 = additionalNodeIPAddress
```

- 3 Generate a certificate signing request (CSR) by using your `openssl.cnf` configuration file. At the command line, enter:

```
openssl req -new -out myRequest.csr -newkey rsa:4096 \
-keyout myKey.key -nodes -config openssl.cnf
```

The command outputs a private key file `myKey.key` and the file `myRequest.csr`.

- 4 To generate a signed certificate:
 - If you use your organization's certificate authority, submit `myRequest.csr` to the certificate authority. The certificate authority uses the file to generate a signed server certificate. For instance, `server_cert.cer`.
 - If you use self-signed certificates, at the command line, enter:

```
openssl x509 -req -days 365 -in myRequest.csr -signkey myKey.key \
-out self-cert.pem -extensions v3_req -extfile openssl.cnf
```

The command outputs self-signed certificate `self-cert.pem`.

- 5 Stop and restart the `admin-docker-agent` binary with this command:

Windows PowerShell	<code>./admin-docker-agent --restart-gateway \ --ssl-cert-file certFile1 \ --ssl-key-file keyFile \ --ssl-ca-file trustedStoreFile</code>
Linux®	<code>./admin-docker-agent --restart-gateway \ --ssl-cert-file certFile1 \ --ssl-key-file keyFile \ --ssl-ca-file trustedStoreFile</code>

certFile1 is the full path of the file you obtained in step 4. *keyFile* is the file you generated in step 3. *trustedStoreFile* is the file you generated in step 4 if you used self-signed certificates. Otherwise, it is the trust store file you use to configure HTTPS. See “Choose Between HTTP and HTTPS Configuration for Polyspace Access” Save your changes.

- 6 In the **Cluster Admin** web interface, click **Restart Apps**.

Contact Technical Support About Polyspace Access Issues

If you need support from MathWorks for Polyspace Access issues, go to this page and create a service request. You need a MathWorks login and password to create a service request.


Before you contact MathWorks, gather this information.

- **Operating system**

To see information about the operating system of the machine where you install Polyspace access, at the command line, enter:


Windows	<code>systeminfo findstr /C:OS</code>
Linux	<code>uname -a</code>

- **Polyspace Access version**

Log into Polyspace Access, then at the top of the window click  > **About Polyspace**. If Polyspace Access is not yet installed or you cannot log into Polyspace Access, at the command line, navigate to the folder where you unzipped the Polyspace Access installation image, and enter:

Windows	<code>type VERSION</code>
Linux	<code>cat VERSION</code>

- **License number**

Log into Polyspace Access, then at the top of the window click  > **About Polyspace**. If Polyspace Access is not yet installed or you cannot log into Polyspace Access, contact your license administrator to obtain your license number.

- **Polyspace Access service logs**

To generate logs for the different Polyspace Access services, at the command line, enter:

```
docker logs -f polyspace-access-web-server-main >> access-web-server.log 2>&1
```

```
docker logs -f polyspace-access-etl-main >> access-etl.log 2>&1
```

```
docker logs -f polyspace-access-db-main >> access-db.log 2>&1
```

```
docker logs -f issuetracker-server-main >> issuetracker-server.log 2>&1
```

```
docker logs -f issuetracker-ui-main >> issuetracker-ui.log 2>&1
```

```
docker logs -f usermanager-server-main >> usermanager-server.log 2>&1
```

```
docker logs -f admin >> admin.log 2>&1
```

```
docker logs -f gateway >> gateway.log 2>&1
```

```
docker logs -f usermanager-ui-main >> usermanager-ui.log 2>&1
```

```
docker logs -f usermanager-db-main >> usermanager-db.log 2>&1
```

```
docker logs -f polyspace-access >> polyspace-access.log 2>&1
```

```
docker logs -f issuetracker >> issuetracker.log 2>&1
```

```
docker logs -f usermanager>> usermanager.log 2>&1
```

Attach the log files to your service request. The commands to generate these logs are the same for Windows and Linux.

- **Polyspace Access web interface log**

To generate a log for the Polyspace Access web interface, log into Polyspace Access. In the left pane, click **SUPPORT REPORT** then **Get support info**. Attach the generated support report file to your service request.

Resolve -xml-annotations-description Errors

Issue

When you use the option `-xml-annotations-description` to apply custom annotations to your Polyspace results, some custom annotations are not applied and you see warnings in the console output or the desktop interface **Output Summary**.

Possible Solutions

Custom Annotation Not Found in Mapping

If you define a custom annotation syntax but you do not map it to the Polyspace annotation syntax, Polyspace detects the custom annotation but does not apply it to the analysis results. You see a warning similar to this warning in the console output or the Polyspace desktop interface **Output Summary**.

```
Verifying sources ...
Verifying zero_div.c (1/1)
Warning: rule :50 from exampleCustomAnnotation not found in the mapping (XML file).
        Skipping the annotation
```

Solution

Check the `<Mapping/>` section of the XML file that you pass to the `-xml-annotations-description` option. If the rule listed in the warning is not mapped to a Polyspace rule, add the appropriate entry to map the rule. For instance, to map rule 50 from the preceding warning to Polyspace coding rule **MISRA C: 2012 Rule 8.4**, add this entry in the `<Mapping/>` section:

```
<Result_Name_Mapping Rule_Identifier="50" Family="MISRA-C3" Result_Name="8.4"/>
```

Polyspace Annotations Do Not Apply to Current Code

If you define a custom annotation syntax and you map it to the Polyspace annotation syntax, Polyspace might not apply some custom annotations to your source code. You see a warning similar to this warning in the console output or the Polyspace desktop interface **Output Summary**.

```
Warning: These Polyspace annotations do not apply to the current code:
|   In file D:\Polyspace\Examples\zero_div.c line 7, annotation MISRA-C3:8.4 with text
|   "Justified by annotation in source"
|   In file D:\Polyspace\Examples\zero_div.c line 20, annotation MISRA-C3:8.4 with text
|   "Justified by annotation in source"
|   Possible reasons:
|     - Issue not detected with selected configuration options.
|     - Issue is fixed.
|     - Annotation syntax is incorrect
```

Solution

Check for these possible causes:

- The issue that the annotation addresses has been fixed in the source code. Polyspace detects the custom annotation but ignores it.
- The issue that the annotation addresses was not detected by Polyspace with the analysis options that you specified. For example, if the custom annotation addresses a MISRA C: 2012 coding standard violation but Polyspace did not check for violations of this coding standard because option `Check MISRA C:2012 (-misra3)` is not specified.
- The issue that the annotation addresses was detected but Polyspace could not match the custom annotation to a corresponding Polyspace annotation. This indicates a syntax error in the

<Mapping/> section of the XML file that you pass to the `-xml-annotations-description` option.

See Also

`-xml-annotations-description`

Related Examples

- “Define Custom Annotation Format” on page 2-28

Configure Polyspace as You Code

Configure Polyspace as You Code Extension in Visual Studio

After installing the Polyspace as You Code analysis engine and Visual Studio extension, configure the extension so that a Polyspace analysis runs smoothly when you save your code or explicitly start an analysis. An analysis has run smoothly if results appear as expected, either as source code markers with tooltips or in a list on the **Results List** pane.

To configure the extension, in Visual Studio, select **Tools > Options**. Specify the various settings on the **Polyspace** node.

- The settings on the **General** subnode apply to any project in Visual Studio.
- The settings on the **Project** subnode apply to the project that is currently open.

All settings retain their current values when you reinstall the extension.

General Settings

Setting	Description
Analysis launch mode	<p>Select whether Polyspace as You Code runs:</p> <ul style="list-style-type: none"> • Automatically(default): Analysis starts on each file save. • Manually: User explicitly starts the analysis. <p>To start an analysis, right-click in the source code and select Run Polyspace analysis.</p>
Polyspace as You Code installation folder	<p>Polyspace as You Code installation folder. This field is read-only and set at the time of installation.</p> <p>If you see errors on the Output pane about starting the Polyspace Connector, check if the folder still exists (and contains a Polyspace as You Code installation).</p>
Working directory for extension	<p>Folder where analysis results are stored. When you start an analysis, a subfolder is created in this folder per Visual Studio solution. Within a subfolder, a second subfolder is created per project and then another per file.</p> <p>For each file, a new run overwrites results of the previous run. If the analysis fails for a given file, you can check the failed subfolder for information useful for troubleshooting, such as the options given to the analysis engine.</p> <p>The default results folder is C:\TEMP\%USERNAME%\Polyspace.</p>

Setting	Description
Polyspace Access configuration > Polyspace Access URL	<p>URL of the Polyspace Access instance from which you get a baseline.</p> <p>After you obtain a baseline from Polyspace Access, subsequent runs of Polyspace as You Code allow you to distinguish between new results and results that were present in existing code.</p> <p>See also “Baseline Polyspace as You Code Results in Visual Studio” on page 5-41.</p>

Project Settings

Build configuration

Setting	Description
Get from solution (default)	The analysis builds the solution, traces the build, and generates a build options file. See “Get Build Configuration from Visual Studio Solution” on page 5-25.
Get from build command line	<p>The analysis traces the build command that you specify and generates a build options file. Specify:</p> <ul style="list-style-type: none"> • The build command in the setting Build command line • The folder from which the build command must be launched in the setting Working directory. <p>See “Get Build Configuration from Build Command” on page 5-26.</p>
Get from JSON Compilation Database	<p>The analysis extracts the build configuration from the JSON compilation database that you specify and generates a build options file. See “Get Build Configuration from JSON Compilation Database” on page 5-26.</p> <p>Specify the path to the JSON file (typically named <code>compile_commands.json</code>) in the setting Path to JSON file.</p>
Get from Polyspace build options file	The analysis uses manually specified options. Provide these options in the options file that you specify in the setting Analysis options file . See “Specify Analysis Options Manually” on page 5-27.
Build options file not required	<p>You do not have to specify Polyspace options related to your building configuration. This is a basic option for simple projects.</p> <p>The analysis uses the default Polyspace build options. So that the analysis runs without errors, you typically should provide Polyspace as You Code with the specificities of your build configuration.</p>

Analysis configuration

Setting	Description
Checkers file	<p>Path to a checkers configuration file.</p> <p>To create this file, click Open. Enable the checkers that you want and save the file.</p> <p>See also “Configure Checkers for Polyspace as You Code in Visual Studio” on page 5-61.</p>
Analysis options file	<p>Path to an options file. The options file contains one Polyspace analysis option per line. For example:</p> <pre>-D _WIN32 -termination-functions exit_handler</pre> <p>You typically do not need to specify additional options in an options file. However, in some situations, you might want to use an options file. For instance, if you want to manually specify Polyspace options related to your build command, select Set manually for the setting Build configuration and enter the options in an options file.</p> <p>See also “Options Files for Polyspace Analysis” on page 5-22.</p>
Import options from Polyspace Desktop project (*.psprj)	<p>Import the analysis options and checkers configuration file from existing Polyspace desktop project file. See “Import Analysis Options from Polyspace Desktop Project” on page 5-28.</p>

Polyspace Access project configuration

Setting	Description
Use baseline from Polyspace Access	<p>Specify whether a baseline must be used and the project on Polyspace Access that you get the baseline from. Enter the project name in the Project path field.</p> <p>See also “Baseline Polyspace as You Code Results in Visual Studio” on page 5-41.</p>
Show only new findings compared to the results baseline	<p>Specify whether only new results must be shown. If you select this option, results are compared with the baseline downloaded from Polyspace Access and only new results are shown.</p> <p>See also “Baseline Polyspace as You Code Results in Visual Studio” on page 5-41.</p>

Expert configuration

Setting	Description
Run analysis using scripts	<p>Run a script each time you save your code (or explicitly run analysis). The script takes the path to the current file as the first argument and the Working directory for extension as the second argument.</p> <p>For example, this simple Windows batch script analyzes the current file and uses the default Polyspace build options:</p> <pre>set INSTALL_DIR=C:\Program Files\Polyspace as You Code\R2021a set ANALYZE=%INSTALL_DIR%\bin\polyspace-bug-finder-access.exe set SOURCES=%1 set RESULTS_FOLDER=%2 "%ANALYZE%" -sources %SOURCES% -results-dir %RESULTS_FOLDER% IF %ERRORLEVEL% NEQ 0 EXIT 1</pre> <p>Use a script if, for instance, you switch between files from components that have different build configurations or you use a custom tool to setup your build environment.</p> <p>If you enable this setting, all other extension settings are ignored.</p> <p>Note The Polyspace as You Code extension does not check the exit status of the commands in your script. Make sure your script checks exit codes (for instance by using %ERRORLEVEL%) and returns a meaningful exit status.</p>
Analysis script	<p>Enter the full path to a script. The script can be written in any language.</p> <p>Each time you run Polyspace as You Code on file save or explicitly, you run this script with two arguments: the full path to the current file and the results folder.</p>

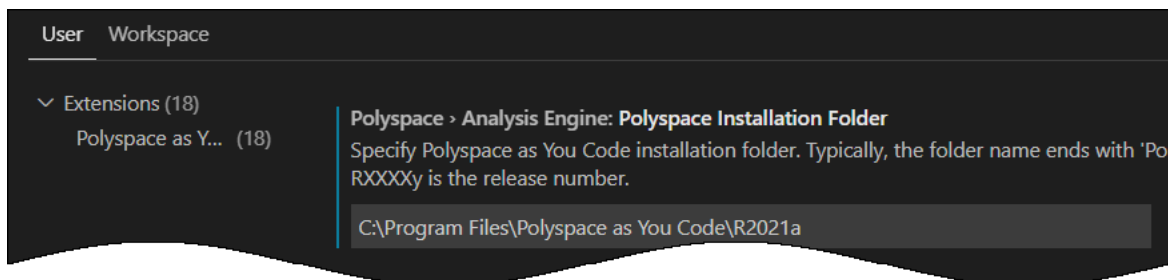
Configure Polyspace as You Code Extension in Visual Studio Code

After installing the Polyspace as You Code analysis engine and IDE extension, configure the extension so that a Polyspace analysis runs smoothly when you save your code or explicitly start an analysis. An analysis has run smoothly if results appear as expected, either as source code markers with tooltips or in a list on the **PROBLEMS** pane.

To configure the extension, in Visual Studio Code, open the settings interface by pressing **Ctrl + ,** (comma) and type `polyspace` in the settings search bar.

For each setting, you can specify a value that applies globally to all workspaces or folders that you open in the Visual Studio editor. For some of the settings, you can also override the global specification with a workspace-specific value.

- To specify global settings, enter the settings on the **User** tab.
- To override the global settings for the currently open workspace or folder, enter the settings on the **Workspace** tab.



To reset a setting to its default value, click the  icon on the left of the setting and select **Reset Setting**. All settings retain their current values when you reinstall the extension.

Tip Type the Setting ID in the settings search bar to view only the settings related to that ID.

Analysis Engine

Setting ID: `polyspace.analysisengine`

These settings are mandatory.

Setting	Description	Available Per Workspace?
Polyspace Installation Folder	Root folder of the Polyspace as You Code installation, for instance, C:\Program Files\Polyspace as You Code\R2021a.	No
Result Folder	Folder where analysis results are stored. Each new run overwrites results of the previous run.	Yes

Analysis Launch Mode

Setting ID: polyspace.analysisoptions.analysislaunchmode

By default, Polyspace as You Code runs each time you save your code. You can choose to disable the automatic runs.

Setting	Description	Available Per Workspace?
Analysis Options: Analysis Launch Mode	<p>Select when Polyspace as You Code must run:</p> <ul style="list-style-type: none"> • Automatically (default): The analysis must run on each save. • Manually: Choose to explicitly start the analysis. You can right-click a source file and select Polyspace: Analyze Current File (or run the same command from the Command Palette). 	Yes

Analysis Setup

Setting ID: polyspace.analysisoptions.analysissetup

You can set up a Polyspace as You Code analysis through extension settings or override extension settings and run a script instead. By default, the analysis uses extension settings.

Setting	Description	Available Per Workspace?
Analysis Options: Analysis Setup	<p>Select between manual setup and script.</p> <ul style="list-style-type: none"> • Manual Setup (default): Set up Polyspace as You Code through extension settings. Specify build-related and other options through the Manual Setup group of settings. • Script: Run a script each time you save your code (or right-click a source file and select Polyspace: Analyze Current File). The script takes the path to the current file as the first argument and the results folder as the second argument. All other extension settings are ignored. 	Yes

Analysis Options > Manual Setup

Setting ID: `polyspace.analysisoptions.manualsetup`

Manual setup of the analysis involves specifying build options, checkers and other analysis options. Extract build options from a Visual Studio Code build task or a JSON Compilation Database file, or specify them explicitly in a build options file. Enable or disable checkers in a checkers selection window. Specify all remaining analysis options explicitly in an options file.

Setting	Description	Available Per Workspace?
Analysis Options > Manual Setup: Build	<p>Specification of build-related Polyspace analysis options. Options are:</p> <ul style="list-style-type: none"> • Build options file not required (default): You do not have to specify Polyspace options related to building your files. This is a basic option for simple projects where the default Polyspace analysis options are sufficient to compile the files. • Get from JSON Compilation Database file: The analysis must create build options from a JSON compilation database. Specify the path to the database file (typically named <code>compile_commands.json</code>) in the setting Analysis Options > Manual Setup > Build Setting: JSON Compilation Database File. <p>Later, when you run Polyspace: Analyze Build in the Command Palette, the <code>polyspace-configure</code> command creates build-related Polyspace analysis options from this database file using the option <code>-compilation-database</code>. Subsequent runs of Polyspace as You code use these options.</p> <ul style="list-style-type: none"> • Get from build task: The analysis must extract build options from a Visual Studio Code build task. Use a build task that performs a complete build of all files in your workspace. Specify the build task name in the setting Analysis Options > 	Yes

Setting	Description	Available Per Workspace?
	<p>Manual Setup > Build Setting: Build Task.</p> <p>Later, when you run Polyspace: Analyze Build in the Command Palette, the <code>polyspace-configure</code> command runs on the command underneath this build task and creates Polyspace analysis options. Subsequent runs of Polyspace as You code use these options.</p> <ul style="list-style-type: none"> • Get from build command: The analysis must extract build options from a build command. Make sure that the command builds all source files in your workspace. Specify the build command in the setting Analysis Options > Manual Setup > Build Setting: Build Command. <p>Later, when you run Polyspace: Analyze Build in the Command Palette, the <code>polyspace-configure</code> command runs on the build command and creates Polyspace analysis options. Subsequent runs of Polyspace as You code use these options.</p> <ul style="list-style-type: none"> • Get from Polyspace build options file: Provide the build options in the options file that you specify in the setting Analysis Options > Manual Setup > Build Setting: Polyspace Build Options File. 	

Setting	Description	Available Per Workspace?
Analysis Options > Manual Setup > Build Setting: Build Command	<p>Use this setting if you choose Get from build command for the setting Analysis Options > Manual Setup: Build.</p> <p>Specify the build command name exactly as you would enter on a command-line terminal or console.</p> <p>Use a build command that performs a complete build of all files in your workspace and not an incremental build.</p> <p>See “Get Build Configuration from Build Command” on page 5-30</p>	Yes
Analysis Options > Manual Setup > Build Setting: Build Task	<p>Use this setting if you choose Get from build task for the setting Analysis Options > Manual Setup: Build.</p> <p>Specify the build task name. The build task name is the name of a command that runs when you select Terminal > Run Task. For more information on tasks, see Visual Studio Code documentation.</p> <p>Use a build task that performs a complete build of all files in your workspace and not an incremental build.</p> <p>See “Get Build Configuration from Build Task” on page 5-29.</p>	Yes

Setting	Description	Available Per Workspace?
<p>Analysis Options > Manual Setup > Build Setting: JSON Compilation Database File</p>	<p>Use this setting if you choose Get from JSON Compilation Database File for the setting Analysis Options > Manual Setup: Build.</p> <p>Specify the full path to a database file (typically named <code>compile_commands.json</code>).</p> <p>See “Get Build Configuration from JSON Compilation Database” on page 5-30.</p>	<p>Yes</p>
<p>Analysis Options > Manual Setup > Build Setting: Polyspace Build Options File</p>	<p>Use this setting if you choose Get from Polyspace Build Options File for the setting Analysis Options > Manual Setup: Build.</p> <p>Specify the full path to a Polyspace build options file. The options file is a text file with one Polyspace analysis option per line.</p> <p>See also “Options Files for Polyspace Analysis” on page 5-22.</p>	<p>Yes</p>
<p>Analysis Options > Manual Setup: Checkers File</p>	<p>Specify the full path to a checkers configuration file.</p> <p>To create this file, in the Command Palette, run Polyspace: Configure Checkers. Enable the checkers that you want and save the file.</p> <p>See also “Configure Checkers for Polyspace as You Code in Visual Studio Code” on page 5-64.</p>	<p>Yes</p>

Setting	Description	Available Per Workspace?
Analysis Options > Manual Setup: Other Analysis Options	<p>Path to an options file. The options file contains one Polyspace analysis option per line. For example:</p> <pre data-bbox="654 447 1049 506">-termination-functions exit_handler -code-behavior-specifications /usr/jdoe/util/checkerModifiers.xml</pre> <p>You typically do not need to specify additional options in an options file. However, in some situations, you might want to use an options file. For instance, you might want to modify some checkers using an XML file that you provide with the option <code>-code-behavior-specifications</code>.</p> <p>See also “Options Files for Polyspace Analysis” on page 5-22.</p>	Yes

Analysis Options > Script

Setting ID: polyspace.analysisoptions.scriptfile

Setting	Description	Available Per Workspace?
<p>Analysis Options > Script: Script File</p>	<p>Use this setting if you choose Script for the setting Analysis Options: Analysis Setup.</p> <p>Enter the full path to a script. The script can be written in any language.</p> <p>Each time you run Polyspace as You Code on file save or explicitly, you run this script with two arguments: the full path to the current file and the Result Folder.</p> <p>For example, this simple Windows batch script analyzes the current file and uses the default Polyspace build options:</p> <pre>set INSTALL_DIR=C:\Program Files\Polyspace as You Code\R2021a set ANALYZE=%INSTALL_DIR%\bin\polyspace-bug-finder-access.exe set SOURCES=%1 set RESULTS_FOLDER=%2 "%ANALYZE%" -sources %SOURCES% -results-dir %RESULTS_FOLDER% IF %ERRORLEVEL% NEQ 0 EXIT 1</pre> <p>Use a script if, for instance, you switch between files from components that have different build configurations or you use a custom tool to setup your build environment.</p> <p>If you enable this setting, all other extension settings are ignored.</p> <hr/> <p>Note The Polyspace as You Code extension does not check the exit status of the commands in your script. Make sure your script checks exit codes (for instance by using %ERRORLEVEL%) and returns a meaningful exit status.</p>	<p>Yes</p>

Baseline

Setting ID: polyspace.baseline

These options are essential only if you want to obtain a baseline from Polyspace Access. After you obtain a baseline from Polyspace Access, subsequent runs of Polyspace as You Code allow you to distinguish between new results and results that were present in existing code. See also “Baseline Polyspace as You Code Results in Visual Studio Code” on page 5-45.

Setting	Description	Available Per Workspace?
Baseline: Polyspace Access Url	Specify Polyspace Access URL.	No
Baseline: Polyspace Access Login	Specify the user name you use to log in to Polyspace Access. Later, when you run Polyspace: Get Baseline in the Command Palette , you are prompted for the password corresponding to this user name.	No
Baseline: Project	Specify a project on Polyspace Access that you use as baseline. Later, when you run Polyspace: Get Baseline in the Command Palette , the <code>polyspace-access</code> command runs with the <code>-download</code> option to download the baseline results from the latest run of this project. Subsequent runs of Polyspace as You Code use this baseline.	Yes
Baseline: Show Only New Findings	Suppress findings that are already present in the Polyspace Access project that you use as baseline. If you select this setting, you must also select Baseline: Use Baseline .	Yes

Setting	Description	Available Per Workspace?
Baseline: Use Baseline	<p>Use Polyspace Access project as baseline.</p> <p>Results that are already present in Polyspace Access show associated review information. If a result has review information that indicates a justified status, for instance, No action planned, the result is not shown at all.</p> <p>In addition, if you select the setting Baseline: Show Only New Findings, results that are already present in Polyspace Access are not shown at all.</p>	Yes

Trace

Setting ID: polyspace.trace.server

The option is useful only for troubleshooting by technical support. You typically do not need to use this option.

Setting	Description	Available Per Workspace?
Trace: Server	<p>Whether the log must show messages related to communication between the internal server that hosts analysis results and the IDE. This option is only useful for troubleshooting internal communication issues.</p>	Yes

Configure Polyspace as You Code Plugin in Eclipse

This topic describes how to configure the Polyspace as You Code plugin in Eclipse™. For Polyspace desktop products such as Polyspace Bug Finder, see the topic "Polyspace Analysis in Eclipse" in the Polyspace Bug Finder documentation.

After installing the Polyspace as You Code analysis engine and Eclipse plugin, configure the plugin so that a Polyspace analysis runs smoothly when you save your code or explicitly start an analysis. An analysis has run smoothly if results appear as expected, either as source code markers with tooltips or in a list on the **Results List** pane.

To configure the plugin, in Eclipse, select items from the **Polyspace** menu.

- To specify options that are applicable to any project in Eclipse, select **Polyspace > Preferences**.
- To specify options that are applicable to a single project only, select **Polyspace > Configure Project**.

The name of the project that you configure is listed in the title bar of the **Configure Project** window.

All settings retain their current values when you reinstall the plugin.

Preferences

Setting	Description
Polyspace as You Code installation folder	Root folder of the Polyspace as You Code installation, for instance, C:\Program Files\Polyspace as You Code\R2021a.
Analysis launch mode	Choose one of these options to trigger the Polyspace as You Code analysis: <ul style="list-style-type: none"> • Automatically(default): The analysis starts each time you save a file. • Manually: To start an analysis, right-click the source file and select Run Polyspace as You Code.
Results folder	Folder where analysis results are stored. Each new run overwrites results of the previous run. The default results folder is: <ul style="list-style-type: none"> • Windows: <i>Documents</i> \Polyspace_Workspace \EclipseProjects where <i>Documents</i> is the Documents folder in Windows. • Linux: ~/Polyspace_Workspace/EclipseProjects

Setting	Description
Polyspace Access URL	<p>URL of the Polyspace Access instance from which you get a baseline.</p> <p>After you obtain a baseline from Polyspace Access, subsequent runs of Polyspace as You Code allow you to distinguish between new results and results that were present in existing code.</p> <p>See also “Baseline Polyspace as You Code Results in Eclipse” on page 5-50.</p>
Show only new findings compared to the results baseline	<p>Specify whether only new results must be shown. If you select this option, results are compared with the baseline downloaded from Polyspace Access and only new results are shown.</p> <p>See also “Baseline Polyspace as You Code Results in Eclipse” on page 5-50.</p>

Configure Project

Build configuration

Setting	Description
Build options file not required	<p>You do not have to specify Polyspace options related to your building configuration. This is a basic option for simple projects.</p> <p>The analysis uses the default Polyspace build options. So that the analysis runs without errors, you typically should provide Polyspace as You Code with the specificities of your build configuration.</p>
Get from Eclipse project (default)	<p>The analysis extracts the build configuration from the Eclipse project and generates a build options file.</p> <p>See “Get Build Configuration from Eclipse Project” on page 5-34.</p>
Get from Polyspace build options file	<p>The analysis uses build options that you manually specify in an options file. Provide the full path to the options file.</p>

Setting	Description
Get from JSON Compilation Database file	<p>The analysis extracts the build configuration from the JSON compilation database that you specify and generates a build options file. See “Get Build Configuration from JSON Compilation Database” on page 5-35.</p> <p>Specify the full path to the JSON file (typically named <code>compile_commands.json</code>). Then click Generate build configuration.</p>
Get from build command	<p>The analysis traces the build command that you specify and generates a build options file.</p> <p>Specify the build command and the folder from which the build command must be launched in setting Build command working folder. Then click Generate build configuration. See “Get Build Configuration from Build Command” on page 5-34.</p>

Other Analysis settings

Setting	Description
Checkers file	<p>Path to a checkers configuration file.</p> <p>To create this file, click the folder icon. Enable the checkers that you want and save the file.</p> <p>See also “Configure Checkers for Polyspace as You Code in Eclipse” on page 5-57.</p>
Analysis options file	<p>Path to an options file. The options file contains one Polyspace analysis option per line. For example:</p> <pre>-D _WIN32 -termination-functions exit_handler</pre> <p>You typically do not need to specify additional options in an options file. However, in some situations, you might want to use an options file. For instance, if you want to manually specify Polyspace options related to your build command, select None for build setting and enter the options in an options file.</p> <p>See also “Options Files for Polyspace Analysis” on page 5-22.</p>
Import options from Polyspace desktop project (*.psrpj)	<p>Import the analysis options and checkers configuration file from existing Polyspace desktop project file. See “Import Analysis Options from Polyspace Desktop Project” on page 5-36.</p>

Polyspace Access settings

Setting	Description
Use baseline from Polyspace Access	<p>Specify whether to get a baseline for Polyspace results.</p> <p>If you enable this setting a specify a Project path, click Download baseline from Polyspace Access to download a baseline.</p> <p>After you obtain a baseline from Polyspace Access, subsequent runs of Polyspace as You Code allow you to distinguish between new results and results that were present in existing code.</p> <p>See also “Baseline Polyspace as You Code Results in Eclipse” on page 5-50.</p>
Project path	<p>Path of project in Polyspace Access Project Explorer that you get the baseline from.</p>

Expert configuration

Setting	Description
Run script for Polyspace analysis	<p>Run a script each time you save your code (or explicitly run analysis). The script takes the path to the current file as the first argument and the Results folder as the second argument.</p> <p>For example, this simple Windows batch script analyzes the current file and uses the default Polyspace build options:</p> <pre>set INSTALL_DIR=C:\Program Files\Polyspace as You Code set ANALYZE=%INSTALL_DIR%\bin\polyspace-bug-finder-ac set SOURCES=%1 set RESULTS_FOLDER=%2 "%ANALYZE%" -sources %SOURCES% -results-dir %RESULTS IF %ERRORLEVEL% NEQ 0 EXIT 1</pre> <p>Use a script if, for instance, you switch between files from components that have different build configurations or you use a custom tool to setup your build environment.</p> <p>If you enable this setting, all other extension settings are ignored.</p> <p>Note The Polyspace as You Code extension does not check the exit status of the commands in your script. Make sure your script checks exit codes (for instance by using %ERRORLEVEL%) and returns a meaningful exit status.</p>
Analysis script	<p>Enter the full path to a script. The script can be written in any language.</p> <p>Each time you run Polyspace as You Code on file save or explicitly, you run this script with two arguments: the full path to the current file and the results folder.</p>

Options Files for Polyspace Analysis

To adapt the Polyspace analysis configuration to your development environment and requirements, you have to modify the default configuration through command-line options such as `-compiler`. Options files are a convenient way to collect multiple options together and reuse them across projects.

What are Options Files

Options files are text files with one option per line. For instance, the content of an options file can look like this:

```
# Options for Polyspace analysis
# Options apply to all projects in Controller module
-compiler visual16.x
-D _WIN32
-code-behavior-specifications "Z:\utils\polyspace\forbiddenfunctions.xml"
```

The lines starting with `#` represent comments for better readability. These lines are ignored during analysis.

Specifying Options Files

Depending on the platform where you run analysis, you can specify an options file in one of the following ways.

Command Line

At the command line (and in scripts), specify an options file as argument to the option `-options-file`.

For instance, instead of the command:

```
polyspace-bug-finder -sources file.c -compiler visual16.x -D _WIN32
-code-behavior-specifications "Z:\utils\polyspace\forbiddenfunctions.xml"
```

Save this content:

```
-compiler visual16.x
-D _WIN32
-code-behavior-specifications "Z:\utils\polyspace\forbiddenfunctions.xml"
```

In a file `options.txt` in the path `Z:\utils\polyspace\` and shorten the command to:

```
polyspace-bug-finder -sources file.c -options-file "Z:\utils\polyspace\options.txt"
```

You can use options files with these Polyspace commands:

- `polyspace-bug-finder`

- `polyspace-bug-finder-server`
- `polyspace-bug-finder-access`
- `polyspace-code-prover`
- `polyspace-code-prover-server`

IDEs

If you run Polyspace as You Code using IDE extensions, you typically specify three groups of options differently:

- *Build options:*

You can extract build options from existing artifacts such as build commands and JSON compilation database, or collect all build options in an options file. You can specify this options file in the appropriate extension setting:

- Visual Studio Code: **Analysis Options > Manual Setup > Build Setting : Polyspace Build Options File**
- Visual Studio: **Get from Polyspace build options file** (in section **Build Configuration**)
- Eclipse: **Get from Polyspace build options file** (in section **Build Configuration**)
- *Checkers:*

You can specify checkers using a checkers selection wizard. For details, see “Setting Checkers in Polyspace as You Code”.

- *Other remaining options:*

All remaining options can be collected in a second options file that goes into the appropriate extension setting:

- Visual Studio Code: **Analysis Options > Manual Setup: Other Analysis Options**
- Visual Studio: **Analysis configuration > Analysis options file**
- Eclipse: **Analysis options file**

If you use options files both for build options and other options, the result is the same as specifying a single options file with the other options appended to the build options. See also “Specifying Multiple Options Files” on page 5-24.

For more information on IDE extensions, see:

- “Configure Polyspace as You Code Extension in Visual Studio” on page 5-2
- “Configure Polyspace as You Code Extension in Visual Studio Code” on page 5-6
- “Configure Polyspace as You Code Plugin in Eclipse” on page 5-17

Polyspace User Interface

In the user interface of the Polyspace desktop products, you typically do not require an options file. Most options can be specified on the **Configuration** pane in the Polyspace user interface.

However, some options are available only at the command line and do not have a counterpart in the user interface. If you have to specify multiple command-line-only options, you can collect them in an options file, for instance `commandLineStyleOptions.txt`. On the **Configuration** pane, under the **Advanced Settings** node, you can enter the following in the **Other** field:

```
-options-file commandLineStyleOptions.txt
```

Specifying Multiple Options Files

You can specify multiple options files in an analysis. For instance, at the command line, you can enter:

```
polyspace-bug-finder -sources file.c -options-file opts1.txt -options-file opts2.txt
```

When you specify multiple options files in an analysis, all options from the options files are appended to the analysis command. For instance, the preceding command has the same effect as using a single options file that places the content of `opts1.txt` above `opts2.txt`.

If an option appears in multiple files with conflicting arguments, the argument in the last options file prevails. For instance, in the preceding command, if `opts1.txt` contains:

```
-checkers all  
-misra3 all
```

And `opts2.txt` contains:

```
-misra3 single-unit-rules
```

The analysis uses only the argument `single-unit-rules` for the option `-misra3`.

You can use this stacking of options files to override options. For instance, suppose you use a read-only options file that applies to your entire team but want to override some of the options in the file. You can override the options by using a second options file that you create and specifying your options file *after* the team-wide options file.

You can also specify the option `-options-file` within an options file and aggregate several options files in this way.

See Also

Related Examples

- “Run Polyspace as You Code from Command Line and Export Results” on page 6-14

Generate Build Options for Polyspace as You Code Analysis in Visual Studio

Polyspace as You Code checks the source code file that is currently active in your Visual Studio® IDE for bugs and coding standards violations.

So that the analysis runs without errors, provide Polyspace as You Code with the specificities of your build configuration, such as data type sizes and compiler macro definitions. To provide your build configuration information, you can:

- Configure Polyspace as You Code to extract the build configuration information from your Visual Studio solution, build command, or JSON compilation database.
- Manually specify analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 5-22.
- Import the analysis options from a Polyspace desktop product project file.

Configure Polyspace as You Code to Extract Build Configuration

To extract your build configuration information from the Visual Studio solution, build command, or JSON compilation database:

- 1 Go to **Tools > Options**.
- 2 Select the appropriate **Build configuration** option on the **Project** subnode of the **Polyspace** node. See “Configure Polyspace as You Code Extension in Visual Studio” on page 5-2.

The **Build configuration** option that you select applies to all the projects in the Visual Studio solution.

Polyspace extracts the build information and generates an options file that the Polyspace as You Code analysis engine uses in subsequent analyses. The file contains analysis options that emulate your build configuration. Make sure that the build completes successfully before you extract the build information.

The generated options file is stored in the `.polyspace-configure` folder under the `workingDirectory/projectName` folder or one of its subfolders. The `workingDirectory` path is the **Working directory of extension** folder path that you specify in the **General** options of the Polyspace extension. The `projectName` is the name of the project that contains the files you are currently analyzing.

When you configure Polyspace as You Code to extract your build configuration information from the Visual Studio solution, build command, or JSON compilation database, the software extracts the build configuration only if:

- You start an analysis and Polyspace cannot find a generated options file in the `.polyspace-configure` folder for the project that contains the currently analyzed file.
- You explicitly regenerate an options file, for instance after you make changes to your build configuration. See “Update Generated Build Options File” on page 5-27.

Get Build Configuration from Visual Studio Solution

To extract your build configuration information from the Visual Studio solution:

- 1 Go to **Tools > Options**.
- 2 Select **Get from solution** on the **Project** subnode of the **Polyspace** node. See “Configure Polyspace as You Code Extension in Visual Studio” on page 5-2.

The **Build configuration** option that you select applies to all the projects in the Visual Studio solution.

Polyspace builds your solution, traces the build to extract the configuration information, and generates an options file.

Get Build Configuration from Build Command

To extract your build configuration information from your build command:

- 1 Go to **Tools > Options**.
- 2 Select **Get from build command line** on the **Project** subnode of the **Polyspace** node. See “Configure Polyspace as You Code Extension in Visual Studio” on page 5-2.
- 3 Specify your build command in the **Build command line** field. The build command that you specify must perform a full build. For instance:

```
"C:\Program Files\Polyspace as You Code\R2021a\sys\tcc\win64\tcc.exe" -g -o output dll.c fib.c hello_dll.c hello_win.c
```

- 4 Specify the full path of the folder where Polyspace runs the build command in the **Working directory** field. For instance:

```
C:\Program Files\Polyspace as You Code\R2021a\sys\tcc\win64\examples
```

The **Build configuration** option that you select applies to all the projects in the Visual Studio solution.

Polyspace runs your build command, traces the build to extract the configuration information, and generates an options file.

Get Build Configuration from JSON Compilation Database

If your build system supports the generation of a JSON compilation database file, use this setting. The file contains compiler calls for all the translation units in your project. See JSON compilation database.

To extract your build configuration information from the JSON compilation database:

- 1 Generate a JSON compilation database file. For an example of how to generate this file, see “Create Polyspace Options File from JSON Compilation Database”.

If you use a JSON compilation database that was not generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

- 2 Go to **Tools > Options**.
- 3 Select **Get from JSON Compilation Database** on the **Project** subnode of the **Polyspace** node. See “Configure Polyspace as You Code Extension in Visual Studio” on page 5-2.
- 4 Specify the full path to the JSON compilation database file that you generated in step 1 in the **Path to JSON file** field.

The **Build configuration** option that you select applies to all the projects in the Visual Studio solution.

Polyspace extracts the build configuration information from the compilation database and generates an options file.

If you make changes to your build configuration, regenerate the compilation database file before you update the generated options file.

Update Generated Build Options File

If you make changes to your build configuration, for instance if you add a source file to your project or workspace or rename an existing file, update the generated options file to reflect those changes. Before you update the options file, make sure that your build completes successfully with the new configuration.

To update the options file, from the project context menu in the **Solution Explorer**, select **Generate Polyspace build configuration**.

If you extract your build information from a JSON compilation database file, regenerate the compilation database before you update the build options file.

See also “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 5-83.

Specify Analysis Options Manually

Use this setting if:

- You know the details of your build system and you want to specify the Polyspace analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 5-22.

For a list of available analysis options, see “Polyspace as You Code Analysis Engine Options”.

- You reuse a Polyspace options file that you or someone else on your team has configured for your build system.

If you reuse an options file that was not configured or generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

To specify an analysis options file:

- 1 Go to **Tools > Options**.
- 2 Select **Get from Polyspace build options file** on the **Project** subnode of the **Polyspace** node. See “Configure Polyspace as You Code Extension in Visual Studio” on page 5-2.
- 3 Specify the full path to the options file in the **Build options file** field.

The Polyspace as You Code analysis engine uses the specified options file in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 8-2.

Import Analysis Options from Polyspace Desktop Project

If you configure an analysis in the Polyspace desktop product, you can use the information from the resulting Polyspace desktop PSPRJ file to configure your Polyspace as You Code analysis.

To import the analysis options from a Polyspace desktop PSPRJ file:

- 1 Go to **Tools > Options**.
- 2 Select **Build options file not required** on the **Project** subnode of the **Polyspace** node. See “Configure Polyspace as You Code Extension in Visual Studio” on page 5-2.

This selection allows you to leave the **Build options file** field empty.

- 3 Click **Import from Polyspace desktop project** and select the PSPRJ file that you import from.

Polyspace generates an options file and an XML checkers activation file on page 5-61, and populates the corresponding **Analysis configuration** fields. The Polyspace as You Code analysis engine uses these files in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 8-2.

See Also

Related Examples

- “Configure Polyspace as You Code Extension in Visual Studio” on page 5-2
- “Configure Checkers for Polyspace as You Code in Visual Studio” on page 5-61
- “Baseline Polyspace as You Code Results in Visual Studio” on page 5-41

Generate Build Options for Polyspace as You Code Analysis in Visual Studio Code

Polyspace as You Code checks the source code file that is currently active in your Visual Studio Code editor for bugs and coding standards violations.

So that the analysis runs without errors, provide Polyspace as You Code with the specificities of your build configuration, such as data type sizes and compiler macro definitions. To provide your build configuration information, you can:

- Configure Polyspace as You Code to extract the build configuration information from your build task, build command, or JSON compilation database.
- Manually specify analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 5-22.
- Import the analysis options from a Polyspace desktop product project file.

Configure Polyspace as You Code to Extract Build Configuration

To extract your build configuration information from the build task, build command, or JSON compilation database:

- 1 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar and set **Polyspace > Analysis Options: Analysis Setup** to `Manual setup`.

- 2 Set the appropriate **Polyspace > Analysis Options > Manual Setup: Build** options and fill out the corresponding **Build Setting** field.

See “Configure Polyspace as You Code Extension in Visual Studio Code” on page 5-6.

- 3 Open the **Command Palette (Ctrl+Shift+P)** and enter `Polyspace: Analyze Build`.

Polyspace extracts the build information and generates an options file that the Polyspace as You Code analysis engine uses in subsequent analyses. The file contains analysis options that emulate your build configuration.

The generated options file is stored in the `.polyspace-configure` folder under the `workingDirectory/projectName` folder or one of its subfolders.

The `workingDirectory` path is the **Polyspace > Analysis Engine: Result Folder** path that you specify in the Polyspace as You Code extension settings. The `projectName` is the name of the top-level folder in the **EXPLORER** that contains the files that you are currently analyzing.

Get Build Configuration from Build Task

Visual Studio Code enables you to define tasks so that you can run an external tool from your code editor. See Integrate with External Tools via Tasks.

If you define a custom task that calls your compiler to perform a full build of your project, Polyspace can extract your build configuration from this build task.

- 1 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar.

- 2 Set these **Polyspace > Analysis Options** settings to the values listed in the table.

Setting	Value
Analysis Setup	Manual setup
Manual Setup: Build	Get from build task
Manual Setup > Build Setting: Build Task	<p>Specify the name of the build task. This corresponds to the "label" field of the task definition in the <code>tasks.json</code> file. The task that you specify must perform a full build.</p> <p>Polyspace supports the use of only these Visual Studio Code predefined variables in task definitions:</p> <ul style="list-style-type: none"> • <code>\${workspaceFolder}</code> • <code>\${workspaceFolderBasename}</code> • <code>\${cwd}</code>

- 3 Open the **Command Palette (Ctrl+Shift+P)** and enter **Polyspace: Analyze Build**.

Polyspace runs the build command specified by the task, traces the build to extract the configuration information, and generates an options file.

Get Build Configuration from Build Command

To extract your build configuration information from your build command:

- 1 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar.

- 2 Set these **Polyspace > Analysis Options** settings to the values listed in the table.

Setting	Value
Analysis Setup	Manual setup
Manual Setup: Build	Get from build command
Manual Setup > Build Setting: Build Command	<p>Specify your build command, for instance:</p> <pre>"C:\Program Files\Polyspace as You Code\R2021a\sys\tcc\win64\tcc.exe" -g -o output dll.c fib.c hello_dll.c hello_win.c</pre> <p>The command that you specify must perform a full build</p>

- 3 Open the **Command Palette (Ctrl+Shift+P)** and enter **Polyspace: Analyze Build**.

Polyspace runs your build command, traces the build to extract the configuration information, and generates an options file.

Get Build Configuration from JSON Compilation Database

If your build system supports the generation of a JSON compilation database file, use this setting. The file contains compiler calls for all the translation units in your project. See JSON compilation database.

To extract your build configuration information from the JSON compilation database:

- 1 Generate a JSON compilation database file. For an example of how to generate this file, see “Create Polyspace Options File from JSON Compilation Database”.

If you use a JSON compilation database that was not generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

- 2 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar.

- 3 Set these **Polyspace > Analysis Options** settings to the values listed in the table.

Setting	Value
Analysis Setup	Manual setup
Manual Setup: Build	Get from JSON Compilation Database file
Manual Setup > Build Setting: JSON Compilation Database File	Specify the full path to the file that you generated in step 1. The file is typically named <code>compile_commands.json</code> .

- 4 Open the **Command Palette (Ctrl+Shift+P)** and enter **Polyspace: Analyze Build**.

Polyspace extracts the build configuration information from the compilation database and generates an options file.

Update Generated Build Options File

If you make changes to your build configuration, for instance if you add a source file to your project or workspace or rename an existing file, update the generated options file to reflect those changes. Before you update the options file, make sure that your build completes successfully with the new configuration.

To update the options file, Open the **Command Palette (Ctrl+Shift+P)** and enter **Polyspace: Analyze Build**.

If you extract your build information from a JSON compilation database file, regenerate the compilation database before you update the build options file.

See also “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 5-83.

Specify Analysis Options Manually

Use this setting if:

- You know the details of your build system and you want to specify the Polyspace analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 5-22.

For a list of available analysis options, see “Polyspace as You Code Analysis Engine Options”.

- You reuse a Polyspace options file that you or someone else on your team has configured for your build system.

If you reuse an options file that was not configured or generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

To specify an analysis options file:

- 1 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar.

- 2 Set these **Polyspace > Analysis Options** settings to the values listed in the table.

Setting	Value
Analysis Setup	Manual setup
Manual Setup: Build	Get from Polyspace build options file
Manual Setup > Build Setting: Polyspace Build Options File	Specify the full path to the Polyspace options file.

The Polyspace as You Code analysis engine uses the specified options file in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 8-2.

Import Analysis Options from Polyspace Desktop Project

If you configure an analysis in the Polyspace desktop product, you can use the information from the resulting Polyspace desktop PSPRJ file to configure your Polyspace as You Code analysis.

To import the analysis options from a Polyspace desktop PSPRJ file, open a terminal in Visual Studio Code and enter this command:

```
polyspace-checkers-selection -import-options-from-psprj pathToPsprjFile
```

The `polyspace-checkers-selection` binary is available under the `polyspace/bin` folder in your Polyspace as You Code installation folder.

The `pathToPsprjFile` path is the full path of the PSPRJ file.

Polyspace generates an options file (`analysis_options.txt`) and an XML checkers activation file on page 5-64 (`checkers_activation_file.xml`). The generated files are stored in the `import` folder in the same location as the PSPRJ file.

To complete the configuration of the Polyspace as You Code analysis:

- 1 Open the Visual Studio Code settings by pressing **Ctrl+,** (comma).

Enter `polyspace.analysisoptions` in the settings search bar.

- 2 Set these **Polyspace > Analysis Options** settings to the values listed in the table.

Setting	Value
Analysis Setup	Manual setup

Setting	Value
Manual Setup: Build	Build options file not required This setting ignores the file specified in the Build Setting: Polyspace Build Options File field.
Manual Setup: Checkers File	Full file path of checkers_activation_file.xml
Manual Setup: Other Analysis Options	Full file path of analysis_options.txt

The Polyspace as You Code analysis engine uses these files in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 8-2.

See Also

Related Examples

- “Configure Polyspace as You Code Extension in Visual Studio Code” on page 5-6
- “Configure Checkers for Polyspace as You Code in Visual Studio Code” on page 5-64
- “Baseline Polyspace as You Code Results in Visual Studio Code” on page 5-45

Generate Build Options for Polyspace as You Code Analysis in Eclipse

Polyspace as You Code checks the source code of the file that is currently active in your Eclipse IDE for bugs and coding standards violations.

So that the analysis runs without errors, provide Polyspace as You Code with the specificities of your build configuration, such as data type sizes and compiler macro definitions. To provide your build configuration information, you can:

- Configure Polyspace as You Code to extract the build configuration information from your Eclipse project, build command, or JSON compilation database.
- Manually specify analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 5-22.
- Import the analysis options from a Polyspace desktop product project file.

Configure Polyspace as You Code to Extract Build Configuration

To extract your build configuration information from the Eclipse project, build command, or JSON compilation database:

- 1** Go to **Polyspace > Configure Project**.
- 2** Select the appropriate **Build configuration** option. See “Configure Polyspace as You Code Plugin in Eclipse” on page 5-17.

Get Build Configuration from Eclipse Project

To extract your build configuration information from your Eclipse project:

- 1** Go to **Polyspace > Configure Project**.
- 2** Select **Get from Eclipse project**. See “Configure Polyspace as You Code Plugin in Eclipse” on page 5-17.

Each time you start an analysis, Polyspace extracts the build configuration information from the project toolchain and generates an options file. The Polyspace analysis engine uses that options file in the subsequent analysis.

To view the details of the toolchain configuration:

- 1** Select a project in the **Project Explorer** and go to **Project > Properties**.
- 2** Under the **C/C++ General** node, select **Paths and symbols** and **Preprocessor Include Paths, Macros, etc**.

Get Build Configuration from Build Command

To extract your build configuration information from your build command:

- 1** Go to **Polyspace > Configure Project**.
- 2** Select **Get from build command line** and specify your build command. See “Configure Polyspace as You Code Plugin in Eclipse” on page 5-17.

The build command that you specify must perform a full build. For instance:

```
"C:\Program Files\Polyspace as You Code\R2021a\sys\tcc\win64\tcc.exe" -g -o output dll.c fib.c hello_dll.c hello_win.c
```

- 3 Specify the full path of the folder where Polyspace runs the build command in the **Build command working folder** field. For instance:

```
C:\Program Files\Polyspace as You Code\R2021a\sys\tcc\win64\examples
```

- 4 Click **Generate build configuration**.

Polyspace runs your build command, traces the build to extract the configuration information, and generates an options file. The Polyspace as You Code analysis engine uses the generated options file in subsequent analyses.

The generated options file is stored in the `.polyspace-configure` folder under the `workingDirectory/projectName` folder or one of its subfolders. The `workingDirectory` path is the **Results folder** path that you specify in the **Polyspace > Preferences**. The `projectName` is the name of the project that contains the files you are currently analyzing.

Get Build Configuration from JSON Compilation Database

If your build system supports the generation of a JSON compilation database file, use this setting. The file contains compiler calls for all the translation units in your project. See JSON compilation database.

To extract your build configuration information from the JSON compilation database:

- 1 Generate a JSON compilation database file. For an example of how to generate this file, see “Create Polyspace Options File from JSON Compilation Database”.

If you use a JSON compilation database that was not generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

- 2 Go to **Polyspace > Configure Project**.
- 3 Select **Get from JSON Compilation Database file** and specify the full path to the JSON compilation database file that you generated in step 1. See “Configure Polyspace as You Code Plugin in Eclipse” on page 5-17.
- 4 Click **Generate build configuration**.

Polyspace extracts the build configuration information from the compilation database and generates an options file. The Polyspace as You Code analysis engine uses the generated options file in subsequent analyses.

The generated options file is stored in the `.polyspace-configure` folder under the `workingDirectory/projectName` folder or one of its subfolders. The `workingDirectory` path is the **Results folder** path that you specify in the **Polyspace > Preferences**. The `projectName` is the name of the project that contains the files you are currently analyzing.

Update Generated Build Options File

If you make changes to your build configuration, for instance if you add a source file to your project or workspace or rename an existing file, update the generated options file to reflect those changes. Before you update the options file, make sure that your build completes successfully with the new configuration.

To update the options file, select **Polyspace > Generate Build Configuration**. You do not need to update the options file if you extract your build from an Eclipse project.

If you extract your build information from a JSON compilation database file, regenerate the compilation database before you update the build options file.

See also “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 5-83.

Specify Analysis Options Manually

Use this setting if:

- You know the details of your build system and you want to specify the Polyspace analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 5-22.

For a list of available analysis options, see “Polyspace as You Code Analysis Engine Options”.

- You reuse a Polyspace options file that you or someone else on your team has configured for your build system.

If you reuse an options file that was not configured or generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

To specify an analysis options file:

- 1 Go to **Polyspace > Configure Project**.
- 2 Select **Get from Polyspace build options file** and specify the full path to the options file. See “Configure Polyspace as You Code Plugin in Eclipse” on page 5-17.

The Polyspace as You Code analysis engine uses the options file that you specify in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 8-2.

Import Analysis Options from Polyspace Desktop Project

If you configure an analysis in the Polyspace desktop product, you can use the information from the resulting Polyspace desktop PSPRJ file to configure your Polyspace as You Code analysis.

To import the analysis options from a Polyspace desktop PSPRJ file:

- 1 Go to **Polyspace > Configure Project**.
- 2 Select **Build options file not required**. See “Configure Polyspace as You Code Plugin in Eclipse” on page 5-17.

This selection allows you to leave the **Get from Polyspace build options file** field empty.

- 3 Click **Import options from Polyspace desktop project** and select the PSPRJ file that you import from.

Polyspace generates an options file and an XML checkers activation file on page 5-57, and populates the corresponding fields with the paths to the generated files. The Polyspace as You Code analysis engine uses these files in subsequent analyses.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 8-2.

See Also

Related Examples

- “Configure Polyspace as You Code Plugin in Eclipse” on page 5-17
- “Configure Checkers for Polyspace as You Code in Eclipse” on page 5-57
- “Baseline Polyspace as You Code Results in Eclipse” on page 5-50

Generate Build Options for Polyspace as You Code Analysis at the Command Line

Polyspace as You Code checks your code for bugs and coding standards violations while you work in your IDE or code editor.

So that the analysis runs without errors, provide Polyspace as You Code with the specificities of your build configuration, such as data type sizes and compiler macro definitions. To provide your build configuration information, you can:

- Use the `polyspace-configure` binary to extract the build configuration information from your build command or JSON compilation database.
- Manually specify analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 5-22.
- Import the analysis options from a Polyspace desktop product project file.

Use `polyspace-configure` to Generate Build Options File

The `polyspace-configure` binary enables you to extract the build configuration information from a build command or a JSON compilation database file. The binary uses the extracted information to generate a build options file which contains a set of options that emulate your build configuration.

`polyspace-configure` is available with your Polyspace as You Code installation, in the `polyspaceAsYouCodeRoot/polyspace/bin` folder, where `polyspaceAsYouCodeRoot` is your Polyspace as You Code installation folder.

Get Build Configuration from Build Command

To extract the build configuration information from your build command, provide a build command that performs a full build. For instance, if you use `make` on Linux to build your project, use this command:

```
polyspace-configure \
-no-sources -allow-overwrite \
-output-options-file path/To/buildOptions.txt \
-merge-common-options make -B
```

Polyspace runs your build command, traces the build to extract the configuration information, and generates `buildOptions.txt` inside `path/To`. For more information about the `polyspace-configure` options, see `polyspace-configure`.

Use the generated options file in subsequent analyses of source files from your project. For instance:

```
polyspace-bug-finder-access -sources file.c -options-file path/To/buildOptions.txt
```

Get Build Configuration from JSON Compilation Database

If your build system supports the generation of a JSON compilation database file, use this workflow.

The compilation database file contains compiler calls for all the translation units in your project. See JSON compilation database.

To extract your build configuration information from the JSON compilation database:

- 1 Generate a JSON compilation database file. For an example of how to generate this file, see “Create Polyspace Options File from JSON Compilation Database”. The generated file is typically named `compile_commands.json`.

If you use a JSON compilation database that was not generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

- 2 Pass the compilation database file to `polyspace-configure`. For instance:

```
polyspace-configure \  
-no-sources -allow-overwrite \  
-output-options-file path/To/buildOptions.txt \  
-merge-common-options \  
-compilation-database otherPath/To/compile_commands.json
```

Polyspace extracts the build configuration information from the compilation database and generates an options file. For more information about the `polyspace-configure` options, see `polyspace-configure`

Use the generated options file in subsequent analyses of source files from your project. For instance:

```
polyspace-bug-finder-access -sources file.c -options-file path/To/buildOptions.txt
```

Update Generated Build Options File

If you make changes to your build configuration, for instance if you add a source file to your project or workspace or rename an existing file, update the generated options file to reflect those changes. Before you update the options file, make sure that your build completes successfully with the new configuration.

To update the options file, rerun the command that you used to generate the file and specify the same set of options you used.

If you extract your build information from a JSON compilation database file, regenerate the compilation database before you update the build options file.

Specify Analysis Options Manually

Use this workflow if:

- You know the details of your build system and you want to specify the Polyspace analysis options that emulate your build configuration in an options file. See “Options Files for Polyspace Analysis” on page 5-22.

For a list of available analysis options, see “Polyspace as You Code Analysis Engine Options”.

- You reuse a Polyspace options file that you or someone else on your team has configured for your build system.

If you reuse an options file that was not configured or generated on your local machine, make sure that the paths listed in the file are accessible from the location where you run Polyspace as You Code.

If you make changes to your build configuration, edit the options file to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 8-2.

Import Analysis Options from Polyspace Desktop Project

If you configure an analysis in the Polyspace desktop product, you can use the information from the resulting Polyspace desktop PSPRJ file to configure your Polyspace as You Code analysis.

To import the analysis options from a Polyspace desktop PSPRJ file, use this command:

```
polyspace-checkers-selection -import-options-from-psprj pathToPsprjFile
```

The `polyspace-checkers-selection` binary is available under the `polyspace/bin` folder in your Polyspace as You Code installation folder.

The *pathToPsprjFile* path is the full path of the PSPRJ file.

Polyspace generates an options file (`analysis_options.txt`) and an XML checkers activation file on page 5-68 (`checkers_activation_file.xml`). The generated files are stored in the `import` folder in the same location as the PSPRJ file.

Use the generated options file and checkers activation file in subsequent analyses of source files from your project. For instance:

```
polyspace-bug-finder-access -sources file.c \  
-options-file path/To/import/analysis_options.txt \  
-checkers-activation-file path/To/import/checkers_activation_file.xml
```

If you make changes to your build configuration, edit the options file (`analysis_options.txt`) to reflect those changes. See “Specify Target Environment and Compiler Behavior” on page 8-2.

See Also

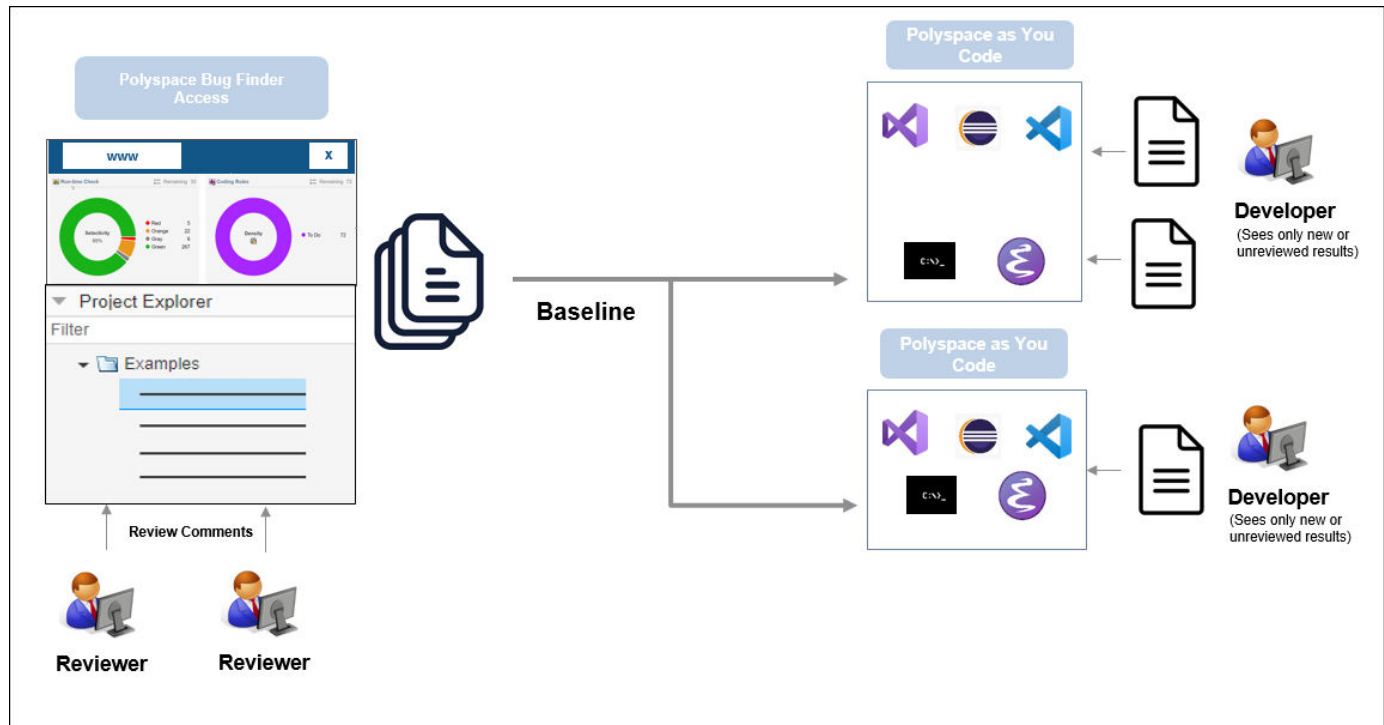
`polyspace-bug-finder-access` | `polyspace-configure`

Related Examples

- “Options Files for Polyspace Analysis” on page 5-22
- “Configure Checkers for Polyspace as You Code at the Command Line” on page 5-68
- “Baseline Polyspace as You Code Results on Command Line” on page 5-53

Baseline Polyspace as You Code Results in Visual Studio

For more efficient bug fixing, you can baseline the results of a Polyspace as You Code analysis with previous results. When you baseline the results, you compare them against the results of a previous run and focus on new results only or on unreviewed results only. You baseline Polyspace as You Code results using previous Polyspace Bug Finder Server results that you download from Polyspace Access.



What Baselined Results Look Like

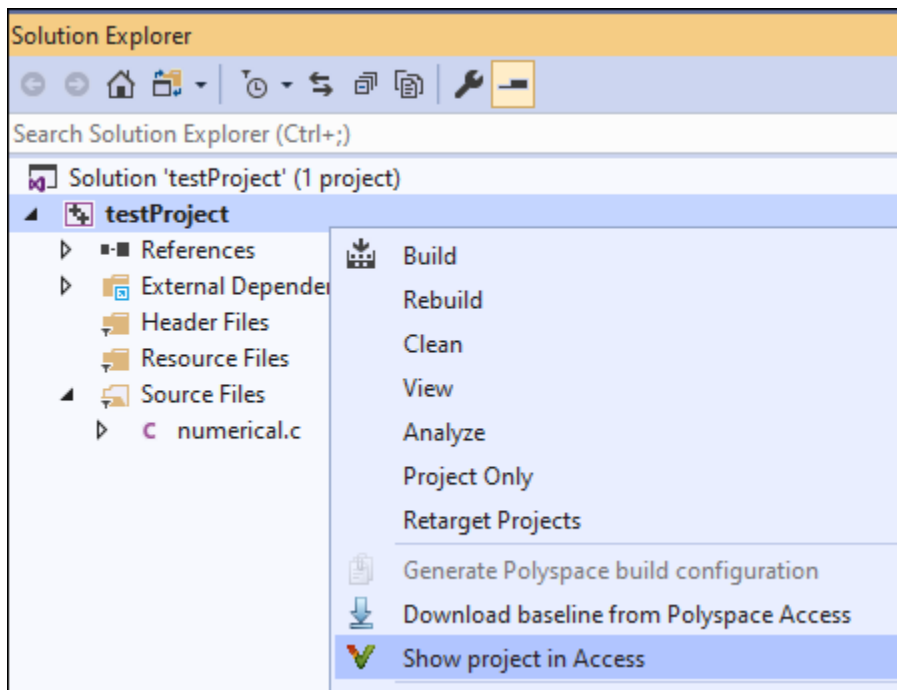
If you baseline Polyspace as You Code results using Polyspace Bug Finder Server results that you downloaded from Polyspace Access, you can see the following benefits:

- Results that have a justified **Status** on Polyspace Access (**No Action Planned**, **Justified**, or **Not a Defect**) are no longer shown.
- Results that have a non-justified **Status** on Polyspace Access carry over all review information to the **Polyspace Results List** pane in Visual Studio. If a result is reviewed in Polyspace Access and marked as such, *one of the following* is true:
 - The **Status** is different from **Unreviewed**.
 - The **Severity** is different from **Unset**.
 - The **Comment** is not blank.

For instance, the fact that the **Float division by zero** defect has associated review information indicates that the defect is also present in the baseline. In Polyspace Access, the defect has been reviewed and assigned a **Status** of **To fix**.

Polyspace Results List											
Family	New	Ln	Ch	Check	Type	Group	Information	Status	Severity	Comment	
○	No	314	16	Invalid use of standard library integer routine	Defect	Numerical	Impact: High	Unreviewed	Unset		
○	No	312	16	Invalid use of standard library integer routine	Defect	Numerical	Impact: High	Unreviewed	Unset		
○	No	185	12	Float conversion overflow	Defect	Numerical	Impact: High	Unreviewed	Unset		
○	No	112	12	Integer conversion overflow	Defect	Numerical	Impact: High	Unreviewed	Unset		
○	No	290	17	Absorption of float operand	Defect	Numerical	Impact: High	Unreviewed	Unset		
○	No	70	16	Float division by zero	Defect	Numerical	Impact: High	To fix	Unset		
○	No	36	14	Integer division by zero	Defect	Numerical	Impact: High	Unreviewed	Unset		

You can also open the Polyspace Access project used as baseline in a web browser. In Visual Studio, right-click the project on the **Solution Explorer** pane and select **Show project in Access**.



- If you specify the Polyspace as You Code extension setting **Show only new findings compared to the results baseline**, you see only results that are new in the current run. That way, you can focus only on results that explicitly occurred because of the changes you made since the last Polyspace Server run.

Baselining Steps

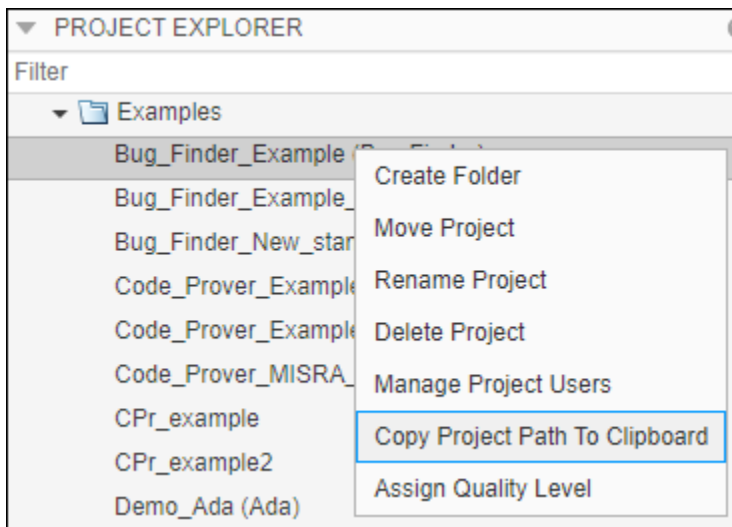
To use Polyspace Bug Finder Server results as baseline for a Polyspace as You Code analysis, follow the steps below. Once a baseline is downloaded, if you choose to point to the baseline, each subsequent run, whether on file save or on-demand, uses the baseline.

Step 1: Identify Project to Use as Baseline

First, identify a project in Polyspace Access that you want to use as baseline. The project must contain results of a Polyspace Bug Finder Server analysis on files that you will analyze in Polyspace as You Code.

Copy the path to the project for use in the Visual Studio Code extension settings. To copy this path:

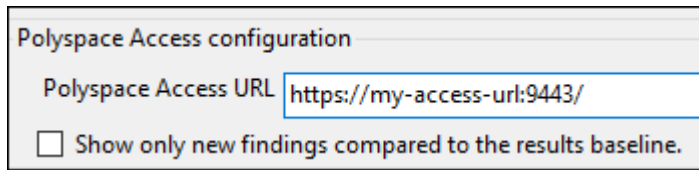
- 1 Open the Polyspace Access web interface in a web browser.
- 2 On the **Project Explorer** pane, right-click the project and select **Copy Project Path to Clipboard**.



Step 2: Refer to Project from Polyspace as You Code

Next, refer to the Polyspace Access project from the Polyspace as You Code extension settings in Visual Studio.

- 1 Open the extension settings.
To open the settings, select **Tools > Options**. The Polyspace as You Code extension settings appear under the **Polyspace** node.
- 2 Specify on the **General** node the **Polyspace Access URL**, that is, the URL of the server that hosts Polyspace Access. For instance, `https://my-access-url:9443/`.



- 3 Specify these settings on the **Project** node:
 - **Use baseline from Polyspace Access:** Select this option to use the project on Polyspace Access as baseline.
 - **Project path:** The path to the project in Polyspace Access that you want to use as baseline. You previously copied this path from the Polyspace Access web interface.
 - **Show only new findings compared to the results baseline:** Select this option to suppress results that are already present in the project in Polyspace Access.

Step 3: Download Baseline

Explicitly download the Polyspace Access result to use as baseline.

- 1 Right-click the project on the **Solution Explorer** pane and select **Download baseline from Polyspace Access**.
- 2 Enter the username and password that you use to log in to Polyspace Access. The baseline download begins.

To follow the progress of download, select **View > Output** and from the dropdown on the top, select **Polyspace**. Wait for the message:

```
Baseline downloaded successfully for Access project ProjectName
```

After download, subsequent runs use the baseline. To disable baseline usage, disable the extension setting **Use baseline from Polyspace Access**.

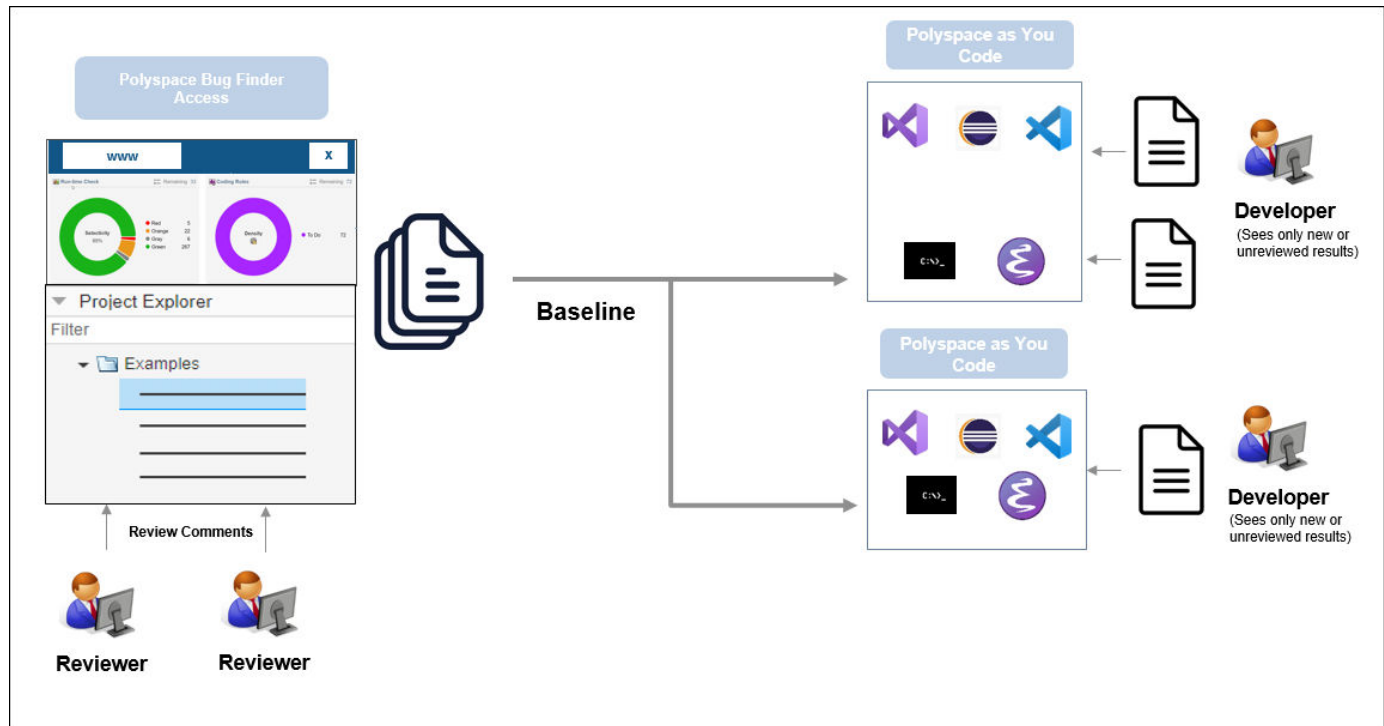
See Also

More About

- “Configure Polyspace as You Code Extension in Visual Studio” on page 5-2
- “Run Polyspace as You Code in Visual Studio and Review Results” on page 6-2

Baseline Polyspace as You Code Results in Visual Studio Code

For more efficient bug fixing, you can baseline the results of a Polyspace as You Code analysis using previous results. When you baseline the results, you compare them against the results of a previous run and focus on new results only or on unreviewed results only. You baseline Polyspace as You Code results using previous Polyspace Bug Finder Server results that you download from Polyspace Access.

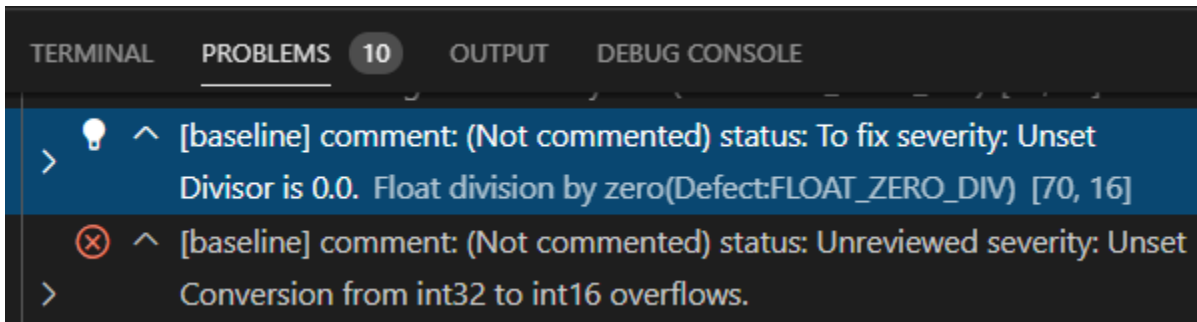


What Baselined Results Look Like

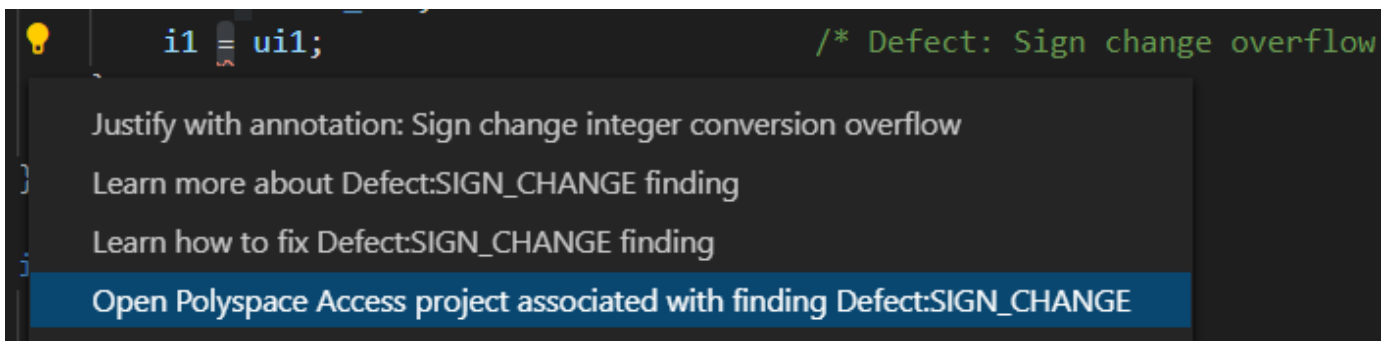
If you baseline Polyspace as You Code results using Polyspace Bug Finder Server results that you downloaded from Polyspace Access, you can see the following benefits:

- Results that have a justified **Status** on Polyspace Access (**No Action Planned**, **Justified**, or **Not a Defect**) are no longer shown.
- Results that have a non-justified **Status** on Polyspace Access carry over all review information to the **PROBLEMS** pane in Visual Studio Code.

For instance, the word [baseline] next to the result below shows that the **Float division by zero** defect is also present in the baseline. In Polyspace Access, the defect has been reviewed and assigned a **Status** of **To fix**.



If a Polyspace as You Code result also appears in the baseline, you can open the result on Polyspace Access in a web browser directly from Visual Studio Code. Click the source code token with a Polyspace as You Code result (wavy red underlining). Then, click the bulb icon that appears and select **Open Polyspace Access project associated with finding**.



- If you specify the Polyspace as You Code extension setting **Show Only New Findings**, you see only results that are new in the current run. That way, you can only focus on results that explicitly occurred because of the changes you made since the last Polyspace Server run.

Baselining Steps

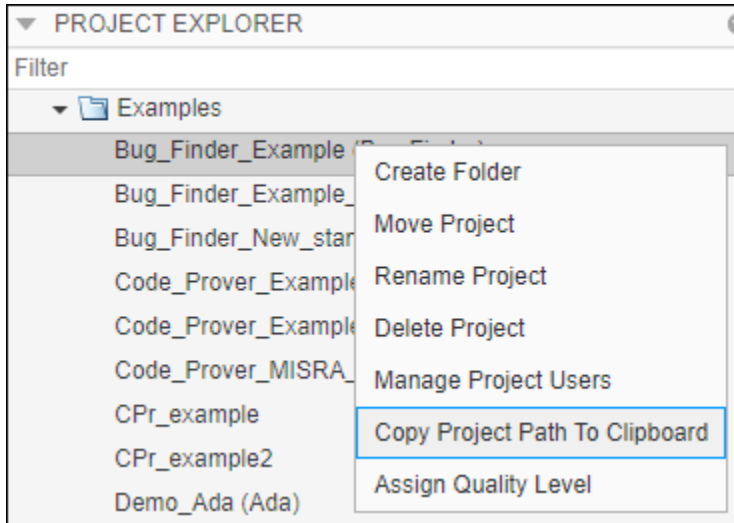
To use Polyspace Bug Finder Server results as baseline for a Polyspace as You Code analysis, follow the steps below. Once a baseline is downloaded, if you choose to point to the baseline, each subsequent run, whether on file save or on-demand, uses the baseline.

Step 1: Identify Project to Use as Baseline

First, identify a project in Polyspace Access that you want to use as baseline. The project must contain results of a Polyspace Bug Finder Server analysis on files that you will analyze in Polyspace as You Code.

Copy the path to the project for use in the Visual Studio Code extension settings. To copy this path:


- 1 Open the Polyspace Access web interface in a web browser.
- 2 On the **Project Explorer** pane, right-click the project and select **Copy Project Path to Clipboard**.



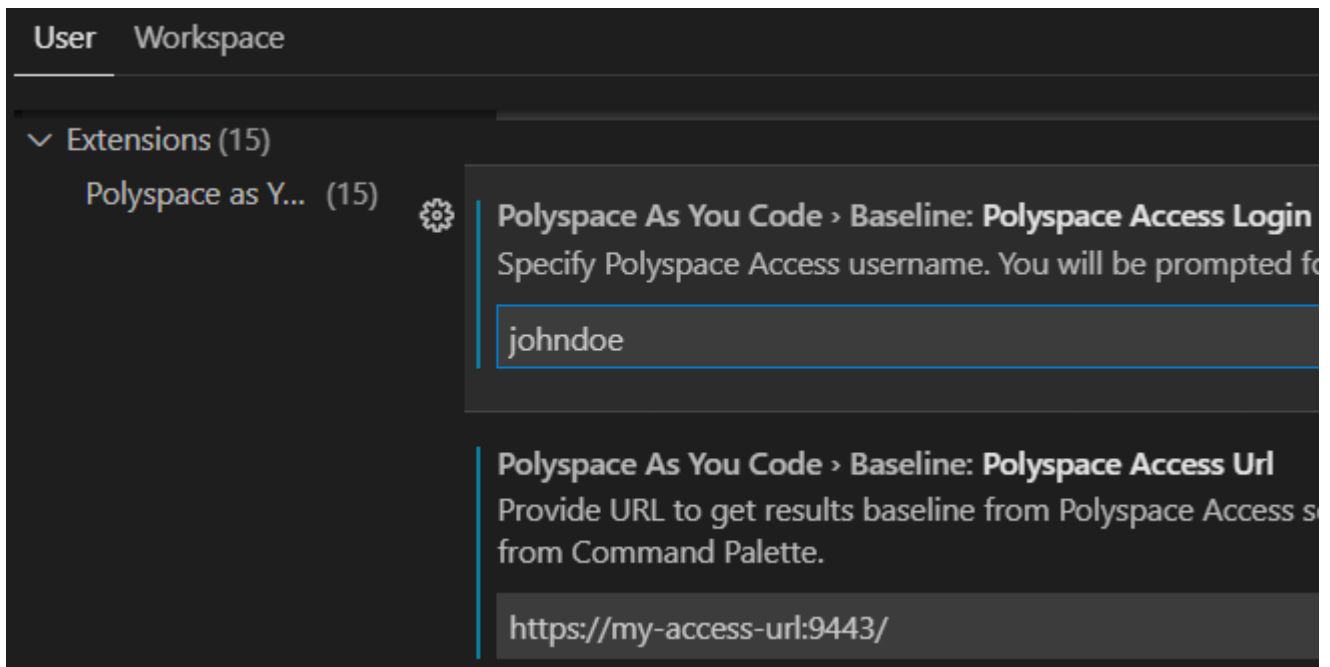
Step 2: Refer to Project from Polyspace as You Code

Next, refer to the Polyspace Access project from the Polyspace as You Code extension settings in Visual Studio Code.

- 1 Open the extension settings.

To open the settings, select **View > Extensions** and then the  icon next to **Polyspace as You Code**. Select **Extension Settings**.

- 2 Specify these settings on the **User** tab:
 - **Baseline: Polyspace Access Url:** The URL of the server that hosts Polyspace Access. For instance, `https://my-access-url:9443/`.
 - **Baseline: Polyspace Access Login:** The username that you use to log in to Polyspace Access.



3 Specify these settings on the **Workspace** tab:

- **Baseline: Project:** The path to the project in Polyspace Access that you want to use as baseline. You previously copied this path from the Polyspace Access web interface.
- **Baseline: Use Baseline:** Select this option to use the project on Polyspace Access as baseline.
- **Baseline: Show Only New Findings:** Select this option to suppress results that are already present in the project in Polyspace Access.

Step 3: Download Baseline

Explicitly download the Polyspace Access result to use as baseline.

- 1 Select **View > Command Palette** and then enter **Polyspace: Get Baseline**.
- 2 Enter the password that you use to log in to Polyspace Access. The baseline download begins.

To follow the progress of download, select **View > Output** and from the dropdown on the upper right, select **Polyspace as You Code**. Wait for the message:

Baseline download completed!

After download, subsequent runs use the baseline. To disable baseline usage, disable the extension setting **Baseline: Use Baseline**.

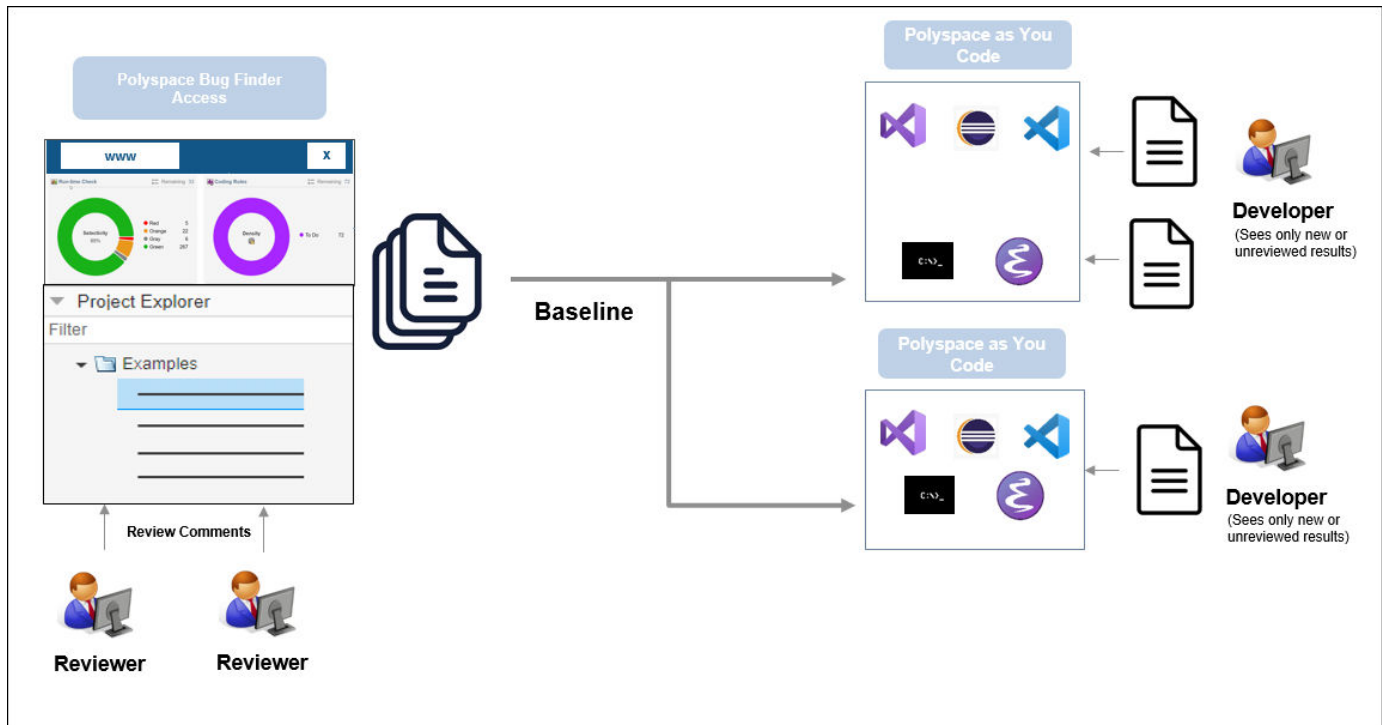
See Also

More About

- “Configure Polyspace as You Code Extension in Visual Studio” on page 5-2
- “Run Polyspace as You Code in Visual Studio Code and Review Results” on page 6-6

Baseline Polyspace as You Code Results in Eclipse

For more efficient bug fixing, you can baseline the results of a Polyspace as You Code analysis with previous results. When you baseline the results, you compare them against the results of a previous run and focus on new results only or on unreviewed results only. You baseline Polyspace as You Code results using previous Polyspace Bug Finder Server results that you download from Polyspace Access.



What Baselined Results Look Like

If you baseline Polyspace as You Code results using Polyspace Bug Finder Server results that you downloaded from Polyspace Access, you can see the following benefits:

- Results that have a justified **Status** on Polyspace Access (**No Action Planned**, **Justified**, or **Not a Defect**) are no longer shown.
- Results that have a non-justified **Status** on Polyspace Access carry over all review information to the **Polyspace Results List** pane in Eclipse. If a result is reviewed in Polyspace Access and marked as such, *one of the following* is true:
 - The **Status** is different from **Unreviewed**.
 - The **Severity** is different from **Unset**.
 - The **Comment** is not blank.
- If you specify the Polyspace as You Code plugin setting **Show only new findings compared to the results baseline**, you see only results that are new in the current run. That way, you can only

focus on results that explicitly occurred because of the changes you made since the last Polyspace Server run.

Baselining Steps

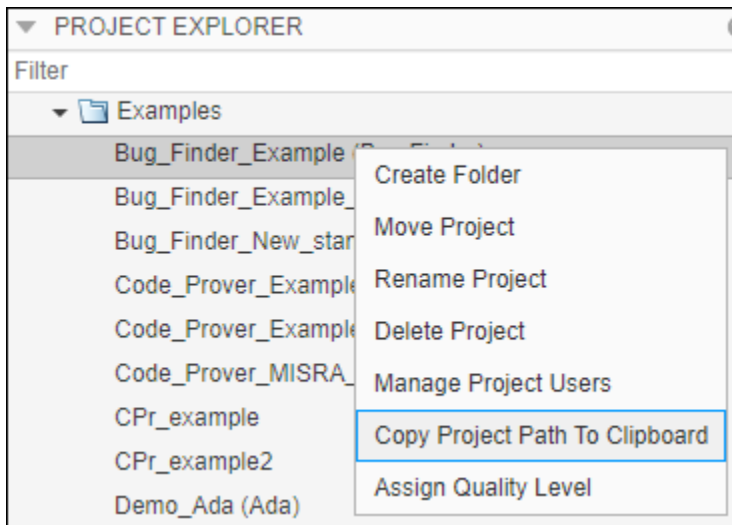
To use Polyspace Bug Finder Server results as baseline for a Polyspace as You Code analysis, follow the steps below. Once a baseline is downloaded, if you choose to point to the baseline, each subsequent run, whether on file save or on-demand, uses the baseline.

Step 1: Identify Project to Use as Baseline

First, identify a project in Polyspace Access that you want to use as baseline. The project must contain results of a Polyspace Bug Finder Server analysis on files that you will analyze in Polyspace as You Code.

Copy the path to the project for use in the Visual Studio Code extension settings. To copy this path:

- 1 Open the Polyspace Access web interface in a web browser.
- 2 On the **Project Explorer** pane, right-click the project and select **Copy Project Path to Clipboard**.



Step 2: Refer to Project from Polyspace as You Code

Next, refer to the Polyspace Access project from the Polyspace as You Code plugin settings in Eclipse.

- 1 Select **Polyspace > Preferences**. Specify the following information:
 - **Polyspace Access URL:** The URL of the server that hosts Polyspace Access. For instance, `https://my-access-url:9443/`.

- **Show only new findings compared to the results baseline:** Select this option to suppress results that are already present in the project in Polyspace Access.

Polyspace Access URL:	<input type="text" value="https://gnb-polyspace-results-server1:9443/"/>
	(Optional) Provide URL to get results baseline from Access server.
	<input checked="" type="checkbox"/> Show only new findings compared to the results baseline

- 2 Select **Polyspace > Configure Project**. Specify the following information:
 - **Use baseline from Polyspace Access:** Select this option to use the project on Polyspace Access as baseline.
 - **Project path:** The path to the project in Polyspace Access that you want to use as baseline. You previously copied this path from the Polyspace Access web interface.

Step 3: Download Baseline

Explicitly download the Polyspace Access result to use as baseline.

- 1 The first time you configure the plugin settings, click **Download baseline from Polyspace Access** to also download the baseline.

To download an updated baseline later, select **Polyspace > Download Results Baseline**. This menu item is available only if you configure the extension settings to use a baseline.

- 2 Enter the username and password that you use to log in to Polyspace Access. The baseline download begins.

To follow the progress of download, close the project settings and select **Window > Show View > Console**. Wait for the popup:

The baseline was successfully downloaded.

After download, subsequent runs use the baseline. To disable baseline usage, disable the plugin setting **Use baseline from Polyspace Access**.

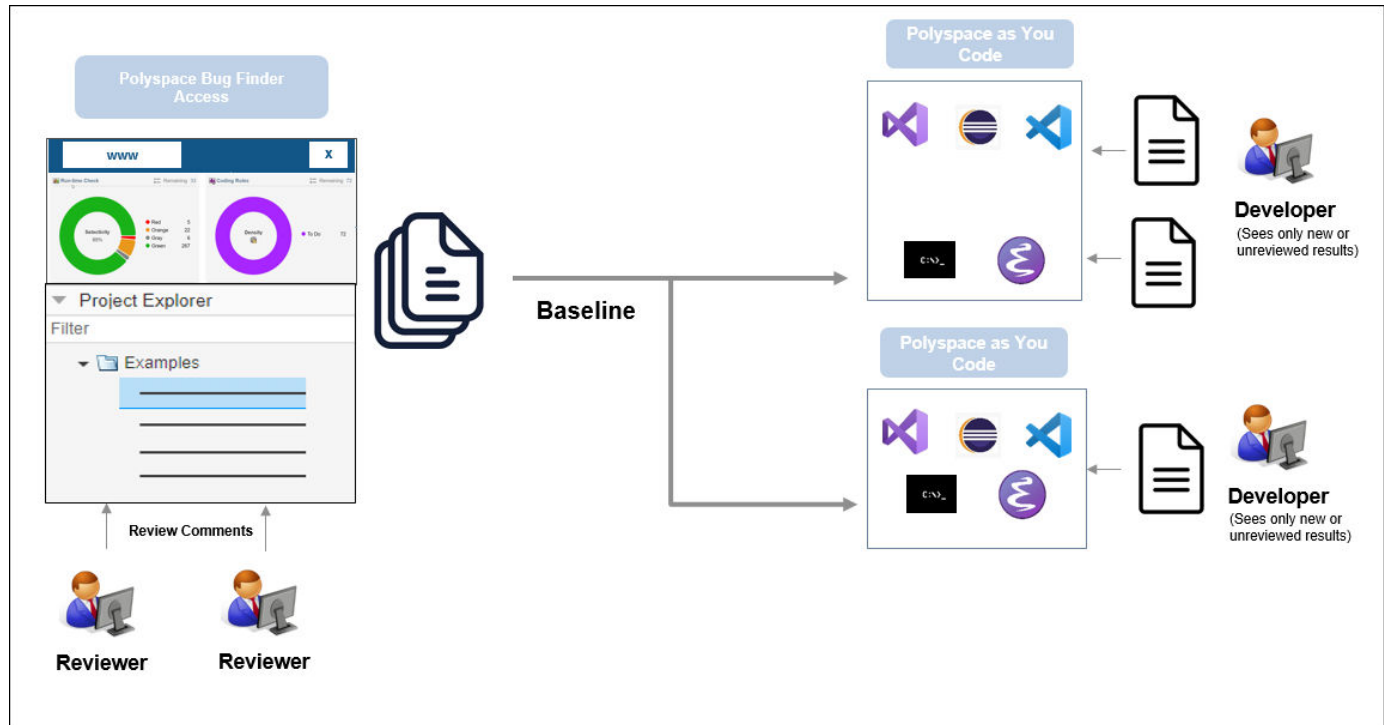
See Also

More About

- “Configure Polyspace as You Code Plugin in Eclipse” on page 5-17
- “Run Polyspace as You Code in Eclipse and Review Results” on page 6-10

Baseline Polyspace as You Code Results on Command Line

For more efficient bug fixing, you can baseline the results of a Polyspace as You Code analysis with previous results. When you baseline the results, you compare them against the results of a previous run and focus on new results only or on unreviewed results only. You baseline Polyspace as You Code results using previous Polyspace Bug Finder Server results that you download from Polyspace Access.



What Baselined Results Look Like

The effect of baselining depends on whether you export results to the console or JSON format (SARIF). For more details on the formats, see `polyspace-results-export`.

Console Output

Results that have a justified **Status** on Polyspace Access (**No Action Planned**, **Justified**, or **Not a Defect**) are no longer shown in the console output.

JSON Output

In the following statements, *obj* represents the JSON object that is exported from the Polyspace results.

- If a result is new and not already present in Polyspace Access, the corresponding property `obj.runs[0].results[n].baselineState` is set to "new":

```
"baselineState" : "new"
```

Otherwise, the property is set to "unchanged".

- Results carry over their review information (**Status**, **Severity** and additional notes) from Polyspace Access to the corresponding properties in `obj.runs[0].results[n].properties`.

For instance, without a baseline, these properties are:

```
"severity" : "Unset",  
"status" : "Unreviewed",  
"comment" : ""
```

With a baseline, the `severity` can be different from "Unset", the `status` different from "Unreviewed", and so on.

- Results that have a justified **Status** on Polyspace Access (**No Action Planned**, **Justified**, or **Not a Defect**) appear with the property `obj.runs[0].results[n].properties.justified` set to `true`:

```
"justified" : true
```

Baselining Steps

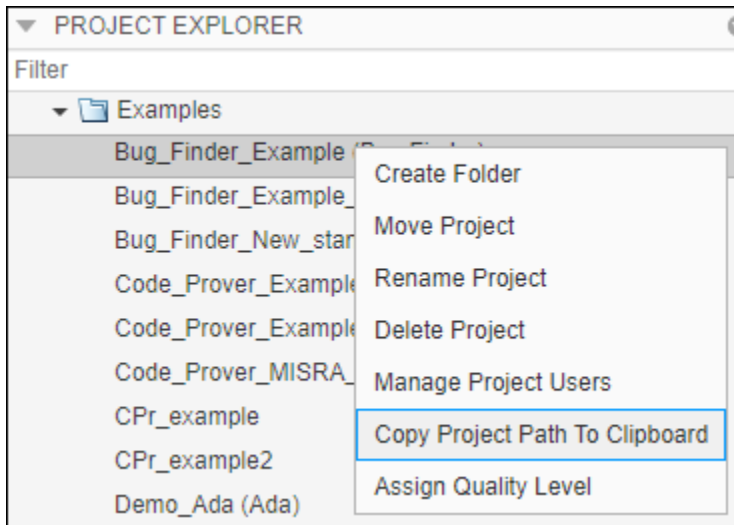
To use Polyspace Bug Finder Server results as baseline for a Polyspace as You Code analysis, follow the steps below.

Step 1: Identify Project to Use as Baseline

First, identify a project in Polyspace Access that you want to use as baseline. The project must contain results of a Polyspace Bug Finder Server analysis on files that you will analyze in Polyspace as You Code.

Copy the path to the project that you want to use as baseline. To copy this path:

- 1 Open the Polyspace Access web interface in a web browser.
- 2 On the **Project Explorer** pane, right-click the project and select **Copy Project Path to Clipboard**.



Step 2: Download Baseline

Next, download the baseline information from the Polyspace Access project. For instance, in a terminal, enter the following:

```
polyspace-access -host hostname -download projectPath -output-folder-path downloadFolder
```

Here:

- *hostname* is the name of the Polyspace Access server.
- *projectPath* is the path to the project on Polyspace Access that is used as baseline. You copied this name from the Polyspace Access web interface.
- *downloadFolder* is the folder to which you download the baseline information.

After download, the folder contains three databases: results (`ps_results.psf`), source files (`ps_sources.db`), and review information (`ps_comments.db`). You cannot open these results in the Polyspace user interface or use them in any other way other than as baseline for Polyspace as You Code runs.

The folder also contains a file `ps_access_info.json` that contains information about the Access project and run ID that was used as baseline. If required, you can write a script to compare this run ID with the latest run ID of the project on Polyspace Access and run this script at certain points in your workflow to make sure that you are always using the latest run of the project as baseline.

Step 3: Use Baseline

Once the baseline information is downloaded, refer to the downloaded baseline information in command-line runs using the option `-baseline-folder`. In a terminal, enter the following:

```
#Linux command
polyspace-bug-finder-access -sources filename -baseline-folder downloadFolder \
```

```
-results-dir resultsFolder
```

```
#DOS command
```

```
polyspace-bug-finder-access.exe -sources filename -baseline-folder downloadFolder ^  
-results-dir resultsFolder
```

Here:

- *filename* is the current file being analyzed.
- *downloadFolder* is the folder to which you previously downloaded the baseline information.
- *resultsFolder* is the folder for storing analysis results.

When you export the analysis results using the `polyspace-results-export` command, for instance:

```
polyspace-results-export -format console -results-dir resultsFolder
```

You can see the effects of using the baseline. See “What Baselined Results Look Like” on page 5-53.

See Also

`polyspace-bug-finder-access` | `polyspace-results-export`

More About

- “Run Polyspace as You Code from Command Line and Export Results” on page 6-14

Configure Checkers for Polyspace as You Code in Eclipse

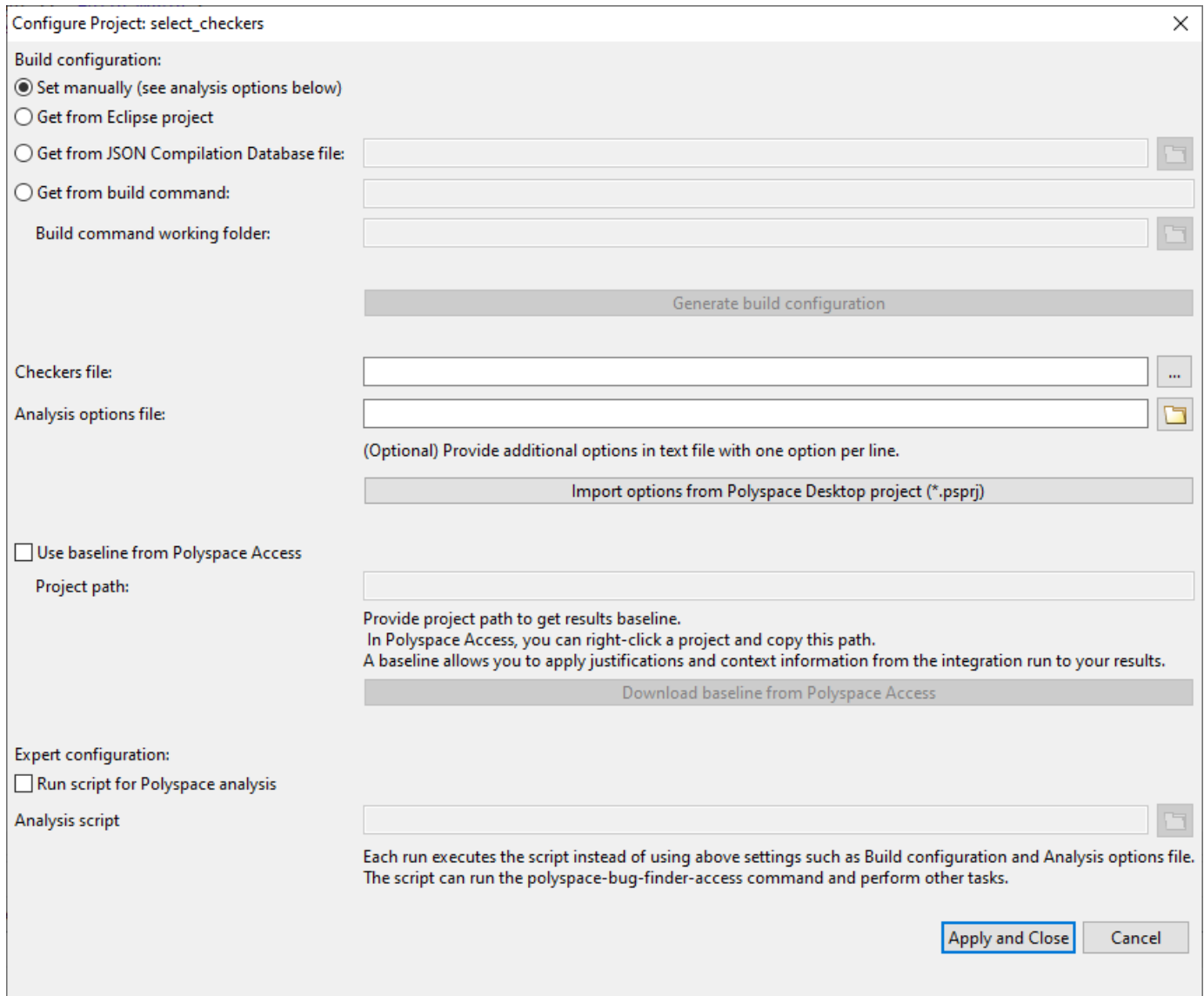
In this section...
“Select Checkers and Coding Rules” on page 5-57
“Modify Checker Behavior” on page 5-60

You can check for various types of defects and coding rule violations by using Polyspace as You Code in Eclipse. See “Defects” and “Coding Standards”. The default analysis checks for a subset of defects. See “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 5-73. To check for nondefault defects and coding rule violations, configure Polyspace as You Code extension in your IDE.

To configure checkers, create a checkers file, and then specify the checkers file in the Configure Project window. For equivalent workflows in the Polyspace desktop and server, see “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” (Polyspace Bug Finder Server).

Select Checkers and Coding Rules

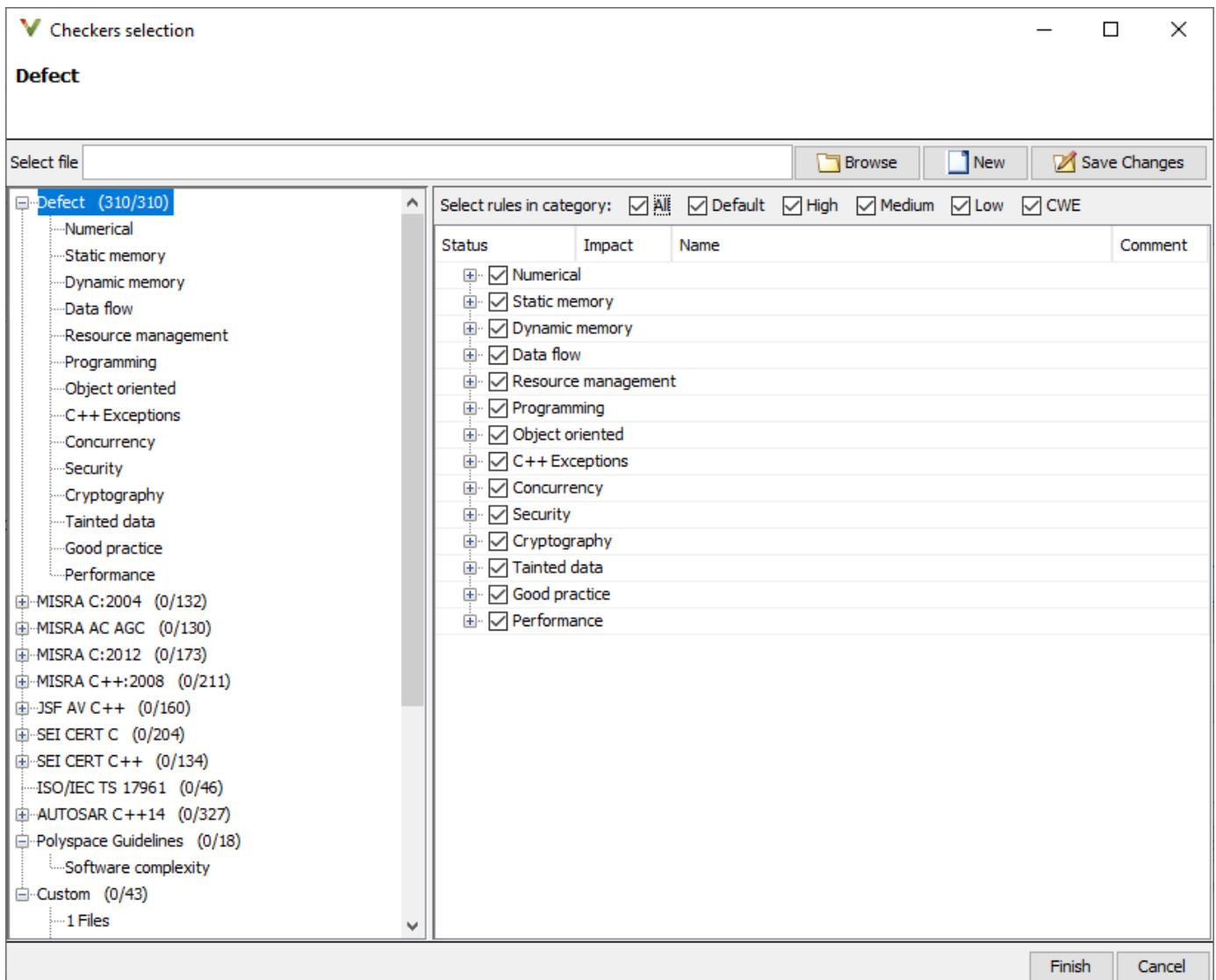
To select coding rule checkers and nondefault defect checkers, click **Polyspace > Configure Project**. Configure the checker selection in the Configure Project window.



Create or Modify Checkers Configuration

Create a new selection or modify an existing selection of checkers and coding rules in the Checker selection window. Save the new configuration in a reusable checkers file.

- 1 In the Configure Project window, open the Checkers Selection window by clicking .



- 2 To create a new selection, in the Checkers Selection window, select the defect and the coding rule checkers that you want to activate. To modify an existing selection, click **Browse**, navigate to the existing checkers file and then modify the checkers selection.

You can also select predefined categories of defect checkers such as **All**, **Default**, **High**, **Medium**, **Low**, and **CWE**. See “Classification of Defects by Impact” on page 3-11. Similarly, you can activate a predefined set of coding rules that are defined by their corresponding standards.

- When selecting **Guidelines** > **Software Complexity** checkers, review their thresholds. If the default thresholds are not acceptable, specify a suitable threshold in the **Threshold** column. See Check Guidelines (-guidelines).
- When selecting **Custom** rules, review the **Pattern** and **Convention** for the rules. See Check custom rules (-custom-rules).

- 3 Save the selection as a reusable checkers XML file by clicking **Save Changes**. After you click **Finish**, the path to the new checkers XML file is specified in the field **Checkers file** in the Configure Project window.

Import Checkers Configuration from Desktop Project

You can import checkers and coding rule configuration from a Polyspace desktop project (*.psprj) file. In the Configure Project window, click **Import options from Polyspace desktop project (*.psprj)**. Browse to the folder containing the project file and specify the project file. The checkers configuration in the desktop project is extracted as a checkers file, which is specified in the field **Checkers file**. The analysis options of the desktop project are extracted as an options file which is specified in the field **Analysis options file**.

Modify Checker Behavior

To modify the default behavior of Bug Finder defect checkers and coding rules, use analysis options. For a list of analysis options that modify the default checker behavior, see “Modify Default Behavior of Bug Finder Checkers” (Polyspace Bug Finder Server).

To specify analysis options in Polyspace as You Code:

- Append the analysis options in the options file specified in the field **Analysis options file**. An options file is a text file with one analysis option for each line. For instance, to add the analysis options `-code-behavior-specifications` and Effective boolean types (`-boolean-types`), in the options file, append these lines:

```
-code-behavior-specifications file1  
-boolean-types boolean1_t,boolean2_t
```
- If you do not have an option file, create an option file that contains the necessary options. Specify the path to the new options file in the field **Analysis options file**. See “Options Files for Polyspace Analysis” on page 5-22.

See Also

More About

- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 5-73
- “Options Files for Polyspace Analysis” on page 5-22
- “Checkers Deactivated in Polyspace as You Code Default Analysis” on page 5-80
- “Modify Default Behavior of Bug Finder Checkers” (Polyspace Bug Finder Server)

Configure Checkers for Polyspace as You Code in Visual Studio

In this section...

“Select Checkers and Coding Rules” on page 5-61

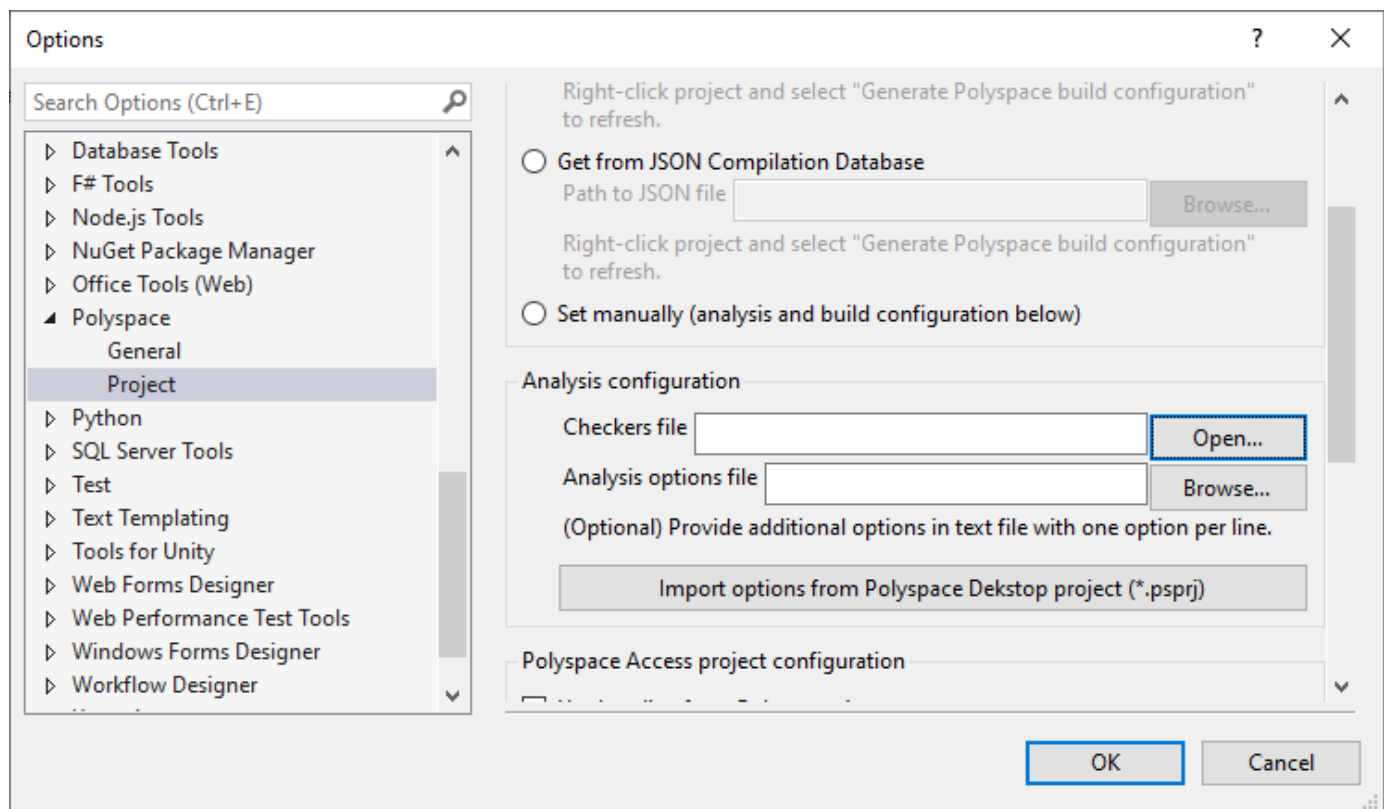
“Modify Checker Behavior” on page 5-63

You can check for various types of defects and coding rule violations by using Polyspace as You Code in Visual Studio. See “Defects” and “Coding Standards”. The default analysis checks for a subset of defects. See “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 5-73. To check for nondefault defects and coding rule violations, configure Polyspace as You Code extension in your IDE.

To configure checkers, create a checkers file, and then specify the checkers file in the Options window. For equivalent workflows in the Polyspace desktop and server, see “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” (Polyspace Bug Finder Server).

Select Checkers and Coding Rules

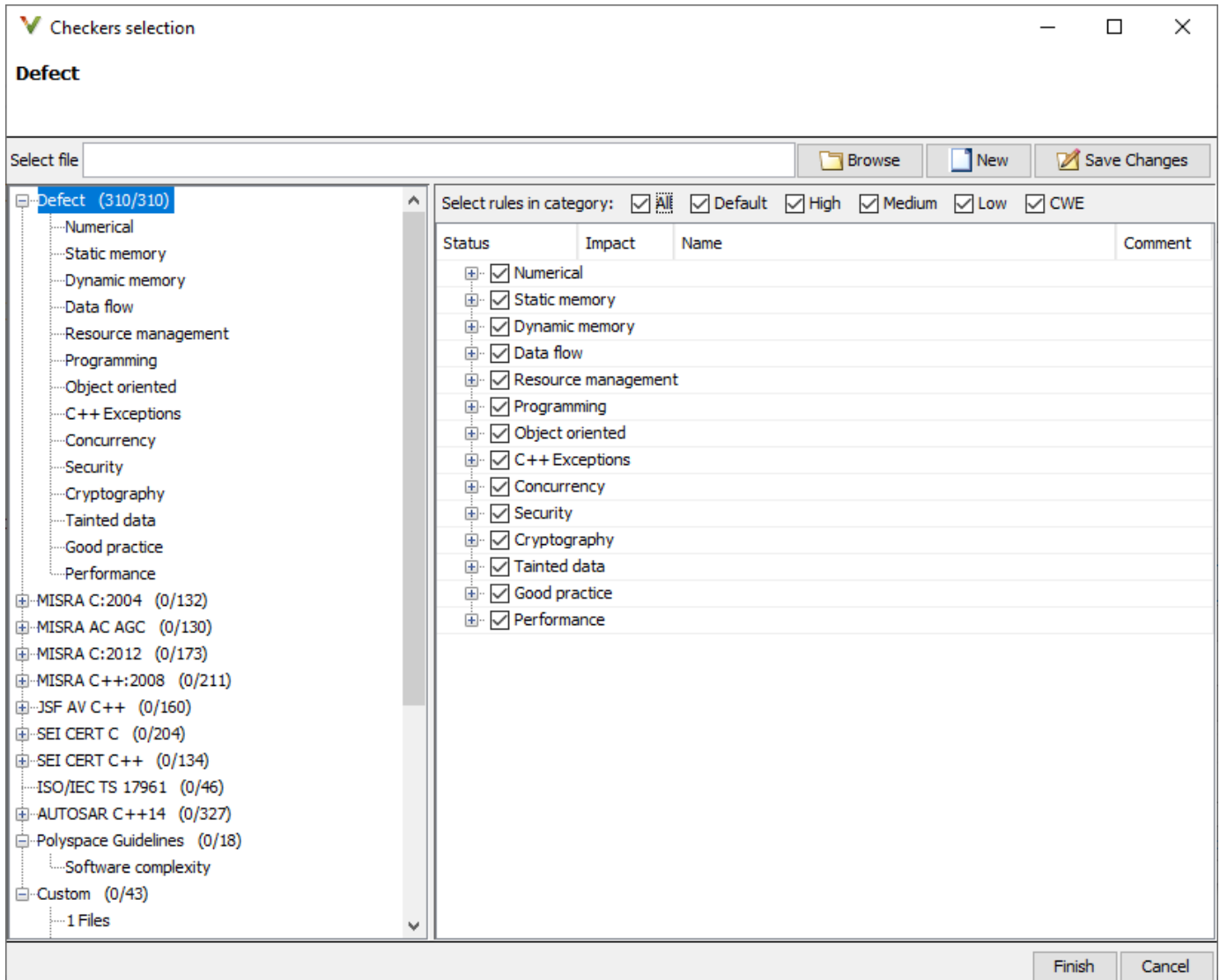
To select coding rule checkers and nondefault defect checkers, click **Tools > Options**. On the **Polyspace > Project** node, specify checkers configuration in the Analysis configuration section.



Create or Modify Checkers Configuration

Create a new selection or modify an existing selection of checkers and coding rules in the Checker selection window. Save the new configuration in a reusable checkers file.

- 1 In the **Polyspace > Project** node, open the Checkers selection window by clicking **Open**.



- 2 To create a new selection, in the Checkers Selection window, select the defect and the coding rule checkers that you want to activate. To modify an existing selection, click **Browse**, navigate to the existing checkers file and then modify the checkers selection.

You can also activate predefined categories of defect checkers such as **All**, **Default**, **High**, **Medium**, **Low**, and **CWE**. See “Classification of Defects by Impact” on page 3-11. Similarly, you can activate predefined set of coding rules that are defined by their corresponding standards.

- When selecting **Guidelines > Software Complexity** checkers, review their thresholds. If the default thresholds are not acceptable, specify a suitable threshold in the **Threshold** column. See Check Guidelines (-guidelines).

- When selecting **Custom** rules, review the **Pattern** and **Convention** for the rules. See Check custom rules (-custom-rules).
- 3 Save the selection as a reusable checkers XML file by clicking **Save Changes**. After you click **Finish**, the path to the new checkers XML file is specified in the field **Checkers file** in the **Polyspace > Project** node.

Import Checkers Configuration from Desktop Project

To import checkers and coding rule selection from a Polyspace desktop project (*.psprj) file, in the **Polyspace > Project** node, click **Import options from Polyspace desktop project (*.psprj)**. Browse to the folder containing the project file and specify the project file. The checkers configuration in the desktop project is extracted as a checkers file, which is specified in the field **Checkers file**. The analysis options of the desktop project are extracted as an options file which is specified in the field **Analysis options file**.

Modify Checker Behavior

To modify the default behavior of Bug Finder defect checkers and coding rules, use analysis options. For a list of analysis options that modify the default checker behavior, see “Modify Default Behavior of Bug Finder Checkers” (Polyspace Bug Finder Server).

To specify analysis options in Polyspace as You Code:

- Append the analysis options in the options file specified in the field **Analysis options file**. An options file is a text file with one analysis option for each line. For instance, to add the analysis options `-code-behavior-specifications` and Effective boolean types (`-boolean-types`), in the options file, append these lines:


```
-code-behavior-specifications file1
-bboolean-types boolean1_t,boolean2_t
```
- If you do not have an option file, create an option file that contains the necessary options. Specify the path to the new options file in the field **Analysis options file**. See “Options Files for Polyspace Analysis” on page 5-22.

See Also

More About

- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 5-73
- “Options Files for Polyspace Analysis” on page 5-22
- “Checkers Deactivated in Polyspace as You Code Default Analysis” on page 5-80
- “Modify Default Behavior of Bug Finder Checkers” (Polyspace Bug Finder Server)

Configure Checkers for Polyspace as You Code in Visual Studio Code

In this section...

“Configure Checkers in Checkers File” on page 5-64

“Modify Checkers Behavior” on page 5-66

You can check for various types of defects and coding rule violations by using Polyspace as You Code in Visual Studio Code. See “Defects” and “Coding Standards”. The default analysis checks for a subset of defects. See “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 5-73. To check for nondefault defects and coding rule violations, configure Polyspace as You Code extension in your IDE.

To configure checkers, create a checkers file, and then specify the checkers file in extension settings. For equivalent workflows in the Polyspace desktop and server, see “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” (Polyspace Bug Finder Server).

Configure Checkers in Checkers File

To configure checkers, first select checkers in a checkers file. Then specify the checkers file in the **Settings** pane.

Step 1: Select Checkers and Coding Rules

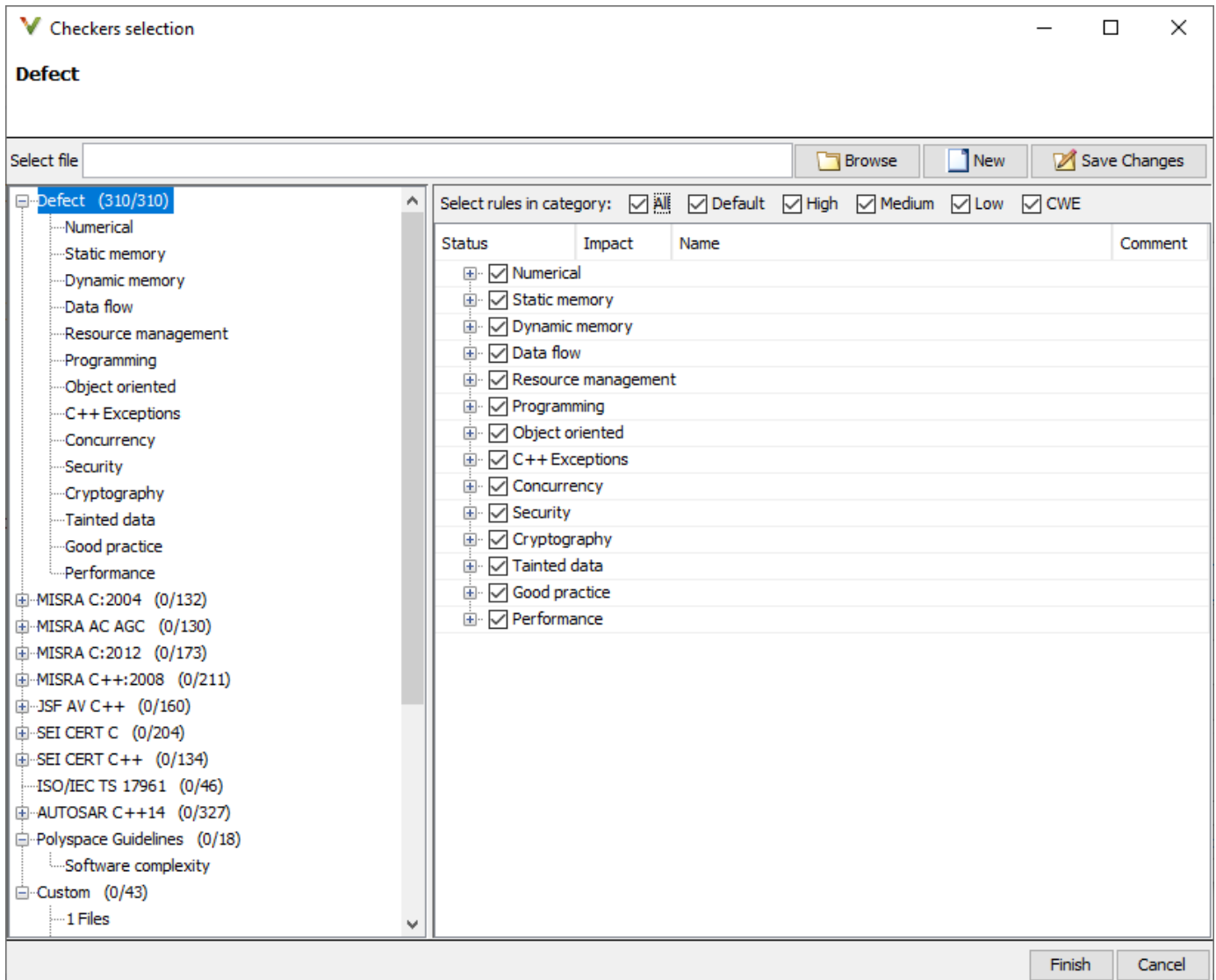
To enable nondefault defects and coding rules, you might:

- Create or modify a checkers file.
- Import a checkers selection from a Polyspace desktop project.

Create or Modify Checkers File

Create a new selection or modify an existing selection of checkers and coding rules in the Checker selection window. Save the new configuration in a reusable checkers file. To use an existing checkers file without modifying the checkers selection, specify a path to it in the **Settings** pane. See “Step 2: Specify Checkers File in Extension Settings” on page 5-66.

- 1 To open the **Checkers selection** user interface, in the command palette, run **Polyspace: Configure Checkers**.



- To create a new selection, in the Checkers Selection window, select the defect and the coding rule checkers that you want to activate. To modify an existing selection, click **Browse**, navigate to the existing checkers file and then modify the checkers selection.

You can also activate predefined categories of defect checkers such as **All**, **Default**, **High**, **Medium**, **Low**, and **CWE**. See “Classification of Defects by Impact” on page 3-11. Similarly, you can activate predefined set of coding rules that are defined by their corresponding standards.

- When selecting **Guidelines > Software Complexity** checkers, review their thresholds. If the default thresholds are not acceptable, specify a suitable threshold in the **Threshold** column. See Check Guidelines (-guidelines).
 - When selecting **Custom** rules, review the **Pattern** and **Convention** for the rules. See Check custom rules (-custom-rules).
- Save the selection as a reusable checkers XML file by clicking **Save Changes**. You can later reuse the checkers XML file as an input in the field **Checkers file**. Click **Finish**.

Import Checkers Configuration from desktop Project

If you have a Polyspace desktop project file (*.psprj), you can import checkers configuration from it. In the Visual Studio Code terminal, run:

#Linux command

```
polyspace-checkers-selection -checkers-selection-output-file PathToOutputFile.json \  
-import-options-from-psprj PathToProject.psprj
```

#DOS command

```
polyspace-checkers-selection.exe -checkers-selection-output-file PathToOutputFile.json ^  
-import-options-from-psprj PathToProject.psprj
```


where *PathToProject.psprj* is the full path to the polyspace desktop project file and *PathToOutputFile.json* is the full path to a JSON file. The JSON file must be in a writable folder. The JSON file contains the location of the produced checkers file in this format:

```
{  
  "checkers-activation-file": "GeneratedCheckersActivationFile",  
  "analysis-options-file": "GeneratedAnalysisOptionFile"  
}
```

The checkers file in *GeneratedCheckersActivationFile* contains the imported checker configurations from the Polyspace desktop project file.

Step 2: Specify Checkers File in Extension Settings

After creating the checkers file, specify the path to it in the **Settings** pane:

- On the Side bar, click the **Extensions** button. The **Extensions** pane opens where your installed extensions are listed.
- Locate Polyspace as You Code in the **Extensions** pane. Click  and select **Extension Settings**.
- In the **Settings** pane, specify the path to the checkers file in the field **Checkers File**.

Modify Checkers Behavior

To modify the default behavior of Bug Finder defect checkers and coding rules, use analysis options. For a list of analysis options that modify the default checker behavior, see “Modify Default Behavior of Bug Finder Checkers” (Polyspace Bug Finder Server).

To specify analysis options in Polyspace as You Code:

- Append the analysis options in the options file specified in the field **Analysis options file**. An options file is a text file with one analysis option for each line. For instance, to add the analysis options `-code-behavior-specifications` and Effective boolean types (`-boolean-types`), in the options file, append these lines:

```
-code-behavior-specifications file1  
-boolean-types boolean1_t,boolean2_t
```
- If you do not have an option file, create an option file that contains the necessary options. Specify the path to the new options file in the field **Other Analysis Options**. See “Options Files for Polyspace Analysis” on page 5-22.

See Also

More About

- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 5-73
- “Options Files for Polyspace Analysis” on page 5-22
- “Checkers Deactivated in Polyspace as You Code Default Analysis” on page 5-80
- “Modify Default Behavior of Bug Finder Checkers” (Polyspace Bug Finder Server)

Configure Checkers for Polyspace as You Code at the Command Line

In this section...

“Configure Checkers and Coding Rules Directly at the Command Line” on page 5-68

“Configure Checkers in Checkers file” on page 5-69

“Modify Checkers Behavior” on page 5-71

If you use an unsupported IDE, you can check for various types of defects and coding rule violations by using Polyspace as You Code at the command line. See “Defects” and “Coding Standards”. The default analysis checks for a subset of defects. See “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 5-73. To check for other defects and coding rule violations, configure Polyspace as You Code.

To configure checkers, create a checkers file and then specify the checkers file at the command line. For equivalent workflows in the Polyspace desktop and server, see “Prepare Checkers Configuration for Polyspace Bug Finder Analysis” (Polyspace Bug Finder Server).

Configure Checkers and Coding Rules Directly at the Command Line

When running Polyspace as You Code in an unsupported IDE, you can specify a selection of checkers and coding rules by using these analysis options with appropriate values directly at the command line:

- Find defects (-checkers)
- Check MISRA C:2004 (-misra2)
- Check MISRA C:2012 (-misra3)
- Check SEI CERT-C (-cert-c)
- Check ISO/IEC TS 17961 (-iso-17961)
- Check MISRA C++:2008 (-misra-cpp)
- Check SEI CERT-C++ (-cert-cpp)
- Check AUTOSAR C++ 14 (-autosar-cpp14)
- Check JSF AV C++ rules (-jsf-coding-rules)
- Check custom rules (-custom-rules)
- Check Guidelines (-guidelines)

For instance, to activate the performance checkers and MISRA C:2012 coding rule, in the command line interface, run

```
polyspace-bug-finder-access -sources <source.c> -checkers performance -misra3 all
```

See the documentation of the analysis options for their command line syntax. To view the results, use `polyspace-results-export`.

Specifying checkers and coding rule selection enables you to select predefined subsets of checkers and coding rules. To select a customized subset of checkers and coding rules, configure checkers by using a checker file.

Configure Checkers in Checkers file

To configure checkers, first select checkers in a checkers file. Then specify the checkers file in the **Settings** pane.

Step 1: Select Checkers and Coding Rule

To enable nondefault defects and coding rules, you might:

- Create or modify a checkers file.
- Import a checkers selection from a Polyspace desktop project.

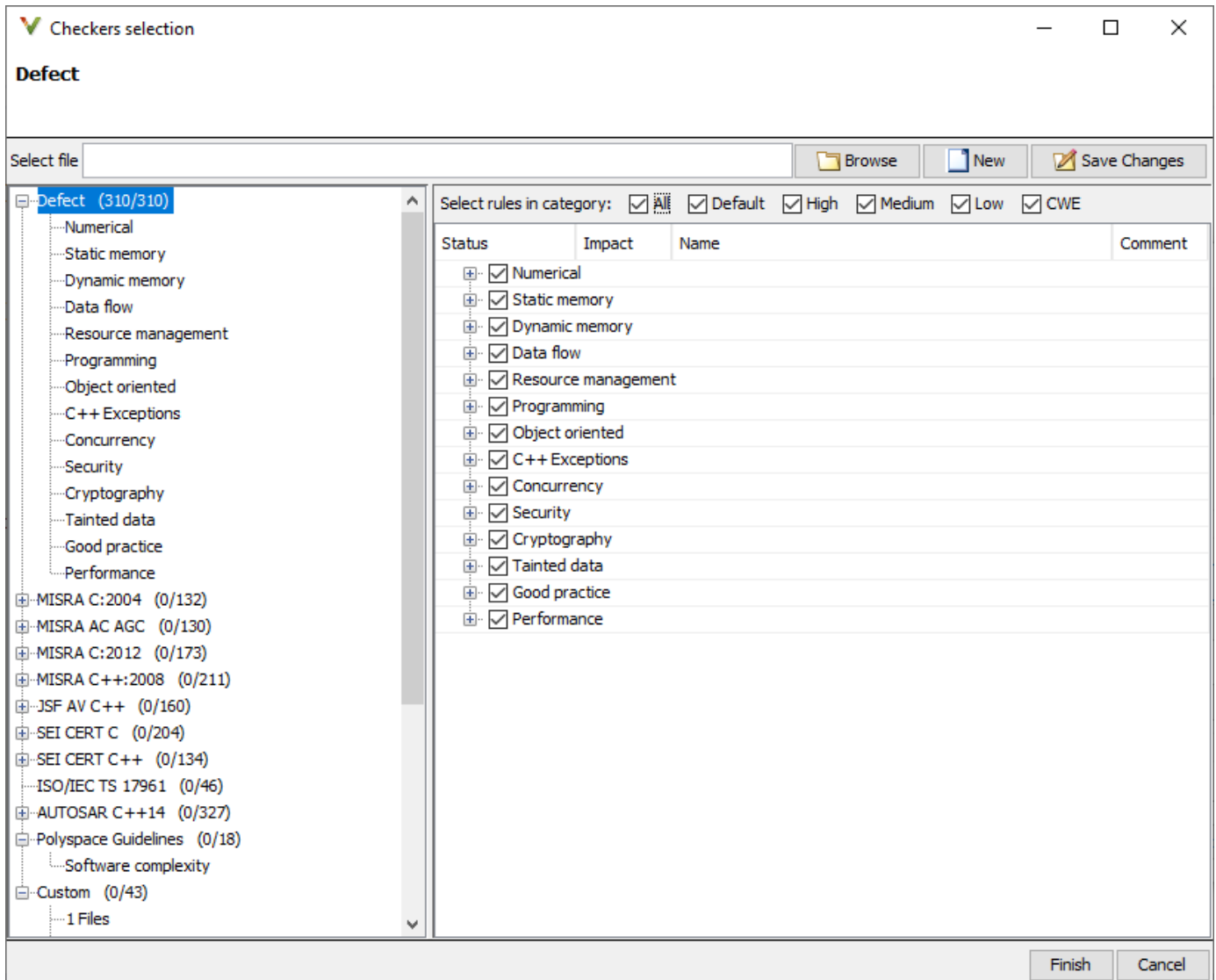
Create or Modify Checkers Configuration

Create a new selection or modify an existing selection of checkers and coding rules in the Checker selection window. Save the new configuration in a reusable checkers file. To use an existing checkers file without modifying the checkers selection, specify it at the command line. See “Step 2: Specify Checker File at the Command Line” on page 5-71.

- 1 To open the Checkers selection user interface, in the command line, run:

```
polyspace-checkers-selection
```

The Checkers Selection interface opens.



- 2 To create a new selection, in the Checkers Selection window, select the defect and the coding rule checkers that you want to activate. To modify an existing selection, click **Browse**, navigate to the existing checkers file and then make your selection.

You can also activate predefined categories of defect checkers such as **All**, **Default**, **High**, **Medium**, **Low**, and **CWE**. See “Classification of Defects by Impact” on page 3-11. Similarly, you can activate predefined set of coding rules that are defined by their corresponding standards.

- When selecting **Guidelines** > **Software Complexity** checkers, review their thresholds. If the default thresholds are not acceptable, specify a suitable threshold in the **Threshold** column. See Check Guidelines (-guidelines).
 - When selecting **Custom** rules, review the **Pattern** and **Convention** for the rules. See Check custom rules (-custom-rules).
- 3 Save the selection as a reusable checkers XML file as *CreatedCheckerFile.xml* and then **Finish**. You can later reuse *CreatedCheckerFile.xml* as a value to `-checkers-activation-file`.

Import Checkers Configuration from Desktop Project

If you have a Polyspace desktop Project (*.psprj) file, you can import checker selection from it. At the command line, run

```
#Linux command
polyspace-checkers-selection -checkers-selection-output-file PathToOutputFile.json \
-import-options-from-psprj PathToProject.psprj
```

```
#DOS command
polyspace-checkers-selection -checkers-selection-output-file PathToOutputFile.json \
-import-options-from-psprj PathToProject.psprj
```

where *PathToProject.psprj* is the full path to the polyspace desktop project file and *PathToOutputFile.json* is the full path to a JSON file. The JSON file must be in a writable folder. The JSON file contains the location of the produced checkers file in this format:

```
{
  "checkers-activation-file": "PathToCreatedCheckerFile",
  "analysis-options-file": "CreatedOptionsFile"
}
```

The checkers file in *PathToCreatedCheckerFile* contains the checker configurations in the Polyspace desktop project file.

Step 2: Specify Checker File at the Command Line

After you obtain the checkers file, specify its full path as an argument to `-checkers-activation-file`. For instance:

```
#Linux command
polyspace-bug-finder-access -sources <source.c> \
-checkers-activation-file PathToCreatedCheckerFile
```

```
#DOS command
```

```
polyspace-bug-finder-access -sources <source.c> ^
-checkers-activation-file PathToCreatedCheckerFile
```

where *PathToCreatedCheckerFile* is the full path to the checkers file.

Modify Checkers Behavior

To modify the default behavior of Bug Finder defect checkers and coding rules, use analysis options. For a list of analysis options that modify the default checker behavior, see “Modify Default Behavior of Bug Finder Checkers” (Polyspace Bug Finder Server).

To specify analysis options in Polyspace as You Code:

- Use the options in the command line. For instance, to modify the trust boundary of your analysis, in the command line, run:

```
polyspace-bug-finder-access -sources <source.c> -checkers-activation-file CreatedCheckerFile -
```

You can specify multiple behavior modifying options in the command line.

- Append the analysis options in the options file specified in the field **Analysis options file**. An options file is a text file with one analysis option for each line. For instance, to add the analysis options `-code-behavior-specifications` and Effective boolean types (`-boolean-types`), in the options file, append these lines:

```
-code-behavior-specifications file1  
-boolean-types boolean1_t,boolean2_t
```

- If you do not have an existing options file, create an options file containing the necessary options. See “Options Files for Polyspace Analysis” on page 5-22.

See Also

More About

- “Polyspace Bug Finder Defects Checkers Enabled by Default” on page 5-73
- “Options Files for Polyspace Analysis” on page 5-22
- “Checkers Deactivated in Polyspace as You Code Default Analysis” on page 5-80
- “Modify Default Behavior of Bug Finder Checkers” (Polyspace Bug Finder Server)
- “Run Polyspace as You Code from Command Line and Export Results” on page 6-14

Polyspace Bug Finder Defects Checkers Enabled by Default

When you start a Bug Finder analysis, these checkers are enabled by default:

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
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 on adjacent bit fields	DATA_RACE_BIT_FIELDS
Data race through standard library function call	DATA_RACE_STD_LIB
Dead code	DEAD_CODE
Deadlock	DEADLOCK
Deallocation of previously deallocated pointer	DOUBLE_DEALLOCATION

Defect	Command-line Name
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
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
Incorrect value forwarding	INCORRECT_VALUE_FORWARDING
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 deletion of pointer	BAD_DELETE
Invalid free of pointer	BAD_FREE
Invalid use of = (assignment) operator	BAD_EQUAL_USE
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

Defect	Command-line Name
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
Lambda used as typeid operand	LAMBDA_TYPE_MISUSE
Memory comparison of padding data	MEMCMP_PADDING_DATA
Memory comparison of strings	MEMCMP_STRINGS
Missing lock	BAD_UNLOCK
Missing null in string array	MISSING_NULL_CHAR
Missing return statement	MISSING_RETURN
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
Move operation on const object	MOVE_CONST_OBJECT
Noexcept function exits with exception	NOEXCEPT_FUNCTION_THROWS
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
Operator new not overloaded for possibly overaligned class	MISSING_OVERLOAD_NEW_FOR_ALIGNED_OBJ
Partial override of overloaded virtual functions	PARTIAL_OVERRIDE
Partially accessed array	PARTIALLY_ACCESSED_ARRAY
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

Defect	Command-line Name
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
Self assignment not tested in operator	MISSING_SELF_ASSIGN_TEST
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
Throw argument raises unexpected exception	THROW_ARGUMENT_EXPRESSION_THROWS
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
Unnamed namespace in header file	UNNAMED_NAMESPACE_IN_HEADER
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

Defect	Command-line Name
Variable length array with nonpositive size	NON_POSITIVE_VLA_SIZE
Variable shadowing	VAR_SHADOWING
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 type used in sizeof	PTR_SIZEOF_MISMATCH

Analysis Scope of Polyspace as You Code

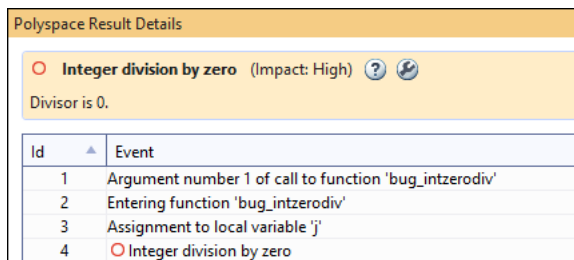
Polyspace as You Code is a static code analysis software meant for regular use by C/C++ developers within their Integrated Development Environments (IDEs). Polyspace as You Code can find bugs and coding standard violations on the file that is currently active in the IDE.

This topic outlines the analysis scope of Polyspace as You Code and the benefits of using Polyspace Bug Finder or Polyspace Bug Finder Server for full integration analysis.

Results Involve Current File Only

Polyspace as You Code is designed to provide results that are of immediate interest to developers. So the tool only shows results in files that you are currently working on. After installing the Polyspace as You Code extension, each time you open or save a file in your IDE, the analysis runs silently in the background and highlights issues in the file.

All issues found originate within the source file itself and can also be fixed within this file. You can either implement the fix at the highlighted location or another related location still within the current file. For instance, the following integer division by zero result is shown with related events on previous lines. You can implement a guard against division by zero just before the division or implement some checks on inputs to the function where the division is performed.



The screenshot shows a 'Polyspace Result Details' window. At the top, there is a red circle icon followed by the text 'Integer division by zero (Impact: High)'. Below this, it says 'Divisor is 0.'. A table below lists four events related to the error:

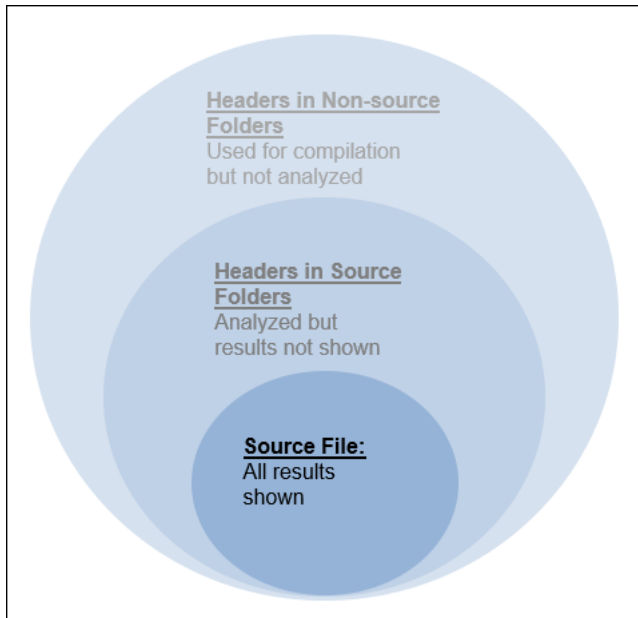
Id	Event
1	Argument number 1 of call to function 'bug_intzerodiv'
2	Entering function 'bug_intzerodiv'
3	Assignment to local variable 'j'
4	Integer division by zero

Results that involve multiple files, for instance, declaration mismatch across files or data flow between functions in different files, are not shown in the default Polyspace as You Code analysis. To see complete integration results on your project, analyze your project with Polyspace Bug Finder on your desktop or with Polyspace Bug Finder Server on a continuous integration (CI) server.

Some checkers that are not likely to find issues in a single-file analysis are completely disabled in Polyspace as You Code. See “Checkers Deactivated in Polyspace as You Code Default Analysis” on page 5-80.

Headers Included in Current File Not Analyzed

If a source file `#include-s` a header that is in the same folder as the source (or in subfolders), the analysis considers this header but does not show any result inside the header. All other header files are taken into account for compilation but not analyzed further.



The reason for this default behavior is the following:

- *Headers close to sources:*

The underlying assumption is that headers in the source folders are more closely related to the current source file and are therefore relevant for the analysis. However, a developer might not own the headers and might not want to fix issues such as coding standard violations in the headers. Therefore, results in these headers are not shown.

- *Headers in non-source folders:*

Headers in other folders typically come from third party libraries and are not analyzed.

You can change this default behavior using these options:

- **Generate results for sources and (-generate-results-for):** Use this option to expand the scope of which headers must be analyzed.
- **Do not generate results for (-do-not-generate-results-for):** Use this option to expand the scope of which headers must not be analyzed.

Even using these options, you cannot see results in headers `#include-d` through source files. If you want results in a header file, analyze the header file directly and not through a source file.

Checkers Deactivated in Polyspace as You Code Default Analysis

Checkers and Coding Rule Deactivated in Polyspace as You Code

By default, Polyspace as You Code runs an analysis on the currently active file in your IDE. If finding an issue typically requires multiple source files, the default Polyspace as You Code analysis might not flag it. Checkers corresponding to these issues are deactivated in a default Polyspace as you Code analysis. These issues can be detected by running an integration analysis on your project by using Polyspace Bug Finder or Polyspace Bug Finder Server.

Deactivated Bug Finder Checkers

The Bug Finder checkers that are deactivated in the default Polyspace as You Code analysis include:

- Declaration mismatch
- Qualifier removed in conversion
- Typedef mismatch
- “Concurrency Defects”

Deactivated CERT C Rules

The CERT C coding rules that are deactivated in the default Polyspace as You Code analysis include:

- CERT C: Rule DCL40-C
- CERT C: Rec. DCL15-C

Deactivated Cert C++ Rules

The CERT C++ coding rules that are deactivated in the default Polyspace as You Code analysis include:

- CERT C++: DCL40-C

Deactivated MISRA C:2004 and MISRA AC AGC Rules

The MISRA C:2004 and MISRA AC AGC coding rules that are deactivated in the default Polyspace as You Code analysis include:

- MISRA C:2004 and MISRA AC AGC Rules 5.1, 5.4, 5.6, 8.4, 8.8, 8.9, 8.10. See “Supported MISRA C:2004 and MISRA AC AGC Rules” on page 7-3

Deactivated MISRA C:2012 Rules

The MISRA C:2012 coding rules that are deactivated in the default Polyspace as You Code analysis include:

- MISRA C:2012 Rule 2.3
- MISRA C:2012 Rule 2.4
- MISRA C:2012 Rule 2.5
- MISRA C:2012 Rule 5.1

- MISRA C:2012 Rule 5.6
- MISRA C:2012 Rule 5.8
- MISRA C:2012 Rule 5.9
- MISRA C:2012 Rule 8.3
- MISRA C:2012 Rule 8.5
- MISRA C:2012 Rule 8.6
- MISRA C:2012 Rule 8.7

Deactivated ISO/IEC TS 17961 Rules

The ISO/IEC TS 17961 coding rules that are deactivated in the default Polyspace as You Code analysis include:

- ISO/IEC TS 17961 [funcdecl]

Deactivated MISRA C++:2008 Rules

The MISRA C++:2008 coding rules that are deactivated in the default Polyspace as You Code analysis include:

- MISRA C++:2008 Rule 0-1-3
- MISRA C++:2008 Rule 2-10-5
- MISRA C++:2008 Rule 3-2-1
- MISRA C++:2008 Rule 3-2-2
- MISRA C++:2008 Rule 3-2-3
- MISRA C++:2008 Rule 3-2-4
- MISRA C++:2008 Rule 15-4-1

Deactivated AUTOSAR C++14 Rules

The AUTOSAR C++14 coding rules that are deactivated in the default Polyspace as You Code analysis include:

- AUTOSAR C++14 Rule M0-1-3
- AUTOSAR C++14 Rule M3-2-1
- AUTOSAR C++14 Rule M3-2-2
- AUTOSAR C++14 Rule M3-2-3
- AUTOSAR C++14 Rule M3-2-4

Deactivated JSF C++ Coding Rules

The JSF C++ coding rules that are deactivated in the default Polyspace as You Code analysis include:

- JSF C++ Rule 46,137,139. See “Supported JSF C++ Coding Rules” on page 7-48.

Checkers with Reduced Scope in Polyspace as You Code

The checkers that finds fewer issues in the default Polyspace as You Code analysis are listed in the table. The issues that are not found are related to multiple-file analysis.

Checker	Behavior in the default Polyspace as You Code Analysis
CERT C: Rule EXP37-C	Does not check for “Function declaration mismatch”.
CERT C++: EXP37-C	Does not check for “Function declaration mismatch”.
CERT C++: DCL60-CPP	Does not check for “Nonidentical Definitions of Function or Object Across Modules”.
CERT C: Rec. DCL19-C	Does not check for “Function or object declared without static specifier and referenced in only one file”.
CERT C: Rec. DCL23-C	Does not check for “External identifiers not distinct”.
AUTOSAR C++14 Rule A0-1-3	Does not check for “Private Member Function Not Used”.
ISO/IEC TS 17961 [argcomp]	Does not check for “Conflicting declarations or conflicting declaration and definition”.

Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code

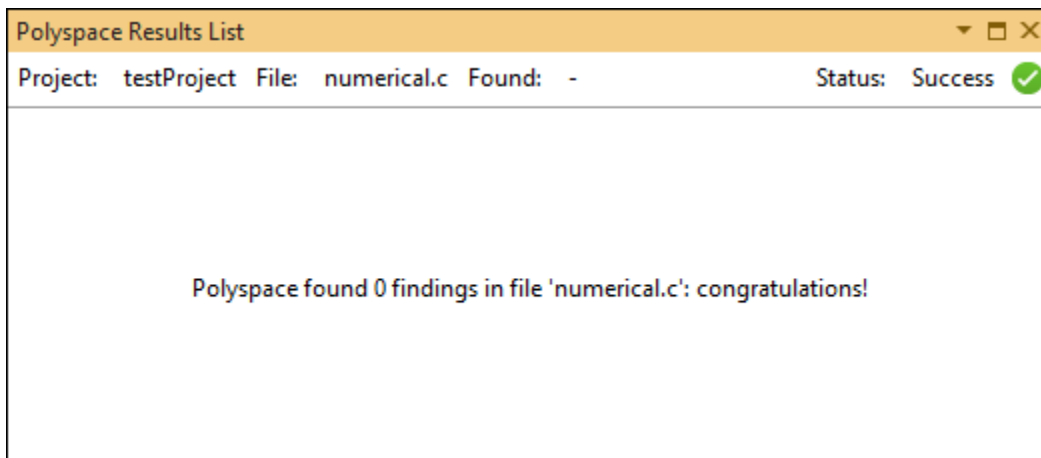
Issue

After installing and configuring Polyspace as You Code in your IDE, you should see analysis results as source code markers within a few seconds of starting the analysis (slightly longer for C++ files). If you do not see results, it could mean that the analysis did not find any issue or the analysis failed to complete or even failed to start.

If you run Polyspace as You Code on each save, some of the runs might fail because a file does not compile yet. If you do not see results despite successful compilation, you might have to investigate further and change the analysis options or extension settings. (Note that you can enable the checker `File does not compile` so that you always see at least one result, even when the file does not compile.)

Possible Solutions

If you do not see results, first confirm that the analysis reached completion. If the analysis completed but did not find any issue, on the IDE pane that shows the full list of results, you see a status message indicating completion. For instance, in Visual Studio, on the **Polyspace Results List** pane, you see this message:



If the analysis failed to complete, you also see a status message indicating failure. For further diagnosis, check the analysis log within the IDE. For instance, in Visual Studio, open the **Output** pane, select **Polyspace** from the **Show output from** drop-down list, and check the messages. You might have to scroll up a bit to see the root cause of the failure.

For more information on how to follow analysis progress in specific IDEs, see:

- “Run Polyspace as You Code in Visual Studio and Review Results” on page 6-2
- “Run Polyspace as You Code in Visual Studio Code and Review Results” on page 6-6
- “Run Polyspace as You Code in Eclipse and Review Results” on page 6-10

Check if Build Analysis is Outdated

The most common cause of an analysis failure is a compilation error. If a file compiles with your compiler but fails with Polyspace, it means that the analysis requires more information to emulate your compiler. In the most common scenario, the error indicates that you have to reanalyze your build.

If you specify in your extension settings that the analysis must use options extracted from a build command, a build task or a JSON compilation database, you must analyze your build command first and then run Polyspace as You Code. The build analysis must run on a command or task that performs a full build of your project or workspace and not an incremental build.

If you add a new file to your project or workspace but forget to rerun build analysis, you might see compilation errors when trying to analyze the new file. The most common error is that include files cannot be found. To fix the issue, simply rerun your build analysis and then run Polyspace as You Code. For details, see “Analyzing Build in Polyspace as You Code”.

Check for Mistakes in Options File

If you specify an options file in your extension settings, the analysis appends options from this file to the underlying run command. If an option is incorrectly written, for instance, refers to a nonexistent file or uses an incorrect argument, the analysis can fail.

You can see all errors and warnings related to options in the analysis log. To see a more detailed log, use the analysis option `-no-quiet`. You can enter this option in the same options file that contains your other options. See “Options Files for Polyspace Analysis” on page 5-22.

Check for Incorrect Path to Analysis Engine

In Visual Studio Code, you can change the extension setting that points to the Polyspace installation folder. If you enter an incorrect path in this setting, the Polyspace as You Code extension fails to start. You see a message indicating that the internal server, Polyspace Connector, attempted to start and then failed.

Check that the installation folder that you provided in your extension settings indeed contains a Polyspace installation. The path must contain a subfolder `polyspace\bin`, that contains the `polyspace-bug-finder-access` executable.

See Also

Related Examples

- “Run Polyspace as You Code in Visual Studio and Review Results” on page 6-2
- “Run Polyspace as You Code in Visual Studio Code and Review Results” on page 6-6
- “Run Polyspace as You Code in Eclipse and Review Results” on page 6-10
- “Run Polyspace as You Code from Command Line and Export Results” on page 6-14

Reduce Software Complexity by Using Polyspace Checkers

Software complexity refers to various quantifiable metrics of a software module or source files, such as number of lines, number of paths, number of functions, or the complexity of the function call tree. The Polyspace software complexity checkers are raised when these metrics exceeds a threshold. High software complexity might indicate that your code is difficult to read, understand, and debug. It is more efficient to maintain the acceptable level of software complexity during development instead of refactoring complex projects later on. Use the software complexity checkers to detect complex modules early in the development cycle to reduce later refactoring efforts.

You can also calculate the absolute values of code complexity metrics for all files and functions. See “Compute Code Complexity Metrics” (Polyspace Bug Finder Server).

Configure Thresholds for Software Complexity Checkers

Each software complexity checker corresponds to a complexity metric. Polyspace raises a software complexity checker when the corresponding code complexity metric exceeds a threshold.

The default thresholds of these checkers follow the Hersteller Initiative Software (HIS) Code Complexity standard. See “HIS Code Complexity Metrics” (Polyspace Bug Finder Server). For checkers that are not present in the HIS standard, the default thresholds are high enough that the code complexity metrics of your code might always be below the threshold. To use these checkers effectively, specify an appropriate threshold for them.

Determine an appropriate set of thresholds for these checkers depending on the best practice for your use case. For instance, when analyzing new projects or newly developed code, you might want to reduce the use of GOTO statements by setting the threshold of `Number of goto statements exceeds threshold` to zero. When analyzing modules containing legacy libraries, you might want to set the threshold to a higher number.

Depending on your Polyspace product, use the user interface or the command-line interface to specify the threshold. For instance:

- In Polyspace desktop or Server products, in the Checkers selection window, navigate to **Guidelines > Software Complexity** and specify the threshold. In the command line, use the analysis option `Check Guidelines (-guidelines)`. See “Check for Coding Standard Violations” (Polyspace Bug Finder Server).
- In Polyspace as You Code extension, start the Checkers selection window and specify the thresholds in the **Guidelines > Software Complexity** node.
 - In Eclipse, open the Checkers selection window from the Configure Project window. See “Configure Checkers for Polyspace as You Code in Eclipse” on page 5-57.
 - In Visual Studio, open the Checkers selection window from the **Polyspace > Project** node of the Options window. See “Configure Checkers for Polyspace as You Code in Visual Studio” on page 5-61.
 - In Visual Studio Code, open the Checkers selection window from the command palette. See “Configure Checkers for Polyspace as You Code in Visual Studio Code” on page 5-64.
 - At the command line, open the Checkers selection window by running the command `polyspace-checkers-selection`. See “Configure Checkers for Polyspace as You Code at the Command Line” on page 5-68.

Identify and Reduce Software Complexity

Identify Software Complexity by Running Bug Finder Analysis

To identify software complexity, configure the thresholds of the checkers. For instance, set the thresholds of the checkers listed in this table.

Checker	Threshold
Comment density below threshold	20
Call tree complexity exceeds threshold	10
Number of call occurrences exceeds threshold	10
Language scope exceeds threshold	400

The thresholds indicate the acceptable level of software complexity. To identify issues in your code that might lead to a higher level of complexity, after configuring the software complexity checkers, run a Polyspace Bug Finder analysis. Consider this code:

```
long long power(double x, int n){
    long long BN = 1;
    for(int i = 0; i<n;++i){
        BN*=x;
    }
    return BN;
}

double AppxIndex(double m, double f){//Noncompliant
    double U = (power(m,2) - 1)/(power(m,2)+2);
    double V = (power(m,4) + 27*power(m,2)+38)/(2*power(m,2)+3);
    return (1+2*f*power(U,2)*(1+power(m,2)*U*V+ power(m,3)
        /power(m,3)*(U-V)))/( (1-2*f*power(U,2)*(1+power(m,2)*U*V
        + power(m,3)/power(m,3)*(U-V))));
}
```

The function `AppxIndex` appears complex. It is not obvious how you might reduce the complexity. The software complexity checkers help you identify the sources of complexity.

After the Bug Finder analysis, the configured checkers are raised:

- **Comment density below threshold:** The functions in the code contain no explanatory comments.
- **Call tree complexity exceeds threshold and Number of call occurrences exceeds threshold:** There are too many function calls compared to the number of function definitions. These checks indicate that you can package some of the expressions into separate functions.
- **Language scope exceeds threshold:** The same operand is repeated several times. You can reduce some of the repetition. For instance, the function `power` is called with the same arguments several times.

These checks indicate that the function `AppxIndex` might make the code difficult to read, understand, and debug. To reduce the complexity of the code, address the raised checks.

Reduce Software Complexity

Reduce the complexity of your code by addressing the identified issues. In this case, the root cause of the raised checks is that the function `AppxIndex` performs several tasks instead of performing one single task. For instance, the function first calculates `U`, then it calculates `V`, and finally it evaluates a lengthy expression containing both `U` and `V`. To address these issues, refactor the function `AppxIndex` so that each task is delegated to a separate function. You might break down the lengthy expression into smaller parts. For instance:

```
// This code calculates effective index of materials as described in
// the formula in 10.1364...
// power(x,n) returns the nth power of x (x^n)
// n is an integer
// x is a double
// return type is long long

long long power(double x, int n){//Compliant
    long long BN = 1;
    for(int i = 0; i<n;++i){
        BN*=x;
    }
    return BN;
}
// CalculateU(m) calculates the first intermediate variable
// required to calculate polarization
// m is the relative refractive index
// return type is double;

double CalculateU(double m){//Compliant
    return (power(m,2) - 1)/(power(m,2)+2);
}
// CalculateV(m) calculates the second intermediate variable
// required to calculate polarization
// m is the relative refractive index
// return type is double;

double CalculateV(double m){//Compliant
    return (power(m,4) + 27*power(m,2)+38)/(2*power(m,2)+3);
}
// CalculateMid(m,f) calculates the large term present
// in both numerator and denominator
// of the effective index calculation
// m is the relative refractive index
// f is the fillfactor
// return type is double;

double CalculateMid(double m, double f){//Compliant
    double U = CalculateU(m);
    double V = CalculateU(m);
    return 2*f*power(U,2)*(1+power(m,2)*U*V + power(m,3)/power(m,3)*(U-V));
}
//AppxIndex(m,f) calculates the approximate effective index
// m is the relative refractive index
// f is the fillfactor
//return type is double
double AppxIndex(double m, double f){//Compliant
```

```
    return (1+CalculateMid(m,f))/( (1-CalculateMid(m,f)));  
}
```

In this code, none of the software complexity checkers is raised, which indicates that you reduced the complexity of this code to an acceptable level. To reduce the software complexity:

- 1** Document the code with sufficient comments.
- 2** Break down the The large complex task performed by `AppxIndex` into smaller and simpler tasks, which are then delegated to individual functions such as `CalculateU`, `CalculateV` and `CalculateMid`. The function `power` is now called less frequently. If you later implement a different function to calculate a power and want to use the new function instead of the current one, you have to make fewer replacements.
- 3** Write the new functions to perform one specific task with as little overlap of their functionalities as possible. As a result, these functions contain less repetition of the same operands.

For details about addressing a software complexity check, see the documentation of the checker.

In cases when you are unable to refactor the code, address the checks through code annotations. For instance, if you are using a complex library, you might choose to annotate the checks that are raised on the library. See “Hide Known or Acceptable Polyspace Results” on page 2-5. When you annotate a file or function code metric, the corresponding software complexity checker is also annotated by the same comment.

See Also

More About

- “Guidelines”

Review Results in Polyspace as You Code

Run Polyspace as You Code in Visual Studio and Review Results

You can choose to run Polyspace as You Code on each save in the Visual Studio IDE, or at will. The analysis runs on the file that is currently active in the IDE (the file must be part of a Visual Studio project, which can be part of a larger Visual Studio solution). After analysis, you see bugs and coding standard violations as source code markers or in a separate list.

Confirm Installation of Extension

To confirm that your Visual Studio installation has the Polyspace as You Code extension, check the list of extensions installed.

- In Visual Studio 2019, select **Extensions > Manage Extensions**.
- In Visual Studio 2017, select **Tools > Extensions and Updates**.

You can also confirm that the extension starts as expected on the **Output** pane. Select **View > Output** and then from the dropdown, select **Polyspace**. If the extension starts without errors, you see a message such as:

```
11/25/2020 3:59:37 PM.005: Please wait while Polyspace Connector is starting on port '9091'...
11/25/2020 3:59:41 PM.229: Polyspace Connector has started successfully.
```

The Polyspace Connector is an internal server that handles communication between the Polyspace as You Code analysis engine and the Visual Studio extension. If the default port is not available, the extension increments the port number and attempts to start the Polyspace Connector on the next port. If you use multiple Visual Studio instances, you can run Polyspace as You Code on all the instances. The Polyspace Connector in each instance uses a different port.

Run Analysis on Save

By default, Polyspace as You Code is configured to run analysis on save. Analysis results appear within a few seconds but in case of an error, you can check the progress of analysis on the **Output** pane.

After analysis, results appear as source code markers (lines below source code tokens). You also see the error locations as red circles in the scroll bar on the left.

To disable analysis on save:

- 1 Select **Tools > Options**.
- 2 On the **Polyspace** node, in the **Analysis launch mode** section, select **Manually**.

If results do not appear, see “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 5-83.

Run Analysis on Demand

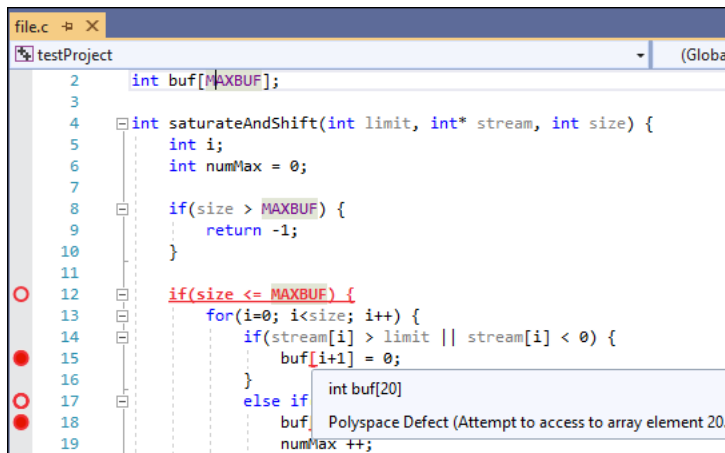
You can also explicitly start a Polyspace analysis. To start an analysis, right-click a source file in the Visual Studio **Solution Explorer** or right-click on the source file content itself, and select **Run Polyspace analysis**.

Review Results

After analysis, the results appear in two forms:

- As source code markers (with a line below source code tokens).

You can hover on a source code token to see more details about a result.



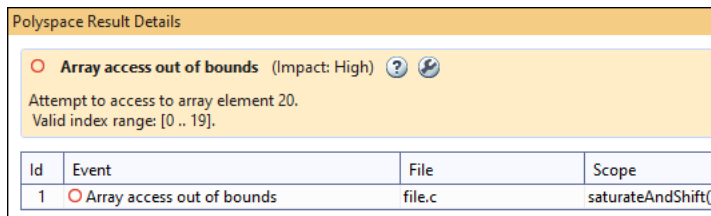
- In a list on the **Polyspace Results List** pane.

To open the pane, select **View > Other Windows > Polyspace Results List**.

Polyspace Results List						
Project: testProject File: file.c Found: 5						
Family	Ln	Ch	Check	Type	Group	Information
○	12	5	Useless if	Defect	Data flow	Impact: Medium
○	15	20	Array access out of bounds	Defect	Static memory	Impact: High
○	17	31	Invalid use of = operator	Defect	Programming	Impact: Medium
○	18	20	Array access out of bounds	Defect	Static memory	Impact: High
○	22	20	Array access out of bounds	Defect	Static memory	Impact: High

The results list shows results only for the file that is currently active in the IDE. For instance, if you switch to another file, the results list shows defects found in the new file that is active.

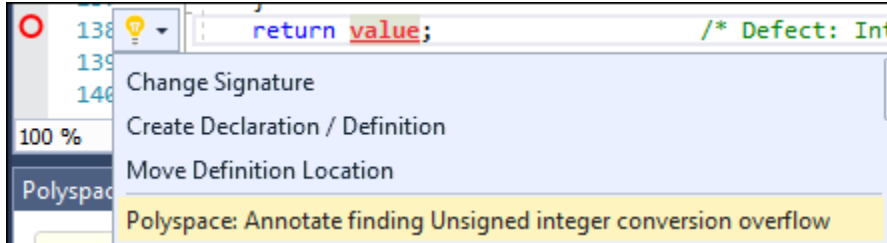
If you select a result in this list, you see further details of the result on the **Polyspace Result Details** pane.



Justify Results Using Code Annotations

If you decide not to fix a result, you can add code annotations to the result to avoid another review. If the annotations follow a specific syntax, subsequent Polyspace as You Code runs can read these annotations and suppress the corresponding results.

To add a code annotation, click the source code token containing a result. Click the light bulb icon that appears and select **Polyspace Annotate finding *result_name***. The annotation is entered on the same line as the result.



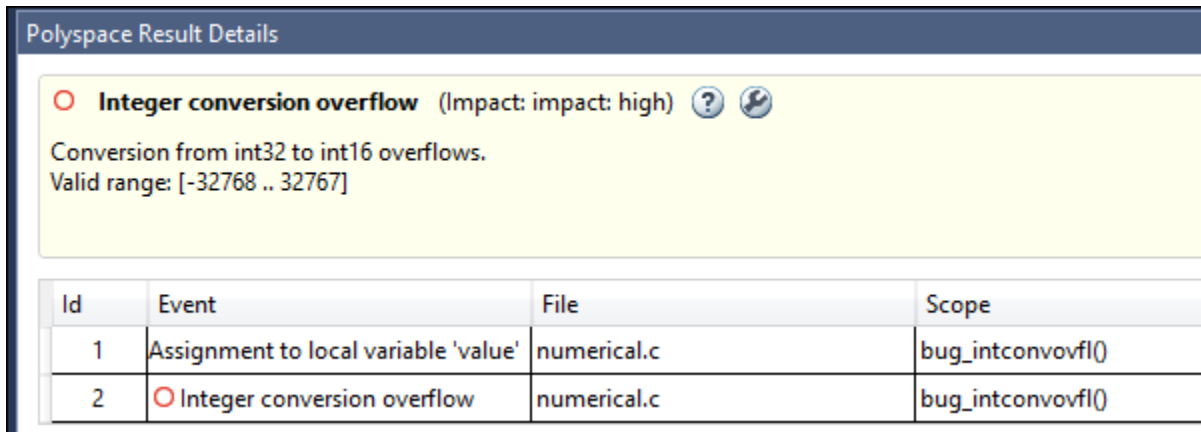
See also:

- “Hide Known or Acceptable Polyspace Results” on page 2-5
- “Short Names of Bug Finder Defect Checkers” on page 2-12

View Help

You can see more information on a type of result by visiting the context-sensitive help page for the result.

- To open the context-sensitive help for a result, first open the **Polyspace Result Details** pane for a result. Then, click the question mark icon next to the result details.
- To navigate directly to the **Fix** section of the context-sensitive help for a result, click the wrench icon next to the result details.



You can also open the full searchable documentation for the Polyspace as You Code extension from within Visual Studio. To open the documentation, select **Help > Open Polyspace Product Help**.

Configure Checkers and Other Settings

By default, Polyspace as You Code checks for defects that are likely to be of most interest to developers. You can expand the set of checkers and perform other configuration through the Polyspace as You Code extension settings in Visual Studio. To open the settings, select **Tools > Options** and specify appropriate settings on the **Polyspace** node.

For instance, you might want to:

- Enable or disable certain checkers.

See “Configure Checkers for Polyspace as You Code in Visual Studio” on page 5-61.

- See only new results.

See “Baseline Polyspace as You Code Results in Visual Studio” on page 5-41.

For the full list of settings, see “Configure Polyspace as You Code Extension in Visual Studio” on page 5-2.

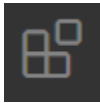
Run Polyspace as You Code in Visual Studio Code and Review Results

You can choose to run Polyspace as You Code on each save in the Visual Studio Code IDE, or at will. The analysis runs on the file that is currently active in the IDE. After analysis, you see bugs and coding standard violations as source code markers or in a separate list.

Confirm Installation of Extension

To confirm that your Visual Studio Code installation has the Polyspace as You Code extension, check the list of extensions installed.

In Visual Studio Code, select **View > Extensions** or click this button on the left:



Look for Polyspace as You Code in the list of extensions installed.


Run Analysis on Save

By default, Polyspace as You Code is configured to run analysis on save. Analysis results appear within a few seconds on the source code. In case of an error, you see a popup with the error message. To diagnose further, select **View > Output**. On the **OUTPUT** pane, from the dropdown on the upper right, select **Polyspace as You Code**.

After analysis, results appear as source code markers (wavy lines below source code tokens). You also see the error locations as red marks on the scroll bar. Click an error location to navigate to the corresponding source code.

To disable analysis on save:

1

On the **EXTENSIONS** pane, click the  icon next to **Polyspace as You Code** and select **Extension Settings**.

2 On the **Workspace** tab, for **Analysis Launch Mode**, select **Manually**.

If results do not appear, see “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 5-83.

Run Analysis on Demand

You can also explicitly start a Polyspace analysis. To start an analysis, do one of the following:

- Right click a source file in the **EXPLORER** pane or right-click on the file content itself and select **Polyspace: Analyze Active File**.
- With your cursor in the source file, press **Ctrl + Shift + Alt + A**.

Review Results

After analysis, the results appear in two forms:

- As source code markers (with a wavy line below source code tokens).

You can hover on a source code token to see more details about a result.

- In a list on the **PROBLEMS** window.

To open the window, select **View > Problems**.

```

C example.c X C initialisations.c
C example.c > Pointer_Arithmetic(void)
99     if (10 > 0) { //get_bus_status() > 0)
100         if (get_oil_pressure() > 0) { // P
101             *p = 5; /* Out of bounds */
102         } else {
103             i++;
104         }
105     }
106
107     i = 5; //get_bus_status();
108

```

TERMINAL PROBLEMS 17 OUTPUT DEBUG CONSOLE

✓ C example.c 17

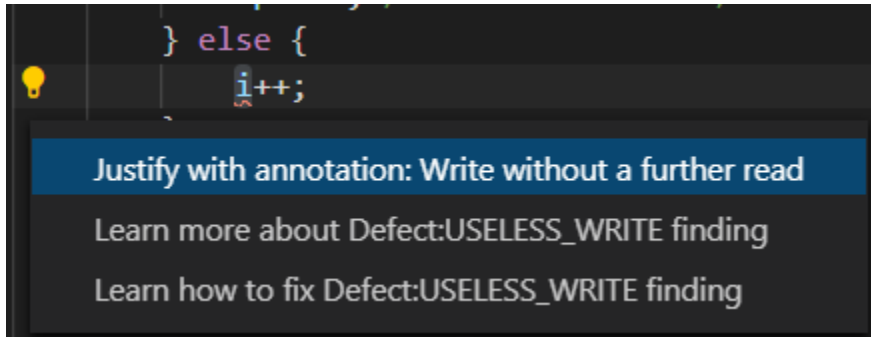
- ⊗ Variable 'y' is never read after this point.Variable 'y' is neve
- ⊗ Variable 'y' is never read after this point.Variable 'y' is neve
- ⊗ Unnecessary code, if-condition is always true.Unnecessary
- ✓ ⚡ Variable 'i' is rewritten later without an intermediate read.V

example.c[107, 5]: Assignment to local variable 'i'

Justify Results Using Code Annotations

If you decide not to fix a result, you can add code annotations to the result to avoid having to fix the result again. If the annotations follow a specific syntax, subsequent Polyspace as You Code runs can read these annotations and suppress the corresponding results.

To add a code annotation, click the light bulb icon beside the source code token containing a result and select **Justify with annotation: *result_name***. The annotation is entered on the same line as the result.



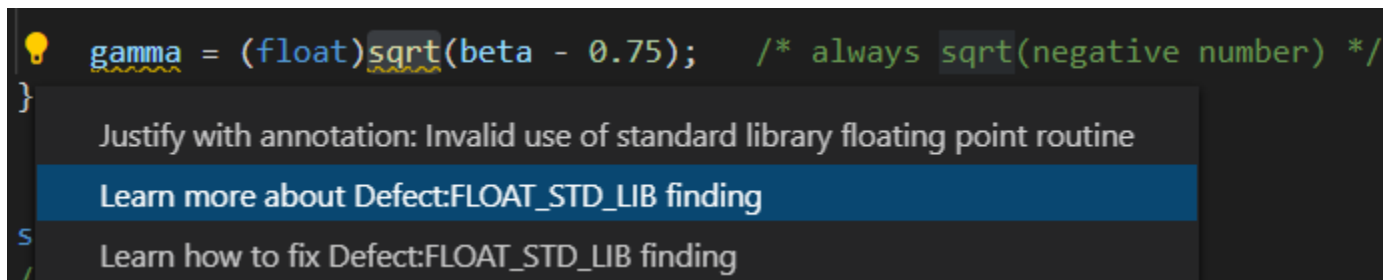
See also:

- “Hide Known or Acceptable Polyspace Results” on page 2-5
- “Short Names of Bug Finder Defect Checkers” on page 2-12

View Context-Sensitive Help for Result

You can see more information on a type of result by visiting the context-sensitive help page for the result.

- To open the context-sensitive help for a result, click the light bulb icon beside the source code token containing a result and select **Learn more about: *result_name***.
- To navigate directly to the **Fix** section of the context-sensitive help for a result, click the light bulb icon beside the source code token containing a result and select **Learn how to fix *result_name***.



You can also open the full searchable documentation for the Polyspace as You Code extension from within Visual Studio Code. To open the documentation, select **View > Command Palette**. In the command palette, select **Polyspace: Open Documentation**.

Configure Checkers and Other Settings

By default, Polyspace as You Code checks for defects that are likely to be of most interest to developers. You can expand the set of checkers and perform other configuration through the Polyspace as You Code extension settings in Visual Studio Code. To open the settings, on the

EXTENSIONS pane, click the  icon next to **Polyspace as You Code** and select **Extension Settings**.

For instance, you might want to:

- Enable or disable certain checkers.
See “Configure Checkers for Polyspace as You Code in Visual Studio Code” on page 5-64.
- See only new results.
See “Baseline Polyspace as You Code Results in Visual Studio Code” on page 5-45.

For the full list of settings, see “Configure Polyspace as You Code Extension in Visual Studio Code” on page 5-6.

Run Polyspace as You Code in Eclipse and Review Results

This topic describes how to run a single-file analysis in Eclipse using Polyspace as You Code. For Polyspace desktop products such as Polyspace Bug Finder, see the topic "Polyspace Analysis in Eclipse" in the Polyspace Bug Finder documentation.

You can choose to run Polyspace as You Code on each save in the Eclipse IDE, or at will. The analysis runs on the file that is currently active in the IDE. After analysis, you see bugs and coding standard violations as source code markers or in a separate list.

Confirm Installation of Plugin

To confirm that your Eclipse installation has the Polyspace as You Code plugin, check the list of plugins installed.

- 1 Select **Help > About Eclipse**.
- 2 Select **Installation Details** and browse through the list of installed plugins.

You can also confirm that the extension starts as expected on the IDE console. To open the console explicitly, select **Window > Show View > Console**. If the extension starts without errors, you see a message such as:

```
11/25/2020 3:59:37 PM.005: Starting Polyspace Connector on port 9093...  
11/25/2020 3:59:41 PM.229: Polyspace Connector successfully started
```

The Polyspace Connector is an internal server that handles communication between the Polyspace as You Code analysis engine and the Eclipse plugin. If the default port is not available, the plugin increments the port number and attempts to start the Polyspace Connector on the next port.

Run Analysis on Save

By default, Polyspace as You Code is configured to run analysis on save. Follow the progress of analysis on the IDE console.

After analysis, results appear as source code markers (lines below source code tokens). You also see the error locations as red marks on the scroll bar. Click an error location to navigate to the corresponding source code.

To disable analysis on save, select **Polyspace > Preferences** and select **Manually for Analysis launch mode**.

If results do not appear, see “Troubleshoot Failed Analysis or Unexpected Results in Polyspace as You Code” on page 5-83.

Run Analysis on Demand

If you disable automatic launch on save, you can also explicitly start a Polyspace analysis. To start an analysis, right-click the source code and select **Run Polyspace as You Code**.

Review Results

After analysis, the results appear in two forms:

- As source code markers (with a line below source code tokens).

You can click the circle on the left next to an underlined source code token to see more details about a result.

- In a list on the **Results List** pane.

If the list does not open automatically, select **Polyspace > Show View > Show Results List**. If you select a result in this list, you see further details of the result on the **Result Details** pane.

The screenshot displays the Eclipse IDE interface. The top pane shows C code with Polyspace markers. The bottom pane shows the 'Results List' view with a table of defects.

Type	Group	Check	Information	Line	...
Defect	Numerical	Integer conversion overflow	Impact: High	112	12
Defect	Numerical	Absorption of float operand	Impact: High	290	17
Defect	Numerical	Float division by zero	Impact: High	70	16
Defect	Numerical	Integer division by zero	Impact: High	36	14
Defect	Numerical	Invalid use of standard library integer ...	Impact: High	314	16
Defect	Numerical	Invalid use of standard library integer ...	Impact: High	312	16
Defect	Numerical	Invalid use of standard library floating...	Impact: High	343	11
Defect	Numerical	Float conversion overflow	Impact: High	185	12
Defect	Numerical	Integer constant overflow	Impact: Medium	427	14
Defect	Numerical	Integer constant overflow	Impact: Medium	456	21

Justify Results Using Code Annotations

If you decide not to fix a result, you can add code annotations to the result to avoid having to fix the result again. If the annotations follow a specific syntax, subsequent Polyspace as You Code runs can read these annotations and suppress the corresponding results.

To add a code annotation, right-click the result on the **Results List** pane and select **Annotate Code and Hide Result**. The annotation is entered on the same line as the result.

The screenshot shows the 'Results List' pane in Polyspace. The table below represents the data visible in the pane. The row for 'Invalid use of standard library integer routine' is selected, and a context menu is open over it.

Type	Group	Check	Information	Line	...
Defect	Numerical	Integer conversion overflow	Impact: High	112	12
Defect	Numerical	Absorption of float operand	Impact: High	290	17
Defect	Numerical	Float division by zero	Impact: High	70	16
Defect	Numerical	Integer division by zero	Impact: High	36	14
Defect	Numerical	Invalid use of standard library integer routine	Impact: High	314	16
Defect	Numerical	Invalid use of standard library integer routine	Impact: High	314	16
Defect	Numerical	Invalid use of standard library integer routine	Impact: High	314	16
Defect	Numerical	Float conversion overflow	Impact: High	150	21
Defect	Numerical	Integer constant overflow	Impact: High	150	21
Defect	Numerical	Integer constant overflow	Impact: Medium	150	21

The context menu for the selected row contains the following options:

- Annotate Code and Hide Result
- Add Pre-Justification To Clipboard
- Select All "Invalid use of standard library integer routine" Results

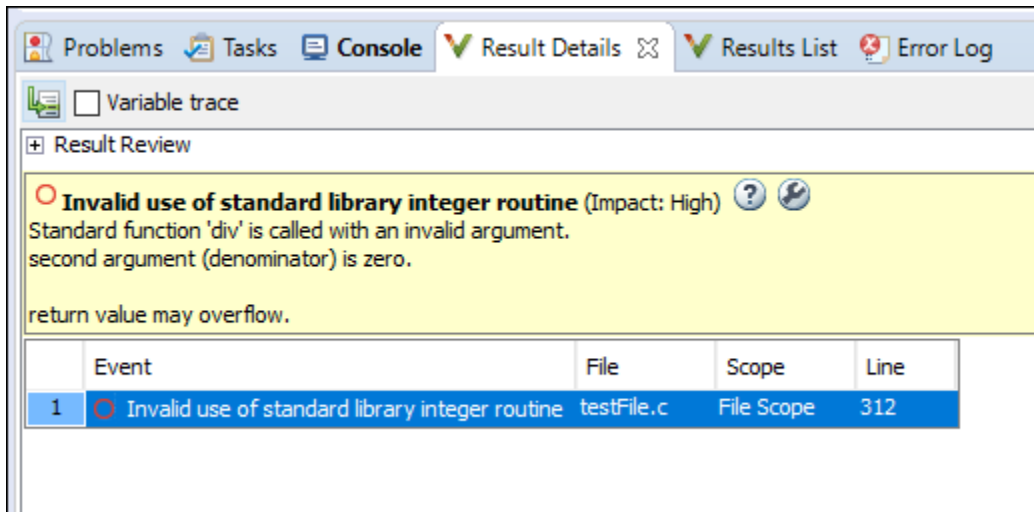
See also:

- “Hide Known or Acceptable Polyspace Results” on page 2-5
- “Short Names of Bug Finder Defect Checkers” on page 2-12

View Context-Sensitive Help for Result

You can see more information on a type of result by visiting the context-sensitive help page for the result.

- To open the context-sensitive help for a result, first open the **Result Details** pane for a result. Then, click the question mark icon next to the result details.
- To navigate directly to the **Fix** section of the context-sensitive help for a result, click the wrench icon next to the result details.



You can also open the full searchable documentation for the Polyspace as You Code extension from within Eclipse. To open the documentation, select **Polyspace > Help**.

Configure Checkers and Other Settings

By default, Polyspace as You Code checks for defects that are likely to be of most interest to developers. You can expand the set of checkers and perform other configuration through the Polyspace as You Code plugin settings in Eclipse. To open the settings, select **Polyspace > Preferences** or **Polyspace > Configure Project**.

For instance, you might want to:

- Enable or disable certain checkers.

See “Configure Checkers for Polyspace as You Code in Eclipse” on page 5-57.

- See only new results.

See “Baseline Polyspace as You Code Results in Eclipse” on page 5-50.

For the full list of settings, see “Configure Polyspace as You Code Plugin in Eclipse” on page 5-17.

Run Polyspace as You Code from Command Line and Export Results

You can run Polyspace as You Code on source files directly at the command line.

For IDEs that are not directly supported with a Polyspace as You Code plugin, you can open a terminal within the IDE and run the commands, or create a menu item to run the commands on the file currently open in the IDE. You can even incorporate these commands in a makefile, so that building your code also runs static analysis on the code. See also “Integrate Polyspace as You Code in IDEs and Editors Without Plugins” on page 6-17.

Add Install Folder to Path

To avoid typing the full path to Polyspace as You Code commands, add the paths to these commands to the PATH environment variable on your operating system.

The paths in the default installation folder are the following:

Windows	C:\Program Files\Polyspace as You Code\R2021a\polyspace\bin
Linux	/usr/local/PolyspaceAsYouCode/R2021a/polyspace/bin

After you add the paths, you can enter commands such as the following in a terminal without errors:

```
polyspace-bug-finder-access -help
```

Run Analysis and See Results on Console

To run Polyspace as You Code, use the `polyspace-bug-finder-access` command. Export the results to the console using the `polyspace-results-export` command.

```
polyspace-bug-finder-access -sources filename
polyspace-results-export -format console
```

In this example, the `polyspace-bug-finder-access` command generates results in the current folder. The `polyspace-results-export` command reads results from the current folder and exports to the console.

The analysis typically takes a few seconds to complete (slightly longer for C++ files). If the analysis fails to complete, further details of the error appear on the console. You can use the option `-no-quiet` to see a more detailed analysis log on the console.

Store Results in Specific Folder

To use a specific results folder *resultsFolder* instead of the current folder, change the preceding lines as follows:

```
polyspace-bug-finder-access -sources filename -results-dir resultsFolder
polyspace-results-export -format console -results-dir resultsFolder
```

Export Results to JSON Format (SARIF Output)

Instead of displaying analysis results on the console, you can export the results to a JSON file. You can then parse this file using a JSON parser method in any language that you want.

```
polyspace-bug-finder-access -sources filename.c
polyspace-results-export -format json-sarif -output-name outputFilePath
```

Here, *outputFilePath* is the full path to the JSON file.

The JSON format follows the standard notation provided by the OASIS Static Analysis Results Interchange Format (SARIF).

Specify Analysis Options by Using Options Files

To adapt the Polyspace analysis configuration to your development environment and requirements, you have to modify the default configuration through command-line options such as `-compiler`. Options files are a convenient way to collect multiple options together and reuse them across projects.

Options files are text files with one option per line. For instance, the content of an options file can look like this:

```
# Options for Polyspace analysis
# Options apply to all projects in Controller module
-compiler visual16.x
-D _WIN32
-checkers-activation-file "Z:\utils\checkers.xml"
```

Specify an options file using the option `-options-file`. For instance:

```
polyspace-bug-finder-access -sources file.c -options-file "Z:\utils\polyspace\options.txt"
```

See also “Options Files for Polyspace Analysis” on page 5-22. For all options available with Polyspace as You Code, see “Polyspace as You Code Analysis Engine Options”.

Create Options File by Analyzing Build

Instead of entering options by hand in an options file, you can create an options file with all Polyspace options required for compilation by analyzing your build system. For instance, you can trace your build command and save the options in a file `buildOptions.txt` that you can use for the subsequent analysis.

```
polyspace-configure -no-sources -output-options-file buildOptions.txt buildCommand
polyspace-bug-finder-access -sources file.c -options-file buildOptions.txt
```

Here, *buildCommand* is a build command that performs a full build of your source code, for instance, `make -B` or `make --always-make`. For build systems that can output compilation options in the JSON compilation database format, you can obtain the options from the JSON file:

```
polyspace-configure -no-sources -output-options-file buildOptions.txt -compilation-database jsonFile
```

Here, *jsonFile* is the full path to the compilation database JSON file.

You can also append a second options file with options related to the analysis such as checkers. For instance, if the second options file is called `checkersOptions.txt`, you can run Polyspace as You Code as follows:

```
polyspace-bug-finder-access -sources file.c -options-file buildOptions.txt -options-file checkersOptions.txt
```

See Also

`polyspace-bug-finder-access` | `polyspace-configure` | `polyspace-results-export`

More About

- “Options Files for Polyspace Analysis” on page 5-22
- “Integrate Polyspace as You Code in IDEs and Editors Without Plugins” on page 6-17

Integrate Polyspace as You Code in IDEs and Editors Without Plugins

Polyspace as You Code supports these IDEs with extensions or plugins: Visual Studio, Visual Studio Code, and Eclipse. Even if an IDE is not explicitly supported with a Polyspace as You Code plugin, you can open a console within the IDE and run Polyspace as You Code commands, or create a menu item to run the commands on the file currently open in the IDE.

This topic demonstrates how to integrate Polyspace as You Code in a simple editor such as Notepad++. You can use the principles here to integrate Polyspace as You Code in most editors or IDEs.

Overview of Approach

In supported IDEs, a Polyspace as You Code extension allows you to analyze the file that is currently active in the IDE and see results within the IDE (as source code markers or in a list). In an unsupported IDE or editor, you can partly emulate this workflow, that is, run analysis within the IDE and view results. The workflow consists of two steps:

- *Running analysis and exporting results*

Most IDEs or editors provides environment variables that resolve to the current file path. You can create menu items that execute a script which runs the `polyspace-bug-finder-access` command on this path. In the same script, you can export the results to the IDE console.

- *Parsing console output to allow navigation to line*

Each Polyspace as You Code result in the console output starts with a line in this format:

```
filepath:lineNumber:columnNumber
```

Here, *filepath* is the path to the current file, *lineNumber* is the line number of the result, and *columnNumber* is the column that starts the token with the result. For instance:

```
C:\MyProj\myFile.c:17:31:
```

indicates that the file `C:\MyProj\myFile.c` contains a result on line 17, starting from column 31. If you can parse the console output, you can enable a navigation to line 31 to the start of the token containing the result.

Integration Steps

This example shows an integration of Polyspace as You Code in a simple editor such as Notepad++. You can follow similar integration steps in other editors such as GNU Emacs, Sublime Text, and so on.

Step 1: Set Up Script Runs from Within Editor

In Notepad++, you can use a plugin such as NppExec that allows you to execute any script from within the editor. The editor also provides the environment variable `$(FULL_CURRENT_PATH)` that resolves to the file that is currently active in the IDE.

The simplest script that can be run within the plugin can be the following:

```
cd $(CURRENT_DIRECTORY)
set POLYSPACE_EXECUTABLES_FOLDER=C:\Program Files\Polyspace as You Code\R2021a\polyspace\bin
set POLYSPACE_ENGINE=$(POLYSPACE_EXECUTABLES_FOLDER)\polyspace-bug-finder-access.exe
set POLYSPACE_REPORT_EXPORTER=$(POLYSPACE_EXECUTABLES_FOLDER)\polyspace-results-export.exe
$(POLYSPACE_ENGINE) -sources $(FULL_CURRENT_PATH)
$(POLYSPACE_REPORT_EXPORTER) -results-dir . -format console
```

In practice, you might want to specify additional analysis options using an options file. If the options file is called `polyspace_options.txt`, the command to run Polyspace as You Code in the preceding script can be replaced with:

```
$(POLYSPACE_ENGINE) -sources $(FULL_CURRENT_PATH) -options-file polyspace_options.txt
```

See also “Options Files for Polyspace Analysis” on page 5-22.

For other command-line examples, see “Run Polyspace as You Code from Command Line and Export Results” on page 6-14. For instance, instead of exporting to the console directly, you can export the results to a JSON format, use a JSON parser to package the results, and then export them to the console or use them in some other way.

Step 2: Set Up Parsing of Console Output

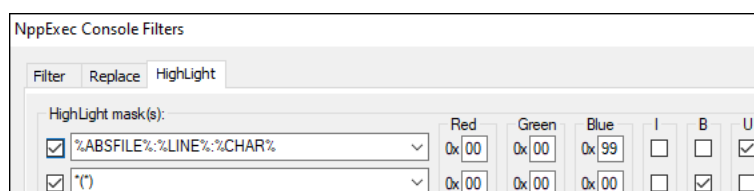
The NppExec plugin allows you to parse console output and navigate to the appropriate line of code. You can also optionally apply specific formatting to the console output.

For instance, your console output can look like the following:

```
C:\MyProj\myFile.c:12:8:
  Useless if (DEFECT:USELESS_IF)
  Details: Unnecessary code, if-condition is always true.
C:\MyProj\myFile.c:15:20:
  Array access out of bounds (DEFECT:OUT_BOUND_ARRAY)
  Details: Attempt to access to array element 20.
  Valid index range: [0 .. 19].
```

You can set up the output so that clicking a link directly takes you to the start of the relevant token on the relevant line of code.

To set up this presentation of results, select **Plugins > NppExec** and then select the **Console Output Filters** option. The following options allow the previous presentation of results:



The first highlight mask indicates that lines having the format

```
.....
```

contain the absolute path to the file before the first colon, the line number between the first and second colon, and the column number (or character number) after the second colon. The mask reads the information (file, line and column), underlines these lines and colors them blue.

The second highlight mask simply bolds lines having the format

```
...( )
```

These lines contain the result name, for instance, the name of a defect.

Further Exploration

The official Polyspace as You Code extensions enable other actions such as analyzing build commands, configuring checkers, and downloading baselines from the Polyspace Access web server. In a real development environment, you want to analyze your build commands to emulate your compilation toolchain as closely as possible, configure the checkers that are most meaningful to you, and baseline results so that you focus only on new results coming from your changes.

You can extend the approach described here to create menu items in your IDE or editor for all these actions. For more information on these workflows from the command line, see:

- “Generate Build Options for Polyspace as You Code Analysis at the Command Line” on page 5-38
- “Configure Checkers for Polyspace as You Code at the Command Line” on page 5-68
- “Baseline Polyspace as You Code Results on Command Line” on page 5-53

With your IDE or editor set up for Polyspace as You Code, you can create a quality gate for submission. You can set up a configuration with checkers for which you do not want any finding in your submission. Before submitting a file, you can make sure that you have fixed all findings from those checkers.

See Also

`polyspace-bug-finder-access | polyspace-results-export`

Coding Rule Sets and Concepts

- “Polyspace MISRA C:2004 and MISRA AC AGC Checkers” on page 7-2
- “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 7-3
- “Polyspace MISRA C:2012 Checkers” on page 7-38
- “Essential Types in MISRA C:2012 Rules 10.x” on page 7-39
- “Unsupported MISRA C:2012 Guidelines” on page 7-41
- “Polyspace MISRA C++ Checkers” on page 7-42
- “Unsupported MISRA C++ Coding Rules” on page 7-43
- “Polyspace JSF AV C++ Checkers” on page 7-47
- “JSF AV C++ Coding Rules” on page 7-48

Polyspace MISRA C:2004 and MISRA AC AGC Checkers

The Polyspace MISRA C:2004 checker helps you comply with the MISRA C 2004 coding standard.¹

When MISRA C rules are violated, the MISRA C checker enables Polyspace software to provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis.

The MISRA C checker can check nearly all of the **142** MISRA C:2004 rules.

The MISRA AC AGC checker checks rules from the OBL (obligatory) and REC (recommended) categories specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

There are subsets of MISRA coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your results. When you set up rule checking, you can select these subsets directly. These subsets are defined in:

- “Software Quality Objective Subsets (C:2004)” on page 1-43
- “Software Quality Objective Subsets (AC AGC)” on page 1-47

Note The Polyspace MISRA checker is based on MISRA C:2004, which also incorporates MISRA C Technical Corrigendum.

See Also

More About

- “MISRA C:2004 and MISRA AC AGC Coding Rules” on page 7-3

1. MISRA and MISRA C are registered trademarks of MIRA Ltd., held on behalf of the MISRA Consortium.

MISRA C:2004 and MISRA AC AGC Coding Rules

In this section...

“Supported MISRA C:2004 and MISRA AC AGC Rules” on page 7-3

“Troubleshooting” on page 7-3

“List of Supported Coding Rules” on page 7-3

“Unsupported MISRA C:2004 and MISRA AC AGC Rules” on page 7-36

Supported MISRA C:2004 and MISRA AC AGC Rules

The following tables list MISRA C:2004 coding rules that the Polyspace coding rules checker supports. Details regarding how the software checks individual rules and any limitations on the scope of checking are described in the “Polyspace Specification” column.

Note The Polyspace coding rules checker:

- Supports MISRA-C:2004 Technical Corrigendum 1 for rules 4.1, 5.1, 5.3, 6.1, 6.3, 7.1, 9.2, 10.5, 12.6, 13.5, and 15.0.
- Checks rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

The software reports most violations during the compile phase of an analysis. However, the software detects violations of rules 9.1 (Non-initialized variable), 12.11 (one of the overflow checks) using `-scalar-overflows-checks signed-and-unsigned`, 13.7 (dead code), 14.1 (dead code), 16.2 and 21.1 during code analysis, and reports these violations as run-time errors.

Note Some violations of rules 13.7 and 14.1 are reported during the compile phase of analysis.

Troubleshooting

If you expect a rule violation but do not see it, check out .

List of Supported Coding Rules

- “Environment” on page 7-5
- “Language Extensions” on page 7-6
- “Documentation” on page 7-9
- “Character Sets” on page 7-9
- “Identifiers” on page 7-9
- “Types” on page 7-11
- “Constants” on page 7-11
- “Declarations and Definitions” on page 7-12

- “Initialisation” on page 7-15
- “Arithmetic Type Conversion” on page 7-16
- “Pointer Type Conversion” on page 7-19
- “Expressions” on page 7-20
- “Control Statement Expressions” on page 7-22
- “Control Flow” on page 7-25
- “Switch Statements” on page 7-27
- “Functions” on page 7-28
- “Pointers and Arrays” on page 7-29
- “Structures and Unions” on page 7-30
- “Preprocessing Directives” on page 7-30
- “Standard Libraries” on page 7-33
- “Runtime Failures” on page 7-36

Environment

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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.	<p>The text <i>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</i> precedes each of the following messages:</p> <ul style="list-style-type: none"> • ANSI® C does not allow '#include_next' • ANSI C does not allow macros with variable arguments list • ANSI C does not allow '#assert' • ANSI C does not allow '#unassert' • ANSI C does not allow testing assertions • ANSI C does not allow '#ident' • ANSI C does not allow '#sccs' • text following '#else' violates ANSI standard. • text following '#endif' violates ANSI standard. • text following '#else' or '#endif' violates ANSI standard. 	All the supported extensions lead to a violation of this MISRA rule. Standard compilation error messages do not lead to a violation of this MISRA rule and remain unchanged.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
1.1 (cont.)		<p>The text <i>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</i> precedes each of the following messages:</p> <ul style="list-style-type: none"> • ANSI C90 forbids 'long long int' type. • ANSI C90 forbids 'long double' type. • ANSI C90 forbids long long integer constants. • Keyword 'inline' should not be used. • Array of zero size should not be used. • Integer constant does not fit within unsigned long int. • Integer constant does not fit within long int. • Too many nesting levels of #includes: N_1. The limit is N_0. • Too many macro definitions: N_1. The limit is N_0. • Too many nesting levels for control flow: N_1. The limit is N_0. • Too many enumeration constants: N_1. The limit is N_0. 	

Language Extensions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
2.1	Assembly language shall be encapsulated and isolated.	Assembly language shall be encapsulated and isolated.	<p>No warnings if code is encapsulated in the following:</p> <ul style="list-style-type: none"> • asm functions or asm pragma • Macros

N.	MISRA Definition	Messages in report file	Polyspace Implementation
2.2	Source code shall only use <code>/** */</code> style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule Note: This rule cannot be annotated in the source code.
2.3	The character sequence <code>/*</code> shall not be used within a comment	The character sequence <code>/*</code> shall not appear within a comment.	This rule violation is also raised when the character sequence <code>/*</code> inside a C++ comment. Note: This rule cannot be annotated in the source code.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
2.4	Sections of code should not be "commented out"	Sections of code should not be "commented out"	<p>The checker uses internal heuristics to detect commented out code. For instance, characters such as #, ;, { or } indicate comments that might potentially contain code. These comments are then evaluated against other metrics to determine the likelihood of code masquerading as comment. For instance, several successive words without a symbol in between reduces this likelihood.</p> <p>The checker does not flag the following comments even if they contain code:</p> <ul style="list-style-type: none"> • Doxygen comments beginning with /** or /*!. • Comments that repeat the same symbol several times, for instance, the symbol = here: <pre data-bbox="1109 1081 1299 1165"> /** ===== * A comment * =====*/ </pre> • Comments on the first line of a file. • Comments that mix the C style (/* */) and C++ style (//). <p>The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.</p>

Documentation

Rule	MISRA Definition	Messages in report file	Polyspace Implementation
3.4	All uses of the <i>#pragma</i> directive shall be documented and explained.	All uses of the <i>#pragma</i> directive shall be documented and explained.	To check this rule, you must list the pragmas that are allowed in source files by using the option <code>Allowed pragmas (-allowed-pragmas)</code> . If Polyspace finds a pragma not in the allowed pragma list, a violation is raised. For more on analysis options, see the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server

Character Sets

N.	MISRA Definition	Messages in report file	Polyspace Implementation
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.	<code>\<character></code> is not an ISO C escape sequence Only those escape sequences which are defined in the ISO C standard shall be used.	
4.2	Trigraphs shall not be used.	Trigraphs shall not be used.	Trigraphs are handled and converted to the equivalent character but lead to a violation of the MISRA rule

Identifiers

N.	MISRA Definition	Messages in report file	Polyspace Implementation
5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters	Identifier 'XX' should not rely on the significance of more than 31 characters.	All identifiers (global, static and local) are checked. For easier review, the rule checker shows all identifiers that have the same first 31 characters as one rule violation. You can see all instances of conflicting identifier names in the event history of that rule violation. <i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" on page 5-80.</i>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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.	<ul style="list-style-type: none"> • Local declaration of XX is hiding another identifier. • Declaration of parameter XX is hiding another identifier. 	Assumes that rule 8.1 is not violated.
5.3	A typedef name shall be a unique identifier	{typedef name}'%s' should not be reused. (already used as {typedef name} at %s:%d)	Warning when a typedef name is reused as another identifier name.
5.4	A tag name shall be a unique identifier	{tag name}'%s' should not be reused. (already used as {tag name} at %s:%d)	<p>Warning when a tag name is reused as another identifier name</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" on page 5-80.</i></p>
5.5	No object or function identifier with a static storage duration should be reused.	{static identifier/parameter name}'%s' should not be reused. (already used as {static identifier/parameter name} with static storage duration at %s:%d)	<p>Warning when a static name is reused as another identifier name</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
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.	{member name}'%s' should not be reused. (already used as {member name} at %s:%d)	<p>Warning when an idf in a namespace is reused in another namespace</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" on page 5-80.</i></p>
5.7	No identifier name should be reused.	{identifier}'%s' should not be reused. (already used as {identifier} at %s:%d)	<p>No violation reported when:</p> <ul style="list-style-type: none"> • Different functions have parameters with the same name • Different functions have local variables with the same name • A function has a local variable that has the same name as a parameter of another function

Types

N.	MISRA Definition	Messages in report file	Polyspace Implementation
6.1	The plain char type shall be used only for the storage and use of character values	Only permissible operators on plain chars are '=', '==' or '!=' operators, explicit casts to integral types and '?' (for the 2nd and 3rd operands)	Warning when a plain char is used with an operator other than =, ==, !=, explicit casts to integral types, or as the second or third operands of the ? operator.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.	<ul style="list-style-type: none"> Value of type plain char is implicitly converted to signed char. Value of type plain char is implicitly converted to unsigned char. Value of type signed char is implicitly converted to plain char. Value of type unsigned char is implicitly converted to plain char. 	Warning if value of type plain char is implicitly converted to value of type signed char or unsigned char.
6.3	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types	<i>typedefs</i> that indicate size and signedness should be used in place of the basic types.	No warning is given in typedef definition.
6.4	Bit fields shall only be defined to be of type <i>unsigned int</i> or <i>signed int</i> .	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type <i>signed int</i> shall be at least 2 bits long.	Bit fields of type signed int shall be at least 2 bits long.	No warning on anonymous signed int bitfields of width 0 - Extended to all signed bitfields of size <= 1 (if Rule 6.4 is violated).

Constants

N.	MISRA Definition	Messages in report file	Polyspace Implementation
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	<ul style="list-style-type: none"> Octal constants other than zero and octal escape sequences shall not be used. Octal constants (other than zero) should not be used. Octal escape sequences should not be used. 	

Declarations and Definitions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	<ul style="list-style-type: none"> • Function XX has no complete prototype visible at call. • Function XX has no prototype visible at definition. 	Prototype visible at call must be complete.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated	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.	Definition of function 'XX' incompatible with its declaration.	Assumes that rule 8.1 is not violated. The rule is restricted to compatible types. Can be turned to Off
8.4	If objects or functions are declared more than once their types shall be compatible.	<ul style="list-style-type: none"> • If objects or functions are declared more than once their types shall be compatible. • Global declaration of 'XX' function has incompatible type with its definition. • Global declaration of 'XX' variable has incompatible type with its definition. 	<p>Violations of this rule might be generated during the link phase.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" on page 5-80.</i></p>
8.5	There shall be no definitions of objects or functions in a header file	<ul style="list-style-type: none"> • Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file. • Fragment of function should not be defined in a header file. 	<p>Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that:</p> <ul style="list-style-type: none"> • Do not have initializers. • Do not have storage class specifiers, or have the <code>static</code> specifier
8.6	Functions shall always be declared at file scope.	Function 'XX' should be declared at file scope.	This rule maps to ISO/IEC TS 17961 ID addresscape.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function	Object 'XX' should be declared at block scope.	Restricted to static objects.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
8.8	An external object or function shall be declared in one file and only one file	Function/Object 'XX' has external declarations in multiple files.	<p>Restricted to explicit extern declarations (tentative definitions are ignored).</p> <p>Polyspace considers that variables or functions declared extern in a non-header file violate this rule.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" on page 5-80.</i></p>
8.9	An identifier with external linkage shall have exactly one external definition.	<ul style="list-style-type: none"> • Procedure/Global variable XX multiply defined. • Forbidden multiple tentative definitions for object XX • Global variable has multiple tentative definitions • Undefined global variable XX 	<p>The checker flags multiple definitions only if the definitions occur in different files.</p> <p>No warnings appear on predefined symbols.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" on page 5-80.</i></p>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required	Function/Variable XX should have internal linkage.	<p>Assumes that 8.1 is not violated. No warning if 0 uses.</p> <p>If your code does not contain a <code>main</code> function and you use options such as <code>-main-generator-writes-variables</code> with value <code>custom</code> to explicitly specify a set of variables to initialize, the checker does not flag those variables. The checker assumes that in a real application, the file containing the <code>main</code> must initialize the variables in addition to any file that currently uses them. Therefore, the variables must be used in more than one translation unit.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" on page 5-80.</i></p>
8.11	The <i>static</i> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	static storage class specifier should be used on internal linkage symbol XX.	
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization	Size of array 'XX' should be explicitly stated.	

Initialisation

N.	MISRA Definition	Messages in report file	Polyspace Implementation
9.1	All automatic variables shall have been assigned a value before being used.		<p>Checked during code analysis.</p> <p>Violations displayed as Non-initialized variable results.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results. In Code Prover, you can also see a difference in results based on your choice for the option <code>Verification level (-to)</code>. See the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server for more on analysis options and how to check for coding standard violations..</p>
9.2	Braces shall be used to indicate and match the structure in the nonzero initialisation of arrays and structures.	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.	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Arithmetic Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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 	<ul style="list-style-type: none"> • Implicit conversion of the expression of underlying type XX to the type XX that is not a wider integer type of the same signedness. • Implicit conversion of one of the binary operands whose underlying types are XX and XX • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary left hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not a wider integer type of the same signedness or • Implicit conversion of the binary ? left hand operand of underlying type XX to XX, but it is a complex expression. • Implicit conversion of complex integer expression of underlying type XX to XX. • Implicit conversion of non-constant integer expression of underlying type XX in function return whose expected type is XX. • Implicit conversion of non-constant integer expression of underlying type XX as argument of function whose corresponding parameter type is XX. 	<p>ANSI C base types order (signed char, short, int, long) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. The same interpretation is applied on the unsigned version of base types.</p> <p>An expression of bool or enum types has int as underlying type.</p> <p>Plain char may have signed or unsigned underlying type (depending on Polyspace target configuration or option setting).</p> <p>The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not token into account and it assumes that only signed unsigned int are used for bitfield (Rule 6.4).</p> <p>This rule violation is not produced on operations involving pointers.</p> <p>No violation reported when:</p> <ul style="list-style-type: none"> • The implicit conversion is a type widening, without change of signedness of integer • The expression is an argument expression or a return expression <p>No violation reported when the following are true:</p> <ul style="list-style-type: none"> • Implicit conversion applies to a constant expression and is a type widening, with a possible change of signedness of integer. • The conversion does not change the representation of

N.	MISRA Definition	Messages in report file	Polyspace Implementation
			<p>the constant value or the result of the operation.</p> <ul style="list-style-type: none"> The expression is an argument expression or a return expression or an operand expression of a non-bitwise operator. <p>Conversions of constants are not reported for these cases to avoid flagging too many violations. If the constant can be represented in both the original and converted type, the conversion is less of an issue.</p>
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 	<ul style="list-style-type: none"> Implicit conversion of the expression from XX to XX that is not a wider floating type. Implicit conversion of the binary ? right hand operand from XX to XX, but it is a complex expression. Implicit conversion of the binary ? right hand operand from XX to XX that is not a wider floating type or Implicit conversion of the binary ? left hand operand from XX to XX, but it is a complex expression. Implicit conversion of complex floating expression from XX to XX. Implicit conversion of floating expression of XX type in function return whose expected type is XX. Implicit conversion of floating expression of XX type as argument of function whose corresponding parameter type is XX. 	<p>ANSI C base types order (float, double) defines that T2 is wider than T1 if T2 is on the right hand of T1 or $T2 = T1$.</p> <p>No violation reported when:</p> <ul style="list-style-type: none"> The implicit conversion is a type widening The expression is an argument expression or a return expression.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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	Complex expression of underlying type XX may only be cast to narrower integer type of same signedness, however the destination type is XX.	<ul style="list-style-type: none"> • The rule checker raises a defect only if the result of a composite expression is cast to a different or wider essential type. For instance, in this example, a violation is shown in the first assignment to <code>i</code> but not the second. In the first assignment, a composite expression <code>i+1</code> is directly cast from a signed to an unsigned type. In the second assignment, the composite expression is first cast to the same type and then the result is cast to a different type. <pre>typedef int int32_T; typedef unsigned char uint8_T; int32_T i; i = (uint8_T)(i+1); /* Noncompliant */ i = (uint8_T) ((int32_T)(i+1)); /* Compliant */</pre> • ANSI C base types order (signed char, short, int, long) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T1 = T2. The same methodology is applied on the unsigned version of base types. • An expression of bool or enum types has int as underlying type. • Plain char may have signed or unsigned underlying type (depending on target configuration or option setting). • The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width

N.	MISRA Definition	Messages in report file	Polyspace Implementation
			is not taken into account and it assumes that only signed, unsigned int are used for bitfield (Rule 6.4).
10.4	The value of a complex expression of float type may only be cast to narrower floating type	Complex expression of XX type may only be cast to narrower floating type, however the destination type is XX.	ANSI C base types order (float, double) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T2 = T1.
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type <i>unsigned char</i> or <i>unsigned short</i> , the result shall be immediately cast to the underlying type of the operand	Bitwise [<< ~] is applied to the operand of underlying type [unsigned char unsigned short], the result shall be immediately cast to the underlying type.	
10.6	The "U" suffix shall be applied to all constants of <i>unsigned</i> types	No explicit 'U' suffix on constants of an unsigned type.	<p>Warning when the type determined from the value and the base (octal, decimal or hexadecimal) is unsigned and there is no suffix u or U.</p> <p>For example, when the size of the <code>int</code> and <code>long int</code> data types is 32 bits, the coding rule checker will report a violation of rule 10.6 for the following line:</p> <pre>int a = 2147483648;</pre> <p>There is a difference between decimal and hexadecimal constants when <code>int</code> and <code>long int</code> are not the same size.</p>

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Implementation
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type	Conversion shall not be performed between a pointer to a function and any type other than an integral type.	<p>Casts and implicit conversions involving a function pointer.</p> <p>Casts or implicit conversions from <code>NULL</code> or <code>(void*)0</code> do not give any warning.</p>
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	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.	<p>There is also a warning on qualifier loss</p> <p>This rule maps to ISO/IEC TS 17961 ID <code>alignconv</code>.</p>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
11.3	A cast should not be performed between a pointer type and an integral type	A cast should not be performed between a pointer type and an integral type.	Exception on zero constant. Extended to all conversions This rule maps to ISO/IEC TS 17961 ID <code>alignconv</code> .
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	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 <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	A cast shall not be performed that removes any <i>const</i> or <i>volatile</i> qualification from the type addressed by a pointer	Extended to all conversions

Expressions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
12.1	Limited dependence should be placed on C's operator precedence rules in expressions	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.	<ul style="list-style-type: none"> The value of '<i>sym</i>' depends on the order of evaluation. The value of volatile '<i>sym</i>' depends on the order of evaluation because of multiple accesses. 	<p>Rule 12.2 check assumes that no assignment in expressions that yield a Boolean values (rule 13.1).</p> <p>The expression is a simple expression of symbols. <code>i = i++;</code> is a violation, but <code>tab[2] = tab[2]++;</code> is not a violation.</p>
12.3	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	The <code>sizeof</code> operator should not be used on expressions that contain side effects.	No warning on volatile accesses
12.4	The right hand operand of a logical <code>&&</code> or <code> </code> operator shall not contain side effects.	The right hand operand of a logical <code>&&</code> or <code> </code> operator shall not contain side effects.	No warning on volatile accesses
12.5	The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions.	<ul style="list-style-type: none"> operand of logical <code>&&</code> is not a primary expression operand of logical <code> </code> is not a primary expression The operands of a logical <code>&&</code> or <code> </code> shall be primary-expressions. 	<p>During preprocessing, violations of this rule are detected on the expressions in <code>#if</code> directives.</p> <p>Allowed exception on associatively (<code>a && b && c</code>), (<code>a b c</code>).</p>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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 !).	<ul style="list-style-type: none"> • Operand of '!' logical operator should be effectively Boolean. • Left operand of '%s' logical operator should be effectively Boolean. • Right operand of '%s' logical operator should be effectively Boolean. • %s operand of '%s' is effectively Boolean. Boolean should not be used as operands to operators other than '&&', ' ', '!', '=', '==', '!=', and '?:'. 	<p>The operand of a logical operator should be a Boolean data type. Although the C standard does not explicitly define the Boolean data type, the standard implicitly assumes the use of the Boolean data type.</p> <p>Some operators may return Boolean-like expressions, for example, (var == 0).</p> <p>Consider the following code:</p> <pre>unsigned char flag; if (!flag)</pre> <p>The rule checker reports a violation of rule 12.6:</p> <pre>Operand of '!' logical operator should be effectively Boolean.</pre> <p>The operand flag is not a Boolean but an unsigned char.</p> <p>To be compliant with rule 12.6, the code must be rewritten either as</p> <pre>if (!(flag != 0)) or if (flag == 0)</pre> <p>The use of the option <code>-boolean-types</code> may increase or decrease the number of warnings generated.</p>
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed	<ul style="list-style-type: none"> • [~/Left Shift/Right shift/&] operator applied on an expression whose underlying type is signed. • Bitwise ~ on operand of signed underlying type XX. • Bitwise [<< >>] on left hand operand of signed underlying type XX. • Bitwise [& ^] on two operands of s 	<p>The underlying type for an integer is signed when:</p> <ul style="list-style-type: none"> • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number

N.	MISRA Definition	Messages in report file	Polyspace Implementation
12.8	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.	<ul style="list-style-type: none"> • shift amount is negative • shift amount is bigger than 64 • Bitwise [<< >>] count out of range [0 ..X] (width of the underlying type XX of the left hand operand - 1).. 	<p>The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63</p> <p>Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression</p>
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	<ul style="list-style-type: none"> • Unary - on operand of unsigned underlying type XX. • Minus operator applied to an expression whose underlying type is unsigned 	<p>The underlying type for an integer is signed when:</p> <ul style="list-style-type: none"> • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
12.10	The comma operator shall not be used.	The comma operator shall not be used.	
12.11	Evaluation of constant unsigned expression should not lead to wraparound.	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	
12.12	The underlying bit representations of floating-point values shall not be used.	The underlying bit representations of floating-point values shall not be used.	<p>Warning when:</p> <ul style="list-style-type: none"> • A float pointer is cast as a pointer to another data type. Casting a float pointer as a pointer to void does not generate a warning. • A float is packed with another data type. For example: <pre>union { float f; int i; } ...</pre>
12.13	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	Warning when ++ or -- operators are not used alone.

Control Statement Expressions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
13.1	Assignment operators shall not be used in expressions that yield Boolean values.	Assignment operators shall not be used in expressions that yield Boolean values.	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	No warning is given on integer constants. Example: if (2) The use of the option <code>-boolean-types</code> may increase or decrease the number of warnings generated.
13.3	Floating-point expressions shall not be tested for equality or inequality.	Floating-point expressions shall not be tested for equality or inequality.	Warning on direct tests only.
13.4	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	The controlling expression of a <i>for</i> statement shall not contain any objects of floating type	If <i>for</i> index is a variable symbol, checked that it is not a float.
13.5	The three expressions of a <i>for</i> statement shall be concerned only with loop control	<ul style="list-style-type: none"> • 1st expression should be an assignment. • Bad type for loop counter (XX). • 2nd expression should be a comparison. • 2nd expression should be a comparison with loop counter (XX). • 3rd expression should be an assignment of loop counter (XX). • 3rd expression: assigned variable should be the loop counter (XX). • The following kinds of <i>for</i> loops are allowed: <ul style="list-style-type: none"> (a) all three expressions shall be present; (b) the 2nd and 3rd expressions shall be present with prior initialization of the loop counter; (c) all three expressions shall be empty for a deliberate infinite loop. 	Checked if the <i>for</i> loop index (V) is a variable symbol; checked if V is the last assigned variable in the first expression (if present). Checked if, in first expression, if present, is assignment of V; checked if in 2nd expression, if present, must be a comparison of V; Checked if in 3rd expression, if present, must be an assignment of V.
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.	Numeric variables being used within a <i>for</i> loop for iteration counting should not be modified in the body of the loop.	Detect only direct assignments if the <i>for</i> loop index is known and if it is a variable symbol.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
13.7	Boolean operations whose results are invariant shall not be permitted	<ul style="list-style-type: none"> • Boolean operations whose results are invariant shall not be permitted. Expression is always true. • Boolean operations whose results are invariant shall not be permitted. Expression is always false. • Boolean operations whose results are invariant shall not be permitted. 	<p>During compilation, check comparisons with at least one constant operand.</p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <ul style="list-style-type: none"> • Bug Finder flags some violations of this rule through the <code>Dead code</code> and <code>Useless if</code> checkers. • Code Prover does not use gray code to flag violations of this rule. <p>In Code Prover, you can also see a difference in results based on your choice for the option <code>Verification level (-to)</code>. See the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server for more on analysis options and how to check for coding standard violations...</p> <p>The rule violation appears when you check whether an <code>enum</code> variable value lies between its lower and upper bound. The violation appears even if you increment or decrement the variable outside its bounds, for instance, in this <code>for</code> loop condition:</p> <pre>enum ec {RED, BLUE, GREEN} col; for(col=RED; col<=GREEN; col++) {}</pre> <p>An <code>enum</code> variable can potentially wrap around when incremented outside its range and the loop condition can be always true. To avoid the rule violation, you can cast the <code>enum</code> to an integer before the comparison, for instance:</p>

N.	MISRA Definition	Messages in report file	Polyspace Implementation
			<pre>enum ec {RED, BLUE, GREEN} col; for(col=RED; (int)col<=GREEN; col++) {}</pre>

Control Flow

N.	MISRA Definition	Messages in report file	Polyspace Implementation
14.1	There shall be no unreachable code.	There shall be no unreachable code.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
14.2	All non-null statements shall either have at least one side effect however executed, or cause control flow to change	<p>All non-null statements shall either:</p> <ul style="list-style-type: none"> • have at least one side effect however executed, or • cause control flow to change 	
14.3	Before preprocessing, a null statement shall 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.	A null statement shall appear on a line by itself	<p>We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when:</p> <ul style="list-style-type: none"> • there are some comments before it on the same line. • there is a comment immediately after it • there is something else than a comment after the ';' on the same line.
14.4	The <i>goto</i> statement shall not be used.	The goto statement shall not be used.	
14.5	The <i>continue</i> statement shall not be used.	The continue statement shall not be used.	
14.6	For any iteration statement there shall be at most one <i>break</i> statement used for loop termination	For any iteration statement there shall be at most one break statement used for loop termination	
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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	<ul style="list-style-type: none">• The body of a do while statement shall be a compound statement.• The body of a for statement shall be a compound statement.• The body of a switch statement shall be a compound statement	
14.9	An <i>if (expression)</i> construct shall be followed by a compound statement. The <i>else</i> keyword shall be followed by either a compound statement, or another <i>if</i> statement	<ul style="list-style-type: none">• An if (expression) construct shall be followed by a compound statement.• The else keyword shall be followed by either a compound statement, or another if statement	
14.10	All <i>if else if</i> constructs should contain a final <i>else</i> clause.	All if else if constructs should contain a final else clause.	

Switch Statements

N.	MISRA Definition	Messages in report file	Polyspace Implementation
15.0	The MISRA C switch syntax shall be used.	switch statements syntax normative restrictions.	<p>Warning on declarations or any statements before the first switch case.</p> <p>Warning on label or jump statements in the body of switch cases.</p> <p>On the following example, the rule is displayed in the log file at line 3:</p> <pre> 1 ... 2 switch(index) { 3 var = var + 1; // RULE 15.0 // violated 4case 1: ... </pre> <p>The code between switch statement and first case is checked as dead code by Polyspace. It follows ANSI standard behavior.</p> <p>This rule is not considered as a required rule in the MISRA C:2004 rules for generated code. In generated code, if you find a violation of rule 15.0 that does not simultaneously violate a later rule in this group, justify the violation with appropriate comments.</p>
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a <i>switch</i> statement	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	
15.2	An unconditional <i>break</i> statement shall terminate every non-empty switch clause	An unconditional break statement shall terminate every non-empty switch clause	Warning for each non-compliant case clause.
15.3	The final clause of a <i>switch</i> statement shall be the <i>default</i> clause	The final clause of a switch statement shall be the default clause	
15.4	A <i>switch</i> expression should not represent a value that is effectively Boolean	A switch expression should not represent a value that is effectively Boolean	The use of the option <code>-boolean-types</code> may increase the number of warnings generated.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
15.5	Every <i>switch</i> statement shall have at least one <i>case</i> clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
16.1	Functions shall not be defined with variable numbers of arguments.	Function XX should not be defined as varargs.	
16.2	Functions shall not call themselves, either directly or indirectly.	Function %s should not call itself.	The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported. You can calculate the total number of recursion cycles using the code complexity metric <code>Number of Recursions</code> .
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.	Identifiers shall be given for all of the parameters in a function prototype declaration.	Assumes Rule 8.6 is not violated.
16.4	The identifiers used in the declaration and definition of a function shall be identical.	The identifiers used in the declaration and definition of a function shall be identical.	Assumes that rules 8.8 , 8.1 and 16.3 are not violated. All occurrences are detected.
16.5	Functions with no parameters shall be declared with parameter type <i>void</i> .	Functions with no parameters shall be declared with parameter type <i>void</i> .	Definitions are also checked.
16.6	The number of arguments passed to a function shall match the number of parameters.	<ul style="list-style-type: none"> • Too many arguments to XX. • Insufficient number of arguments to XX. 	Assumes that rule 8.1 is not violated. This rule maps to ISO/IEC TS 17961 ID argcomp .
16.7	A pointer parameter in a function prototype should be declared as <i>pointer</i> to <i>const</i> if the pointer is not used to modify the addressed object.	Pointer parameter in a function prototype should be declared as <i>pointer</i> to <i>const</i> if the pointer is not used to modify the addressed object.	Warning if a non- <i>const</i> pointer parameter is either not used to modify the addressed object or is passed to a call of a function that is declared with a <i>const</i> pointer parameter.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Missing return value for non-void function XX.	Warning when a non-void function is not terminated with an unconditional return with an expression.

N.	MISRA Definition	Messages in report file	Polyspace Implementation
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.	Function identifier XX should be preceded by a & or followed by a parameter list.	
16.10	If a function returns error information, then that error information shall be tested.	If a function returns error information, then that error information shall be tested.	<p>The checker flags functions with non-void return if the return value is not used or not explicitly cast to a void type.</p> <p>The checker does not flag the functions memcopy, memset, memmove, strcpy, strncpy, strcat, strncat because these functions simply return a pointer to their first arguments.</p>

Pointers and Arrays

N.	MISRA Definition	Messages in report file	Polyspace Implementation
17.1	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Pointer arithmetic shall only be applied to pointers that address an array or array element.	
17.2	Pointer subtraction shall only be applied to pointers that address elements of the same array	Pointer subtraction shall only be applied to pointers that address elements of the same array.	
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	>, >=, <, <= 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.	Array indexing shall be the only allowed form of pointer arithmetic.	<p>Warning on:</p> <ul style="list-style-type: none"> • Operations on pointers. ($p+I$, $I+p$, and $p-I$, where p is a pointer and I an integer). • Array indexing on nonarray pointers.
17.5	A type should not contain more than 2 levels of pointer indirection	A type should not contain 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.	Pointer to a parameter is an illegal return value. Pointer to a local is an illegal return value.	<p>Warning when assigning address to a global variable, returning a local variable address, or returning a parameter address.</p> <p>This rule maps to ISO/IEC TS 17961 ID accfree.</p>

Structures and Unions

N.	MISRA Definition	Messages in report file	Polyspace Implementation
18.1	All structure or union types shall be complete at the end of a translation unit.	All structure or union types shall be complete at the end of a translation unit.	Warning for all incomplete declarations of structs or unions.
18.2	An object shall not be assigned to an overlapping object.	<ul style="list-style-type: none"> An object shall not be assigned to an overlapping object. Destination and source of XX overlap, the behavior is undefined. 	
18.4	Unions shall not be used	Unions shall not be used.	

Preprocessing Directives

N.	MISRA Definition	Messages in report file	Polyspace Implementation
19.1	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments	<code>#include</code> statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a <code>#include</code> directive is preceded by other things than preprocessor directives, comments, spaces or "new lines".
19.2	Nonstandard characters should not occur in header file names in <code>#include</code> directives	<ul style="list-style-type: none"> A message is displayed on characters ', " or /* between < and > in <code>#include <filename></code> A message is displayed on characters ', or /* between " and " in <code>#include "filename"</code> 	
19.3	The <code>#include</code> directive shall be followed by either a <code><filename></code> or "filename" sequence.	<ul style="list-style-type: none"> '<code>#include</code>' expects "FILENAME" or <code><FILENAME></code> '<code>#include_next</code>' expects "FILENAME" or <code><FILENAME></code> 	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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.	Macro '<name>' does not expand to a compliant construct.	<p>We assume that a macro definition does not violate this rule when it expands to:</p> <ul style="list-style-type: none"> • a braced construct (not necessarily an initializer) • a parenthesized construct (not necessarily an expression) • a number • a character constant • a string constant (can be the result of the concatenation of string field arguments and literal strings) • the following keywords: typedef, extern, static, auto, register, const, volatile, __asm__ and __inline__ • a do-while-zero construct
19.5	Macros shall not be #defined and #undef'd within a block.	<ul style="list-style-type: none"> • Macros shall not be #define'd within a block. • Macros shall not be #undef'd within a block. 	
19.6	#undef shall not be used.	#undef shall not be used.	
19.7	A function should be used in preference to a function like-macro.	A function should be used in preference to a function like-macro	Message on all function-like macro definitions.
19.8	A function-like macro shall not be invoked without all of its arguments	<ul style="list-style-type: none"> • arguments given to macro '<name>' • macro '<name>' used without args. • macro '<name>' used with just one arg. • macro '<name>' used with too many (<number>) args. 	
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Macro argument shall not look like a preprocessing directive.	This rule is detected as violated when the '#' character appears in a macro argument (outside a string or character constant)

N.	MISRA Definition	Messages in report file	Polyspace Implementation
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 ##.	Parameter instance shall be enclosed in parentheses.	<p>If x is a macro parameter, the following instances of x as an operand of the # and ## operators do not generate a warning: #x, ##x, and x##. Otherwise, parentheses are required around x.</p> <p>The software does not generate a warning if a parameter is reused as an argument of a function or function-like macro. For example, consider a parameter x. The software does not generate a warning if x appears as (x) or (x, or ,x) or ,x,.</p>
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	'<name>' is not defined.	
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	More than one occurrence of the # or ## preprocessor operators.	
19.13	The # and ## preprocessor operators should not be used	Message on definitions of macros using # or ## operators	
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	'defined' without an identifier.	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Precautions shall be taken in order to prevent multiple inclusions.	When a header file is formatted as, <pre>#ifndef <control macro> #define <control macro> <contents> #endif</pre> or, <pre>#ifndef <control macro> #error ... #else #define <control macro> <contents> #endif</pre> it is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
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.	<ul style="list-style-type: none"> • <code>'#elif'</code> not within a conditional. • <code>'#else'</code> not within a conditional. • <code>'#elif'</code> not within a conditional. • <code>'#endif'</code> not within a conditional. • unbalanced <code>'#endif'</code>. • unterminated <code>'#if'</code> conditional. • unterminated <code>'#ifdef'</code> conditional. • unterminated <code>'#ifndef'</code> conditional. 	

Standard Libraries

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be redefined. • The macro '<code><name></code>' shall not be undefined. 	

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.2	The names of standard library macros, objects and functions shall not be reused.	Identifier XX should not be used.	<p>In case a macro whose name corresponds to a standard library macro, object or function is defined, the rule that is detected as violated is 20.1.</p> <p>Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that:</p> <ul style="list-style-type: none"> • Do not have initializers. • Do not have storage class specifiers, or have the <code>static</code> specifier
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	<p>Warning for argument in library function call if the following are all true:</p> <ul style="list-style-type: none"> • Argument is a local variable • Local variable is not tested between last assignment and call to the library function • Library function is a common mathematical function • Corresponding parameter of the library function has a restricted input domain. <p>The library function can be one of the following : <code>sqrt</code>, <code>tan</code>, <code>pow</code>, <code>log</code>, <code>log10</code>, <code>fmod</code>, <code>acos</code>, <code>asin</code>, <code>acosh</code>, <code>atanh</code>, or <code>atan2</code>.</p> <p>Bug Finder and Code Prover check this rule differently. The analysis can produce different results.</p>
20.4	Dynamic heap memory allocation shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule 20.2 is not violated.
20.5	The error indicator <code>errno</code> shall not be used	The error indicator <code>errno</code> shall not be used	Assumes that rule 20.2 is not violated

N.	MISRA Definition	Messages in report file	Polyspace Implementation
20.6	The macro <i>offsetof</i> , in library <code><stddef.h></code> , shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	Assumes that rule 20.2 is not violated
20.7	The <i>setjmp</i> macro and the <i>longjmp</i> function shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the <i>longjmp</i> function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.8	The signal handling facilities of <code><signal.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.9	The input/output library <code><stdio.h></code> shall not be used in production code.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the input/output library functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.10	The library functions <i>atof</i> , <i>atoi</i> and <i>atoll</i> from library <code><stdlib.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the <i>atof</i> , <i>atoi</i> and <i>atoll</i> functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.11	The library functions <i>abort</i> , <i>exit</i> , <i>getenv</i> and <i>system</i> from library <code><stdlib.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the <i>abort</i> , <i>exit</i> , <i>getenv</i> and <i>system</i> functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated
20.12	The time handling functions of library <code><time.h></code> shall not be used.	<ul style="list-style-type: none"> • The macro '<code><name></code>' shall not be used. • Identifier XX should not be used. 	In case the time handling functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule 20.2 is not violated

Runtime Failures

N.	MISRA Definition	Messages in report file	Polyspace Implementation
21.1	<p>Minimization of runtime failures shall be ensured by the use of at least one of:</p> <ul style="list-style-type: none"> • static verification tools/ techniques; • dynamic verification tools/ techniques; • explicit coding of checks to handle runtime faults. 		<p>Done by Polyspace. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p> <p>In Code Prover, you can also see a difference in results based on your choice for the option <code>Verification level (-to)</code>. See the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server for more on analysis options and how to check for coding standard violations...</p>

Unsupported MISRA C:2004 and MISRA AC AGC Rules

The Polyspace coding rules checker does not check the following MISRA C:2004 coding rules. These rules cannot be enforced because they are outside the scope of Polyspace software. They may concern documentation, dynamic aspects, or functional aspects of MISRA rules. The **Additional Information** column describes the reason each rule is not checked.

Environment

Rule	Description	Additional Information
1.2 (Required)	No reliance shall be placed on undefined or unspecified behavior	Not statically checkable unless the data dynamic properties is taken into account
1.3 (Required)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compiler/assemblers conform.	It is a process rule method.
1.4 (Required)	The compiler/linker/Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	To observe this rule, check your compiler documentation.
1.5 (Advisory)	Floating point implementations should comply with a defined floating point standard.	To observe this rule, check your compiler documentation.

Documentation

Rule	Description	Additional Information
3.1 (Required)	All usage of implementation-defined behavior shall be documented.	To observe this rule, check your compiler documentation. Error detection is based on undefined behavior, according to choices made for implementation- defined constructions.
3.2 (Required)	The character set and the corresponding encoding shall be documented.	To observe this rule, check your compiler documentation.
3.3 (Advisory)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	To observe this rule, check your compiler documentation.
3.5 (Required)	The implementation-defined behavior and packing of bitfields shall be documented if being relied upon.	To observe this rule, check your compiler documentation.
3.6 (Required)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	To observe this rule, check your compiler documentation.

Structures and Unions

Rule	Description	Additional Information
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Polyspace MISRA C:2012 Checkers

The Polyspace MISRA C:2012 checker helps you to comply with the MISRA C 2012 coding standard.²

When MISRA C:2012 guidelines are violated, the Polyspace MISRA C:2012 checker provides messages with information about the violated rule or directive. Most violations are found during the compile phase of an analysis.

Polyspace Bug Finder can check all the MISRA C:2012 rules and most MISRA C:2012 directives. Polyspace Code Prover does not support checking of the following:

- MISRA C:2012 Dir 4.4, Dir 4.7, 4.13 and 4.14
- MISRA C:2012 Rule 21.13, 21.14, and 21.17 - 21.20
- MISRA C:2012 Rule 22.1 - 22.4 and 22.6 - 22.10

Each guideline is categorized into one of these three categories: mandatory, required, or advisory. When you set up rule checking, you can select subsets of these categories to check. For automatically generated code, some rules change categories, including to one additional category: readability. The `Use generated code requirements (-misra3-agc-mode)` option activates the categorization for automatically generated code. For more on analysis options, see the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server.

There are additional subsets of MISRA C:2012 guidelines defined by Polyspace called Software Quality Objectives (SQQ) that can have a direct or indirect impact on the precision of your results. When you set up checking, you can select these subsets. These subsets are defined in “Software Quality Objective Subsets (C:2012)” on page 1-50.

See Also

See Also

More About

- “MISRA C:2012 Directives and Rules”

2. MISRA and MISRA C are registered trademarks of MIRA Ltd., held on behalf of the MISRA Consortium.

Essential Types in MISRA C:2012 Rules 10.x

MISRA C:2012 rules 10.x classify data types in categories. The rules treat data types in the same category as essentially similar.

For instance, the data types `float`, `double` and `long double` are considered as essentially floating. Rule 10.1 states that the `%` operation must not have essentially floating operands. This statement implies that the operands cannot have one of these three data types: `float`, `double` and `long double`.

Categories of Essential Types

The essential types fall in these categories:

Essential type category	Standard types
Essentially Boolean	<code>bool</code> or <code>_Bool</code> (defined in <code>stdbool.h</code>) If you define a boolean type through a <code>typedef</code> , you must specify this type name before coding rules checking. For more information, see Effective boolean types (-boolean-types) . For more on analysis options, see the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server .
Essentially character	<code>char</code>
Essentially enum	named enum
Essentially signed	<code>signed char</code> , <code>signed short</code> , <code>signed int</code> , <code>signed long</code> , <code>signed long long</code>
Essentially unsigned	<code>unsigned char</code> , <code>unsigned short</code> , <code>unsigned int</code> , <code>unsigned long</code> , <code>unsigned long long</code>
Essentially floating	<code>float</code> , <code>double</code> , <code>long double</code>

How MISRA C:2012 Uses Essential Types

These rules use essential types in their statements:

- MISRA C:2012 Rule 10.1: Operands shall not be of an inappropriate essential type.

For instance, the right operand of the `<<` or `>>` operator must be essentially unsigned. Otherwise, negative values can cause undefined behavior.
- MISRA C:2012 Rule 10.2: Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.

For instance, the type `char` does not represent numeric values. Do not use a variable of this type in addition and subtraction operations.
- MISRA C:2012 Rule 10.3: The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.

For instance, do not assign a variable of data type `double` to a variable with the narrower data type `float`.

- MISRA C:2012 Rule 10.4: Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.

For instance, do not perform an addition operation with a signed `int` operand, which belongs to the essentially signed category, and an unsigned `int` operand, which belongs to the essentially unsigned category.

- MISRA C:2012 Rule 10.5: The value of an expression should not be cast to an inappropriate essential type.

For instance, do not perform a cast between essentially floating types and essentially character types.

- MISRA C:2012 Rule 10.6: The value of a composite expression shall not be assigned to an object with wider essential type.

For instance, if a multiplication, binary addition or bitwise operation involves unsigned `char` operands, do not assign the result to a variable having the wider type unsigned `int`.

- MISRA C:2012 Rule 10.7: If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.

For instance, if one operand of an addition operation is a composite expression with two unsigned `char` operands, the other operand must not have the wider type unsigned `int`.

See Also

More About

- “MISRA C:2012 Directives and Rules”

Unsupported MISRA C:2012 Guidelines

The Polyspace coding rules checker does not check the following MISRA C:2012 directives. These directives are not checked either in Bug Finder or Code Prover. These directives cannot be enforced because they are outside the scope of Polyspace software. These guidelines concern documentation, dynamic aspects, or functional aspects of MISRA rules.

For the list of supported rules and directives, see “MISRA C:2012 Directives and Rules”.

Number	Category	AGC Category	Definition
Directive 3.1	Required	Required	All code shall be traceable to documented requirements
Directive 4.2	Advisory	Advisory	All usage of assembly language should be documented

See Also

More About

- “MISRA C:2012 Directives and Rules”

Polyspace MISRA C++ Checkers

The Polyspace MISRA C++ checker helps you comply with the MISRA C++:2008 coding standard.³

When MISRA C++ rules are violated, the Polyspace software provides messages with information about why the code violates the rule. Most violations are found during the compile phase of an analysis. The MISRA C++ checker can check 202 of the 230 MISRA C++ coding rules.

There are subsets of MISRA C++ coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your results. When you set up rule checking, you can select these subsets directly. These subsets are defined in “Software Quality Objective Subsets (C++)” on page 1-56.

Note The Polyspace MISRA C++ checker is based on MISRA C++:2008 - “Guidelines for the use of the C++ language in critical systems.”

See Also

More About

- “MISRA C++:2008 Rules”

3. MISRA is a registered trademark of MIRA Ltd., held on behalf of the MISRA Consortium.

Unsupported MISRA C++ Coding Rules

In this section...
“Language Independent Issues” on page 7-43
“General” on page 7-44
“Lexical Conventions” on page 7-44
“Expressions” on page 7-44
“Declarations” on page 7-44
“Classes” on page 7-45
“Templates” on page 7-45
“Exception Handling” on page 7-45
“Library Introduction” on page 7-45

Polyspace does not check the following MISRAC++ coding rules. These rules are not checked either in Bug Finder or Code Prover. Some of these rules cannot be enforced because they are outside the scope of Polyspace software. The rules concern documentation, dynamic aspects, or functional aspects of MISRA rules.

For the list of supported rules, see “MISRA C++:2008 Rules”.

Language Independent Issues

N.	Category	MISRA Definition	Additional Information
0-1-4	Required	A project shall not contain non-volatile POD variables having only one use.	
0-1-6	Required	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	
0-1-8	Required	All functions with void return type shall have external side effects.	
0-3-1	Required	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	
0-3-2	Required	If a function generates error information, then that error information shall be tested.	
0-4-1	Document	Use of scaled-integer or fixed-point arithmetic shall be documented.	To observe this rule, check your compiler documentation.
0-4-2	Document	Use of floating-point arithmetic shall be documented.	To observe this rule, check your compiler documentation.

N.	Category	MISRA Definition	Additional Information
0-4-3	Document	Floating-point implementations shall comply with a defined floating-point standard.	To observe this rule, check your compiler documentation.

General

N.	Category	MISRA Definition	Additional Information
1-0-2	Document	Multiple compilers shall only be used if they have a common, defined interface.	To observe this rule, check your compiler documentation.
1-0-3	Document	The implementation of integer division in the chosen compiler shall be determined and documented.	To observe this rule, check your compiler documentation.

Lexical Conventions

N.	Category	MISRA Definition	Additional Information
2-2-1	Document	The character set and the corresponding encoding shall be documented.	To observe this rule, check your compiler documentation.

Expressions

N.	Category	MISRA Definition	Additional Information
5-0-16	Required	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	
5-17-1	Required	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	

Declarations

N.	Category	MISRA Definition	Additional Information
7-2-1	Required	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	
7-4-1	Document	All usage of assembler shall be documented.	To observe this rule, check your compiler documentation.

Classes

N.	Category	MISRA Definition	Additional Information
9-6-1	Document	When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.	To observe this rule, check your compiler documentation.

Templates

N.	Category	MISRA Definition	Additional Information
14-5-1	Required	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	
14-7-1	Required	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	
14-7-2	Required	For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.	

Exception Handling

N.	Category	MISRA Definition	Additional Information
15-0-1	Document	Exceptions shall only be used for error handling.	To observe this rule, check your compiler documentation.
15-1-1	Required	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	
15-3-1	Required	Exceptions shall be raised only after start-up and before termination of the program.	
15-3-4	Required	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	

Library Introduction

N.	Category	MISRA Definition	Additional Information
17-0-3	Required	The names of standard library functions shall not be overridden.	

N.	Category	MISRA Definition	Additional Information
17-0-4	Required	All library code shall conform to MISRA C++.	To observe this rule, check your compiler documentation.

See Also

More About

- "MISRA C++:2008 Rules"

Polyspace JSF AV C++ Checkers

The Polyspace JSF C++ checker helps you comply with the Joint Strike Fighter® Air Vehicle C++ coding standards (JSF++). These coding standards were developed by Lockheed Martin® for the Joint Strike Fighter program. They are designed to improve the robustness of C++ code, and improve maintainability.

4

When JSF++ rules are violated, the Polyspace JSF C++ checker enables Polyspace software to provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis.

Note The Polyspace JSF C++ checker is based on JSF++:2005.

See Also

4. JSF and Joint Strike Fighter are Lockheed Martin.

JSF AV C++ Coding Rules

Supported JSF C++ Coding Rules

Code Size and Complexity

N.	JSF++ Definition	Polyspace Implementation
1	Any one function (or method) will contain no more than 200 logical source lines of code (L-SLOCs).	Message in report file: <i><function name></i> has <i><num></i> logical source lines of code.
3	All functions shall have a cyclomatic complexity number of 20 or less.	Message in report file: <i><function name></i> has cyclomatic complexity number equal to <i><num></i> .

Environment

N.	JSF++ Definition	Polyspace Implementation
8	All code shall conform to ISO/IEC 14882:2002(E) standard C++.	Reports the compilation error message
9	Only those characters specified in the C++ basic source character set will be used.	
11	Trigraphs will not be used.	
12	The following digraphs will not be used: <%, %>, <:, :>, %:, %:~:.	Message in report file: The following digraph will not be used: <i><digraph></i> . Reports the digraph. If the rule level is set to warning, the digraph will be allowed even if it is not supported in <code>-compiler iso</code> .
13	Multi-byte characters and wide string literals will not be used.	Report L'c', L"string", and use of wchar_t.
14	Literal suffixes shall use uppercase rather than lowercase letters.	
15	Provision shall be made for run-time checking (defensive programming).	Done with checks in the software.

Libraries

N.	JSF++ Definition	Polyspace Implementation
17	The error indicator <code>errno</code> shall not be used.	<code>errno</code> should not be used as a macro or a global with external "C" linkage.
18	The macro <code>offsetof</code> , in library <code><stddef.h></code> , shall not be used.	<code>offsetof</code> should not be used as a macro or a global with external "C" linkage.

N.	JSF++ Definition	Polyspace Implementation
19	<locale.h> and the <code>setlocale</code> function shall not be used.	<code>setlocale</code> and <code>localeconv</code> should not be used as a macro or a global with external "C" linkage.
20	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.	<code>setjmp</code> and <code>longjmp</code> should not be used as a macro or a global with external "C" linkage.
21	The signal handling facilities of <signal.h> shall not be used.	<code>signal</code> and <code>raise</code> should not be used as a macro or a global with external "C" linkage.
22	The input/output library <stdio.h> shall not be used.	all standard functions of <stdio.h> should not be used as a macro or a global with external "C" linkage.
23	The library functions <code>atof</code> , <code>atoi</code> and <code>atol</code> from library <stdlib.h> shall not be used.	<code>atof</code> , <code>atoi</code> and <code>atol</code> should not be used as a macro or a global with external "C" linkage.
24	The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <stdlib.h> shall not be used.	<code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> should not be used as a macro or a global with external "C" linkage.
25	The time handling functions of library <time.h> shall not be used.	<code>clock</code> , <code>difftime</code> , <code>mktime</code> , <code>asctime</code> , <code>ctime</code> , <code>gmtime</code> , <code>localtime</code> and <code>strftime</code> should not be used as a macro or a global with external "C" linkage.

Pre-Processing Directives

N.	JSF++ Definition	Polyspace Implementation
26	Only the following preprocessor directives shall be used: <code>#ifndef</code> , <code>#define</code> , <code>#endif</code> , <code>#include</code> .	
27	<code>#ifndef</code> , <code>#define</code> and <code>#endif</code> will be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files will not be used.	Detects the patterns <code>#if !defined</code> , <code>#pragma once</code> , <code>#ifdef</code> , and missing <code>#define</code> .
28	The <code>#ifndef</code> and <code>#endif</code> preprocessor directives will only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	Detects any use that does not comply with AV Rule 27. Assuming 35/27 is not violated, reports only <code>#ifndef</code> .
29	The <code>#define</code> preprocessor directive shall not be used to create inline macros. Inline functions shall be used instead.	Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use). Messages in report file: <ul style="list-style-type: none"> • 29.1 : The <code>#define</code> preprocessor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros.

N.	JSF++ Definition	Polyspace Implementation
30	The <code>#define</code> preprocessor directive shall not be used to define constant values. Instead, the <code>const</code> qualifier shall be applied to variable declarations to specify constant values.	Reports <code>#define</code> of simple constants.
31	The <code>#define</code> preprocessor directive will only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of <code>#define</code> that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The <code>#include</code> preprocessor directive will only be used to include header (*.h) files.	

Header Files

N.	JSF++ Definition	Polyspace Implementation
33	The <code>#include</code> directive shall use the <code><filename.h></code> notation to include header files.	
35	A header file will contain a mechanism that prevents multiple inclusions of itself.	
39	Header files (*.h) will not contain non-const variable definitions or function definitions.	Reports definitions of global variables / function in header.

Style

N.	JSF++ Definition	Polyspace Implementation
40	Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.	Reports when type, template, or inline function is defined in source file. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
41	Source lines will be kept to a length of 120 characters or less.	
42	Each expression-statement will be on a separate line.	Reports when two consecutive expression statements are on the same line (unless the statements are part of a macro definition).
43	Tabs should be avoided.	
44	All indentations will be at least two spaces and be consistent within the same source file.	Reports when a statement indentation is not at least two spaces more than the statement containing it. Does not report bad indentation between opening braces following if/else, do/while, for, and while statements. NB: in final release it will accept any indentation
46	User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.	<i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" on page 5-80.</i>

N.	JSF++ Definition	Polyspace Implementation
47	Identifiers will not begin with the underscore character '_'.	
48	Identifiers will not differ by: <ul style="list-style-type: none"> • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O'; with the number '0' or the letter 'D' • The interchange of the letter 'I'; with the number '1' or the letter 'l' • The interchange of the letter 'S' with the number '5' • The interchange of the letter 'Z' with the number 2 • The interchange of the letter 'n' with the letter 'h' 	Checked regardless of scope. Not checked between macros and other identifiers. Messages in report file: <ul style="list-style-type: none"> • Identifier <i>Idf1 (file1.cpp line l1 column c1)</i> and <i>Idf2 (file2.cpp line l2 column c2)</i> only differ by the presence/absence of the underscore character. • Identifier <i>Idf1 (file1.cpp line l1 column c1)</i> and <i>Idf2 (file2.cpp line l2 column c2)</i> only differ by a mixture of case. • Identifier <i>Idf1 (file1.cpp line l1 column c1)</i> and <i>Idf2 (file2.cpp line l2 column c2)</i> only differ by letter 0, with the number 0.
50	The first word of the name of a class, structure, namespace, enumeration, or type created with typedef will begin with an uppercase letter. All others letters will be lowercase.	Messages in report file: <ul style="list-style-type: none"> • The first word of the name of a class will begin with an uppercase letter. • The first word of the namespace of a class will begin with an uppercase letter. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
51	All letters contained in function and variables names will be composed entirely of lowercase letters.	Messages in report file: <ul style="list-style-type: none"> • All letters contained in variable names will be composed entirely of lowercase letters. • All letters contained in function names will be composed entirely of lowercase letters.
52	Identifiers for constant and enumerator values shall be lowercase.	Messages in report file: <ul style="list-style-type: none"> • Identifier for enumerator value shall be lowercase. • Identifier for template constant parameter shall be lowercase.
53	Header files will always have file name extension of ".h".	.H is allowed if you set the option -dos.
53.1	The following character sequences shall not appear in header file names: ', \, /*, //, or ".	
54	Implementation files will always have a file name extension of ".cpp".	Not case sensitive if you set the option -dos.

N.	JSF++ Definition	Polyspace Implementation
57	The public, protected, and private sections of a class will be declared in that order.	
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument will be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	Detects that two parameters are not on the same line, The first parameter should be on the same line as function name. Does not check for the closing parenthesis.
59	The statements forming the body of an if, else if, else, while, do ... while or for statement shall always be enclosed in braces, even if the braces form an empty block.	<p>Messages in report file:</p> <ul style="list-style-type: none"> • The statements forming the body of an if statement shall always be enclosed in braces. • The statements forming the body of an else statement shall always be enclosed in braces. • The statements forming the body of a while statement shall always be enclosed in braces. • The statements forming the body of a do ... while statement shall always be enclosed in braces. • The statements forming the body of a for statement shall always be enclosed in braces.
60	Braces ("{}") which enclose a block will be placed in the same column, on separate lines directly before and after the block.	Detects that statement-block braces should be in the same columns.
61	Braces ("{}") which enclose a block will have nothing else on the line except comments.	
62	The dereference operator '*' and the address-of operator '&' will be directly connected with the type-specifier.	Reports when there is a space between type and "*" "&" for variables, parameters and fields declaration.
63	Spaces will not be used around '.' or '->', nor between unary operators and operands.	<p>Reports when the following characters are not directly connected to a white space:</p> <ul style="list-style-type: none"> • . • -> • ! • ~ • - • ++ • — <p>Note that a violation will be reported for "." used in float/double definition.</p>

Classes

N.	JSF++ Definition	Polyspace Implementation
67	Public and protected data should only be used in structs - not classes.	
68	Unneeded implicitly generated member functions shall be explicitly disallowed.	Reports when default constructor, assignment operator, copy constructor or destructor is not declared.
71.1	A class's virtual functions shall not be invoked from its destructor or any of its constructors.	Reports when a constructor or destructor directly calls a virtual function.
74	Initialization of nonstatic class members will be performed through the member initialization list rather than through assignment in the body of a constructor.	All data should be initialized in the initialization list except for array. Does not report that an assignment exists in ctor body. Message in report file: Initialization of nonstatic class members "<field>" will be performed through the member initialization list.
75	Members of the initialization list shall be listed in the order in which they are declared in the class.	
76	A copy constructor and an assignment operator shall be declared for classes that contain pointers to data items or nontrivial destructors.	Messages in report file: <ul style="list-style-type: none"> • no copy constructor and no copy assign • no copy constructor • no copy assign Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
77.1	The definition of a member function shall not contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	Does not report when an explicit copy constructor exists.
78	All base classes with a virtual function shall define a virtual destructor.	
79	All resources acquired by a class shall be released by the class's destructor.	Reports when the number of "new" called in a constructor is greater than the number of "delete" called in its destructor. Note A violation is raised even if "new" is done in a "if/else".

N.	JSF++ Definition	Polyspace Implementation
81	The assignment operator shall handle self-assignment correctly	<p>Reports when copy assignment body does not begin with "if (this != arg)"</p> <p>A violation is not raised if an empty else statement follows the if, or the body contains only a return statement.</p> <p>A violation is raised when the if statement is followed by a statement other than the return statement.</p>
82	An assignment operator shall return a reference to <code>*this</code> .	<p>The following operators should return <code>*this</code> on method, and <code>*first_arg</code> on plain function:</p> <ul style="list-style-type: none"> • operator= • operator+= • operator-= • operator*= • operator >>= • operator <<= • operator /= • operator %= • operator = • operator &= • operator ^= • Prefix operator++ • Prefix operator-- <p>Does not report when no return exists.</p> <p>No special message if type does not match.</p> <p>Messages in report file:</p> <ul style="list-style-type: none"> • An assignment operator shall return a reference to <code>*this</code>. • An assignment operator shall return a reference to its first arg.
83	An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	Reports when a copy assignment does not assign all data members. In a derived class, it also reports when a copy assignment does not call inherited copy assignments.

N.	JSF++ Definition	Polyspace Implementation
88	Multiple inheritance shall only be allowed in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation.	Messages in report file: <ul style="list-style-type: none"> Multiple inheritance on public implementation shall not be allowed: <code><public_base_class></code> is not an interface. Multiple inheritance on protected implementation shall not be allowed : <code><protected_base_class_1></code>. <code><protected_base_class_2></code> are not interfaces.
88.1	A stateful virtual base shall be explicitly declared in each derived class that accesses it.	
89	A base class shall not be both virtual and nonvirtual in the same hierarchy.	
94	An inherited nonvirtual function shall not be redefined in a derived class.	Does not report for destructor. Message in report file: Inherited nonvirtual function %s shall not be redefined in a derived class.
95	An inherited default parameter shall never be redefined.	
96	Arrays shall not be treated polymorphically.	Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.
97	Arrays shall not be used in interface.	Only to prevent array-to-pointer-decay. Not checked on private methods
97.1	Neither operand of an equality operator (== or !=) shall be a pointer to a virtual member function.	Reports == and != on pointer to member function of polymorphic classes (cannot determine statically if it is virtual or not), except when one argument is the null constant.

Namespaces

N.	JSF++ Definition	Polyspace Implementation
98	Every nonlocal name, except <code>main()</code> , should be placed in some namespace.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
99	Namespaces will not be nested more than two levels deep.	

Templates

N.	JSF++ Definition	Polyspace Implementation
104	A template specialization shall be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Polyspace Implementation
107	Functions shall always be declared at file scope.	
108	Functions with variable numbers of arguments shall not be used.	
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	Reports when "inline" is not in the definition of a member function inside the class definition.
110	Functions with more than 7 arguments will not be used.	
111	A function shall not return a pointer or reference to a non-static local object.	Simple cases without alias effect detected.
113	Functions will have a single exit point.	Reports first return, or once per function.
114	All exit points of value-returning functions shall be through return statements.	
116	Small, concrete-type arguments (two or three words in size) should be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with <code>sizeof <= 2 * sizeof(int)</code> . Does not report for copy-constructor.
117	<p>Arguments should be passed by reference if NULL values are not possible:</p> <ul style="list-style-type: none"> • 117.1: An object should be passed as <code>const T&</code> if the function should not change the value of the object. • 117.2: An object should be passed as <code>T&</code> if the function may change the value of the object. 	<p>The checker flags a parameter passed by pointer if the parameter is not compared against <code>NULL</code> or <code>nullptr</code> in the function body. The absence of a check for null indicates that the parameter cannot be null and therefore can be passed by reference.</p> <p>The checker does not raise a violation:</p> <ul style="list-style-type: none"> • If a parameter is passed using a smart pointer. Only raw pointers are considered. • If the pointer parameter is not dereferenced within the function.
119	Functions shall not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	<p>The checker reports each function that calls itself, directly or indirectly. Even if several functions are involved in one recursion cycle, each function is individually reported.</p> <p>You can calculate the total number of recursion cycles using the code complexity metric <code>Number of Recursions</code>. Note that unlike the checker, the metric also considers implicit calls, for instance, to compiler-generated constructors during object creation.</p>
121	Only functions with 1 or 2 statements should be considered candidates for inline functions.	Reports inline functions with more than 2 statements.

N.	JSF++ Definition	Polyspace Implementation
122	Trivial accessor and mutator functions should be inlined.	<p>The checker uses the following criteria to determine if a method is trivial:</p> <ul style="list-style-type: none"> An accessor method is trivial if it has no parameters and contains one <code>return</code> statement that returns a non-static data member or a reference to a non-static data member. <p>The return type of the method must exactly match or be a reference to the type of the data member.</p> <ul style="list-style-type: none"> A mutator method is trivial if it has a <code>void</code> return type, one parameter and contains one assignment statement that assigns the parameter to a non-static data member. <p>The parameter type must exactly match or be a reference to the type of the data member.</p> <p>The checker reports trivial accessor and mutator methods defined outside their classes without the <code>inline</code> keyword.</p> <p>The checker does not flag template methods or virtual methods.</p>

Comments

N.	JSF++ Definition	Polyspace Implementation
126	Only valid C++ style comments (<code>//</code>) shall be used.	

N.	JSF++ Definition	Polyspace Implementation
127	Code that is not used (commented out) shall be deleted.	<p>The checker uses internal heuristics to detect commented out code. For instance, characters such as #, ;, { or } indicate comments that might potentially contain code. These comments are then evaluated against other metrics to determine the likelihood of code masquerading as comment. For instance, several successive words without a symbol in between reduces this likelihood.</p> <p>The checker does not flag the following comments even if they contain code:</p> <ul style="list-style-type: none"> • Doxygen comments beginning with <code>/**, /*!, /// or //!</code>. • Comments that repeat the same symbol several times, for instance, the symbol = here: <pre data-bbox="906 829 1442 913"> // ===== // A comment // =====*/ </pre> • Comments on the first line of a file. • Comments that mix the C style (<code>/* */</code>) and C++ style (<code>//</code>). <p>The checker considers that these comments are meant for documentation purposes or entered deliberately with some forethought.</p>
133	Every source file will be documented with an introductory comment that provides information on the file name, its contents, and any program-required information (e.g. legal statements, copyright information, etc).	<p>Reports when a file does not begin with two comment lines.</p> <p>Note: This rule cannot be annotated in the source code.</p>

Declarations and Definitions

N.	JSF++ Definition	Polyspace Implementation
135	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	JSF++ Definition	Polyspace Implementation
136	Declarations should be at the smallest feasible scope.	<p>Reports when:</p> <ul style="list-style-type: none"> • A global variable is used in only one function. • A local variable is not used in a statement (expr, return, init ...) of the same level of its declaration (in the same block) or is not used in two sub-statements of its declaration. <hr/> <p>Note</p> <ul style="list-style-type: none"> • Non-used variables are reported. • Initializations at definition are ignored (not considered an access)
137	All declarations at file scope should be static where possible.	<p>Starting in R2021a, this checker is raised on declarations of nonstatic objects that you use in only one file. The checker is raised even if you analyze a single file. The checker is not raised on the declarations of objects that remain unused, such as:</p> <ul style="list-style-type: none"> • Noninstantiated templates • Uncalled <code>static</code> or <code>extern</code> functions • Uncalled and undefined local functions • Unused types and variables <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" on page 5-80.</i></p>
138	Identifiers shall not simultaneously have both internal and external linkage in the same translation unit.	
139	External objects will not be declared in more than one file.	<p>Reports all duplicate declarations inside a translation unit. Reports when the declaration localization is not the same in all translation units.</p> <p><i>This checker is deactivated in a default Polyspace as You Code analysis. See "Checkers Deactivated in Polyspace as You Code Default Analysis" on page 5-80.</i></p>
140	The register storage class specifier shall not be used.	
141	A class, structure, or enumeration will not be declared in the definition of its type.	

Initialization

N.	JSF++ Definition	Polyspace Implementation
142	All variables shall be initialized before use.	Done with Non-initialized variable checks in the software.
144	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	This covers partial initialization.
145	In an enumerator list, the '=' construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Generates one report for an enumerator list.

Types

N.	JSF++ Definition	Polyspace Implementation
147	The underlying bit representations of floating point numbers shall not be used in any way by the programmer.	Reports on casts with float pointers (except with void*).
148	Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	Reports when non enumeration types are used in switches.

Constants

N.	JSF++ Definition	Polyspace Implementation
149	Octal constants (other than zero) shall not be used.	
150	Hexadecimal constants will be represented using all uppercase letters.	
151	Numeric values in code will not be used; symbolic values will be used instead.	Reports direct numeric constants (except integer/float value 1, 0) in expressions, non - const initializations. and switch cases. char constants are allowed. Does not report on templates non-type parameter. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
151.1	A string literal shall not be modified.	The rule checker flags assignment of string literals to: <ul style="list-style-type: none"> • Pointers other than pointers to const objects. • Arrays that are not const-qualified.

Variables

N.	JSF++ Definition	Polyspace Implementation
152	Multiple variable declarations shall not be allowed on the same line.	Reports when two consecutive declaration statements are on the same line (unless the statements are part of a macro definition).

Unions and Bit Fields

N.	JSF++ Definition	Polyspace Implementation
153	Unions shall not be used.	
154	Bit-fields shall have explicitly unsigned integral or enumeration types only.	
156	All the members of a structure (or class) shall be named and shall only be accessed via their names.	Reports unnamed bit-fields (unnamed fields are not allowed).

Operators

N.	JSF++ Definition	Polyspace Implementation
157	The right hand operand of a && or operator shall not contain side effects.	Assumes rule 159 is not violated. Messages in report file: <ul style="list-style-type: none"> The right hand operand of a && operator shall not contain side effects. The right hand operand of a operator shall not contain side effects.
158	The operands of a logical && or shall be parenthesized if the operands contain binary operators.	Messages in report file: <ul style="list-style-type: none"> The operands of a logical && shall be parenthesized if the operands contain binary operators. The operands of a logical shall be parenthesized if the operands contain binary operators. Exception for: X Y Z , Z&&Y &&Z
159	Operators , &&, and unary & shall not be overloaded.	Messages in report file: <ul style="list-style-type: none"> Unary operator & shall not be overloaded. Operator shall not be overloaded. Operator && shall not be overloaded.
160	An assignment expression shall be used only as the expression in an expression statement.	Only simple assignment, not +=, ++, etc.
162	Signed and unsigned values shall not be mixed in arithmetic or comparison operations.	
163	Unsigned arithmetic shall not be used.	

N.	JSF++ Definition	Polyspace Implementation
164	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the left-hand operand (inclusive).	
164.1	The left-hand operand of a right-shift operator shall not have a negative value.	Detects constant case +. Found by the software for dynamic cases.
165	The unary minus operator shall not be applied to an unsigned expression.	
166	The sizeof operator will not be used on expressions that contain side effects.	
168	The comma operator shall not be used.	

Pointers and References

N.	JSF++ Definition	Polyspace Implementation
169	Pointers to pointers should be avoided when possible.	Reports second-level pointers, except for arguments of main.
170	More than 2 levels of pointer indirection shall not be used.	Only reports on variables/parameters.
171	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to: <ul style="list-style-type: none"> • the same object, • the same function, • members of the same object, or • elements of the same array (including one past the end of the same array). 	Reports when relational operator are used on pointer types (casts ignored).
173	The address of an object with automatic storage shall not be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer shall not be de-referenced.	Done with checks in software.
175	A pointer shall not be compared to NULL or be assigned NULL; use plain 0 instead.	Reports usage of NULL macro in pointer contexts.
176	A typedef will be used to simplify program syntax when declaring function pointers.	Reports non-typedef function pointers, or pointers to member functions for types of variables, fields, parameters. Returns type of function, cast, and exception specification.

Type Conversions

N.	JSF++ Definition	Polyspace Implementation
177	User-defined conversion functions should be avoided.	<p>Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones).</p> <p>Does not report copy-constructor.</p> <p>Additional message for constructor case:</p> <p>This constructor should be flagged as "explicit".</p>
178	<p>Down casting (casting from base to derived class) shall only be allowed through one of the following mechanism:</p> <ul style="list-style-type: none"> • Virtual functions that act like dynamic casts (most likely useful in relatively simple cases). • Use of the visitor (or similar) pattern (most likely useful in complicated cases). 	Reports explicit down casting, <code>dynamic_cast</code> included. (Visitor patter does not have a special case.)
179	A pointer to a virtual base class shall not be converted to a pointer to a derived class.	Reports this specific down cast. Allows <code>dynamic_cast</code> .
180	Implicit conversions that may result in a loss of information shall not be used.	<p>Reports the following implicit casts :</p> <pre>integer => smaller integer unsigned => smaller or eq signed signed => smaller or eq un-signed integer => float float => integer</pre> <p>Does not report for cast to <code>bool</code> reports for implicit cast on constant done with the option -<code>scalar-overflows-checks signed-and-unsigned</code></p> <p>Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.</p>
181	Redundant explicit casts will not be used.	Reports useless cast: <code>cast T to T</code> . Casts to equivalent <code>typedefs</code> are also reported.
182	Type casting from any type to or from pointers shall not be used.	Does not report when Rule 181 applies.
184	Floating point numbers shall not be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	Reports <code>float->int</code> conversions. Does not report implicit ones.
185	C++ style casts (<code>const_cast</code> , <code>reinterpret_cast</code> , and <code>static_cast</code>) shall be used instead of the traditional C-style casts.	

Flow Control Standards

N.	JSF++ Definition	Polyspace Implementation
186	There shall be no unreachable code.	Done with gray checks in the software. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
187	All non-null statements shall potentially have a side-effect.	
188	Labels will not be used, except in switch statements.	
189	The <code>goto</code> statement shall not be used.	
190	The <code>continue</code> statement shall not be used.	
191	The <code>break</code> statement shall not be used (except to terminate the cases of a switch statement).	
192	All <code>if, else if</code> constructs will contain either a final <code>else</code> clause or a comment indicating why a final <code>else</code> clause is not necessary.	<code>else if</code> should contain an <code>else</code> clause.
193	Every non-empty <code>case</code> clause in a switch statement shall be terminated with a <code>break</code> statement.	
194	All switch statements that do not intend to test for every enumeration value shall contain a final <code>default</code> clause.	Reports only for missing <code>default</code> .
195	A switch expression will not represent a Boolean value.	
196	Every switch statement will have at least two cases and a potential <code>default</code> .	
197	Floating point variables shall not be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a <code>for</code> loop will perform no actions other than to initialize the value of a single <code>for</code> loop parameter.	Reports if loop parameter cannot be determined. Assumes Rule 200 is not violated. The <code>loop variable</code> parameter is assumed to be a variable.
199	The increment expression in a <code>for</code> loop will perform no action other than to change a single loop parameter to the next value for the loop.	Assumes 1 loop parameter (Rule 198), with non class type. Rule 200 must not be violated for this rule to be reported.
200	Null initialize or increment expressions in <code>for</code> loops will not be used; a <code>while</code> loop will be used instead.	
201	Numeric variables being used within a <code>for</code> loop for iteration counting shall not be modified in the body of the loop.	Assumes 1 loop parameter (AV rule 198), and no alias writes.

Expressions

N.	JSF++ Definition	Polyspace Implementation
202	Floating point variables shall not be tested for exact equality or inequality.	Reports only direct equality/inequality. Check done for all expressions.
203	Evaluation of expressions shall not lead to overflow/underflow.	Done with overflow checks in the software.
204	A single operation with side-effects shall only be used in the following contexts: <ul style="list-style-type: none"> • by itself • the right-hand side of an assignment • a condition • the only argument expression with a side-effect in a function call • condition of a loop • switch condition • single part of a chained operation 	Reports when: <ul style="list-style-type: none"> • A side effect is found in a return statement • A side effect exists on a single value, and only one operand of the function call has a side effect.
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	Reports when: <ul style="list-style-type: none"> • Variable is written more than once in an expression • Variable is read and write in sub-expressions • Volatile variable is accessed more than once <hr/> Note Read-write operations such as ++, are only considered as a write.
205	The volatile keyword shall not be used unless directly interfacing with hardware.	Reports if volatile keyword is used.

Memory Allocation

N.	JSF++ Definition	Polyspace Implementation
206	Allocation/deallocation from/to the free store (heap) shall not occur after initialization.	Reports calls to C library functions: malloc / calloc / realloc / free and all new/delete operators in functions or methods.

Fault Handling

N.	JSF++ Definition	Polyspace Implementation
208	C++ exceptions shall not be used.	Reports try, catch, throw spec, and throw.

Portable Code

N.	JSF++ Definition	Polyspace Implementation
209	The basic types of <code>int</code> , <code>short</code> , <code>long</code> , <code>float</code> and <code>double</code> shall not be used, but specific-length equivalents should be <code>typedef</code> 'd accordingly for each compiler, and these type names used in the code.	Only allows use of basic types through direct typedefs.
213	No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	Reports when a binary operation has one operand that is not parenthesized and is an operation with inferior precedence level. Reports bitwise and shifts operators that are used without parenthesis and binary operation arguments.
215	Pointer arithmetic will not be used.	Reports: <code>p + Ip - Ip++p--p+=p-=</code> Allows <code>p[i]</code> .

Unsupported JSF++ Rules

- “Code Size and Complexity” on page 7-67
- “Rules” on page 7-67
- “Environment” on page 7-67
- “Libraries” on page 7-67
- “Header Files” on page 7-67
- “Style” on page 7-68
- “Classes” on page 7-68
- “Namespaces” on page 7-69
- “Templates” on page 7-69
- “Functions” on page 7-69
- “Comments” on page 7-70
- “Initialization” on page 7-70
- “Types” on page 7-70
- “Unions and Bit Fields” on page 7-70
- “Operators” on page 7-70
- “Type Conversions” on page 7-71
- “Expressions” on page 7-71
- “Memory Allocation” on page 7-71
- “Portable Code” on page 7-71
- “Efficiency Considerations” on page 7-71
- “Miscellaneous” on page 7-71
- “Testing” on page 7-72

Code Size and Complexity

N.	JSF++ Definition
2	There shall not be any self-modifying code.

Rules

N.	JSF++ Definition
4	To break a “should” rule, the following approval must be received by the developer: <ul style="list-style-type: none"> approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)
5	To break a “will” or a “shall” rule, the following approvals must be received by the developer: <ul style="list-style-type: none"> approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) approval from the software product manager (obtained by the unit approval in the developmental CM tool)
6	Each deviation from a “shall” rule shall be documented in the file that contains the deviation. Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.
7	Approval will not be required for a deviation from a “shall” or “will” rule that complies with an exception specified by that rule.

Environment

N.	JSF++ Definition
10	Values of character types will be restricted to a defined and documented subset of ISO 10646 1.

Libraries

N.	JSF++ Definition
16	Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.

Header Files

N.	JSF++ Definition
34	Header files should contain logically related declarations only.
36	Compilation dependencies should be minimized when possible.
37	Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file — not the .h file.
38	Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.

Style

N.	JSF++ Definition
45	All words in an identifier will be separated by the ‘_’ character.
49	All acronyms in an identifier will be composed of uppercase letters.
55	The name of a header file should reflect the logical entity for which it provides declarations.
56	<p>The name of an implementation file should reflect the logical entity for which it provides definitions and have a “.cpp” extension (this name will normally be identical to the header file that provides the corresponding declarations.)</p> <p>At times, more than one .cpp file for a given logical entity will be required. In these cases, a suffix should be appended to reflect a logical differentiation.</p>

Classes

N.	JSF++ Definition
64	A class interface should be complete and minimal.
65	A structure should be used to model an entity that does not require an invariant.
66	A class should be used to model an entity that maintains an invariant.
69	A member function that does not affect the state of an object (its instance variables) will be declared const. Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.
70	A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.
70.1	An object shall not be improperly used before its lifetime begins or after its lifetime ends.
71	Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.
72	<p>The invariant for a class should be:</p> <ul style="list-style-type: none"> • A part of the postcondition of every class constructor, • A part of the precondition of the class destructor (if any), • A part of the precondition and postcondition of every other publicly accessible operation.
73	Unnecessary default constructors shall not be defined.
77	A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).
80	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.
84	Operator overloading will be used sparingly and in a conventional manner.
85	When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.
86	Concrete types should be used to represent simple independent concepts.
87	Hierarchies should be based on abstract classes.
90	Heavily used interfaces should be minimal, general and abstract.

N.	JSF++ Definition
91	Public inheritance will be used to implement “is-a” relationships.
92	<p>A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system:</p> <ul style="list-style-type: none"> • Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. • Postconditions of derived methods must be at least as strong as the postconditions of the methods they override. <p>In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.</p>
93	“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	<p>Elements from a namespace should be selected as follows:</p> <ul style="list-style-type: none"> • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	<p>Templates shall be reviewed as follows:</p> <ol style="list-style-type: none"> 1 with respect to the template in isolation considering assumptions or requirements placed on its arguments. 2 with respect to all functions instantiated by actual arguments.
102	Template tests shall be created to cover all actual template instantiations.
103	Constraint checks should be applied to template arguments.
105	A template definition’s dependence on its instantiation contexts should be minimized.
106	Specializations for pointer types should be made where appropriate.

Functions

N.	JSF++ Definition
112	Function return values should not obscure resource ownership.
115	If a function returns error information, then that error information will be tested.
118	<p>Arguments should be passed via pointers if NULL values are possible:</p> <ul style="list-style-type: none"> • 118.1 - An object should be passed as <code>const T*</code> if its value should not be modified. • 118.2 - An object should be passed as <code>T*</code> if its value may be modified.

N.	JSF++ Definition
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.
123	The number of accessor and mutator functions should be minimized.
124	Trivial forwarding functions should be inlined.
125	Unnecessary temporary objects should be avoided.

Comments

N.	JSF++ Definition
128	Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.
129	Comments in header files should describe the externally visible behavior of the functions or classes being documented.
130	The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function's preamble.

Initialization

N.	JSF++ Definition
143	Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Types

N.	JSF++ Definition
146	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE® Std 754 [1].

Unions and Bit Fields

N.	JSF++ Definition
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Operators

N.	JSF++ Definition
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.

Type Conversions

N.	JSF++ Definition
183	Every possible measure should be taken to avoid type casting.

Expressions

N.	JSF++ Definition
204	<p>A single operation with side-effects shall only be used in the following contexts:</p> <ol style="list-style-type: none"> 1 by itself 2 the right-hand side of an assignment 3 a condition 4 the only argument expression with a side-effect in a function call 5 condition of a loop 6 switch condition 7 single part of a chained operation

Memory Allocation

N.	JSF++ Definition
207	Unencapsulated global data will be avoided.

Portable Code

N.	JSF++ Definition
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.).
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
211	Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
212	Underflow or overflow functioning shall not be depended on in any special way.
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

N.	JSF++ Definition
216	Programmers should not attempt to prematurely optimize code.

Miscellaneous

N.	JSF++ Definition
217	Compile-time and link-time errors should be preferred over run-time errors.

N.	JSF++ Definition
218	Compiler warning levels will be set in compliance with project policies.

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

Configure Target and Compiler Options

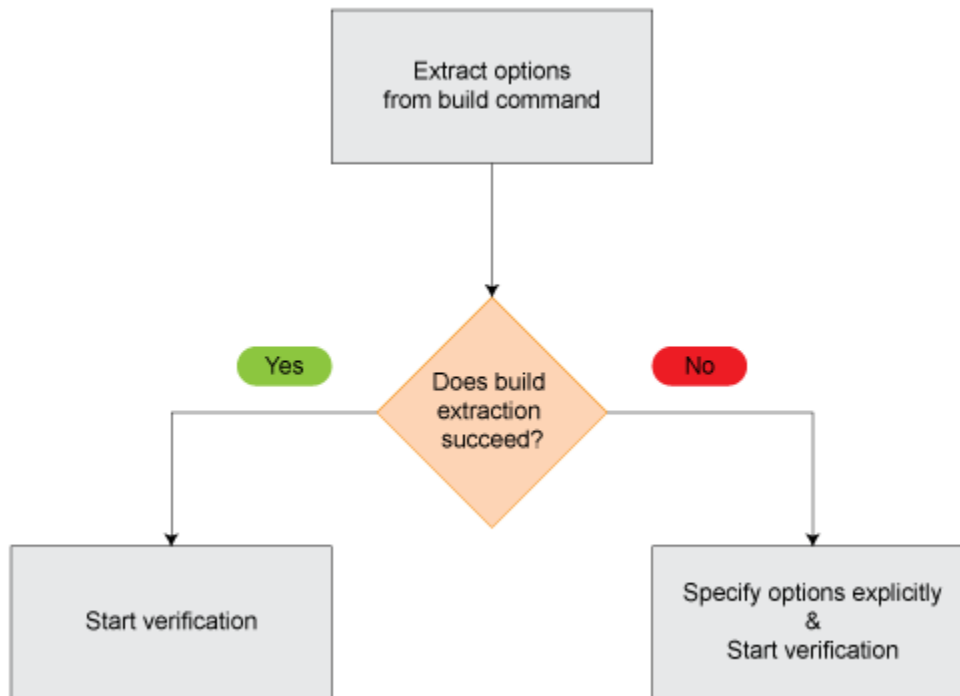
Specify Target Environment and Compiler Behavior

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” (Polyspace Bug Finder Server).

For Polyspace project creation, your build automation script (makefile) must meet certain requirements. See “Requirements for Project Creation from Build Systems” (Polyspace Bug Finder).

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” (Polyspace Bug Finder).

- 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” (Polyspace Bug Finder).

- 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” (Polyspace Bug Finder).

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” (Polyspace Bug Finder).

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” (Polyspace Bug Finder)
- “Provide Standard Library Headers for Polyspace Analysis” (Polyspace Bug Finder)

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
gnu8.x	C11	C++14
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.

Compiler	C Standard	C++ Standard
visual9.0, visual10.0, visual11.0, visual12.0	C99	C++03
visual14.0	C99	C++14
visual15.x	C99	C++14
visual16.x	C99	C++14
keil	C99	C++03
iar	C99	C++03
armcc	C99	C++03
armclang	C11	C++03
codewarrior	C99	C++03
cosmic	C99	Not supported
diab	C99	C++03
greenhills	C99	C++03
iar-ew	C99	C++03
microchip	C99	Not supported
renesas	C99	C++03
tasking	C99	C++03
ti	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” (Polyspace Bug Finder)
- “C++11 Language Elements Supported in Polyspace” (Polyspace Bug Finder)
- “C++14 Language Elements Supported in Polyspace” (Polyspace Bug Finder)
- “C++17 Language Elements Supported in Polyspace” (Polyspace Bug Finder)

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
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” (Polyspace Bug Finder).
<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” (Polyspace Bug Finder)

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 its corresponding Polyspace support. If your code contains nonsupported 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 is introduced and therefore does not affect Polyspace support for C++11.
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	<code>__func__</code> predefined identifier	Yes
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++11 Std Ref	Description	Supported
C++2011-N2427	Concurrency: Atomic operations	No
C++2011-N2429	Concurrency: Memory model	No new syntax/ keyword is introduced and therefore does not affect Polyspace support for C++11.
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++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++11 Std Ref	Description	Supported
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” (Polyspace Bug Finder)
- “C++14 Language Elements Supported in Polyspace” (Polyspace Bug Finder)
- “C++17 Language Elements Supported in Polyspace” (Polyspace Bug Finder)

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 its corresponding Polyspace support. If your code contains nonsupported 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 errors.
C++2014-N3642	Suffixes for user-defined literals indicating time (<code>h</code> , <code>min</code> , <code>s</code> , <code>ms</code> , <code>us</code> , <code>ns</code>) and strings (<code>s</code>)	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++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++14 Std Ref	Description	Supported
C++2014-N3651	Variable templates	Yes
C++2014-N3652	Declarations, conditions and loops in <code>constexpr</code> functions.	Yes
C++2014-N3653	<p>Initialization of aggregate classes with fewer initializers than members</p> <p>For instance, this initialization has fewer initializers than members. The member <code>c</code> is initialized with the value 0 and <code>d</code> is initialized with the value <code>s</code>.</p> <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 errors but the construct is not semantically 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++14 Std Ref	Description	Supported
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++2014-N3924	Discouraging use of <code>rand()</code>	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” (Polyspace Bug Finder)
- “C++11 Language Elements Supported in Polyspace” (Polyspace Bug Finder)
- “C++17 Language Elements Supported in Polyspace” (Polyspace Bug Finder)

C++17 Language Elements Supported in Polyspace

This table provides a partial list of C++ language elements that have been introduced since C++17 and its corresponding Polyspace support. If your code contains nonsupported constructions, Polyspace reports a compilation error.

C++17 Std Ref	Description	Supported
C++2017-N3921	<code>std::string-view</code> : Observe the content of an <code>std::string</code> object without owning the resource	Yes
C++2017-N3922	<ul style="list-style-type: none"> When used in copy-list-initialization, <code>auto</code> deduces the type to be an <code>std::initializer_list</code> if the elements of the initializer list have an identical type. Otherwise, the <code>auto</code> deduction is ill-formed. When using direct list-initialization with a braced initializer list containing a single element, <code>auto</code> deduces the type from that element. When using direct list-initialization with a braced initializer list containing more than a single element, <code>auto</code> deduction of type is ill-formed. 	Yes
C++2017-N3928	The <code>static_assert</code> declaration no longer requires a second argument. Invoking <code>static_assert</code> with no message is now allowed: <code>static_assert(N > 0);</code>	Yes
C++2017-N4051	C++ has templates that are not class templates, such as a template that takes templates as an argument. Previously, declaring such template-template parameters required the use of the <code>class</code> keyword. In C++17, you can use <code>typename</code> when declaring template-template parameters, such as: <pre>template <template <typename> typename Tmpl> struct X;</pre>	Yes
C++2017-N4086	Starting in C++17, trigraphs are no longer supported.	No
C++2017-N4230	Starting in C++17, use a qualified name in a namespace definition to define several nested namespaces at once. For instance, these code snippets are equivalent: <ul style="list-style-type: none"> <pre>namespace base::derived{ //.. }</pre> <pre>namespace { namespace derived{ //... } }</pre> 	Yes

C++17 Std Ref	Description	Supported
C++2017-N4259	The function <code>std::uncaught_exceptions</code> is introduced in C++17, which returns the number of exceptions in your code that are not handled. The function <code>std::uncaught_exception</code> , which returns a Boolean value, is deprecated.	Yes
C++2017-N4266	Starting in C++17, namespaces and enumerators can be annotated with attributes to allow clearer communication of developer intention.	Yes
C++2017-N4267	Starting in C++17, the prefix <code>u8</code> is supported. This prefix creates a UTF-8 character literal. The value of the UTF-8 character literal is equal to its ISO 10646 code point value if the code point value is in the C0 Controls and Basic Latin Unicode block.	Yes
C++2017-N4268	Allow constant evaluation of nontype template arguments.	Yes
C++2017-N4295	Allow fold expressions	Yes
C++2017-N4508	Allow untyped <code>std::shared_mutex</code>	The use of <code>std::shared_mutex</code> does not cause a compilation error. Polyspace does not support sharing mutex objects by using <code>std::shared_mutex</code> .
C++2017-P0001R1	Remove the use of the <code>register</code> keyword	Yes
C++2017-P0002R1	Remove <code>operator++(bool)</code>	Yes
C++2017-P0003R5	Remove deprecated exception specifications by using <code>throw(<>)</code>	Bug Finder removes the exception specification specified by using <code>throw()</code> statements. Code Prover raises a compilation error when <code>throw()</code> statements are present in C++17 code.
C++2017-P0012R1	Make exception specifications part of the type system	Yes
C++2017-P0017R1	Aggregate initialization of classes with base classes	Yes
C++2017-P0018R3	Allow capturing the pointer <code>*this</code> in Lambda expressions	Yes
C++2017-P0024R2	Standardization of the C++ technical specification for Extension for Parallelism	Polyspace supports this feature when you use the Visual 15.x and Intel C++ 18.0 compilers.

C++17 Std Ref	Description	Supported
C++2017-P002842	Using attribute namespaces without repetition	Yes
C++2017-P0035R4	Dynamic memory allocation for over-aligned data	Yes
C++2017-P0036R0	Unary fold expressions and empty parameter packs	Yes
C++2017-P0061R1	Use of <code>__has_include</code> in preprocessor conditionals	Yes
C++2017-P0067R5	Elementary string conversions	No
C++2017-P0083R3	Splicing maps and sets	Polyspace supports this feature when the compiler you use also supports this feature. For instance, Polyspace supports this feature when you use g++ as compiler.
C++2017-P0088R3	<code>std::variant</code>	Partially supported.
C++2017-P0091R3	Template argument deduction for class templates	Partially supported.
C++2017-P0127R2	Non-type template parameters that have auto type	Yes
C++2017-P0135R1	Guaranteed copy elision	Partially supported.
C++2017-P0136R1	New specification for inheriting constructors	No
C++2017-P0137R1	Replacement of class objects containing reference members	Yes
C++2017-P0138R2	Direct-list-initialization of enumerations	Yes
C++2017-P0145R3	Stricter expression evaluation order	No new syntax/keyword is introduced and therefore does not affect Polyspace support for C++17.
C++2017-P0154R1	Hardware interference size	Supported with Visual Studio Compiler
C++2017-P0170R1	<code>constexpr</code> Lambda expressions	Partially supported
C++2017-P018R0	Differing begin and end types in range-based for loops	Yes
C++2017-P0188R1	<code>[[fallthrough]]</code> attribute	Yes

C++17 Std Ref	Description	Supported
C++2017-P0189R1	[[nodiscard]] attribute	Yes
C++2017-P0195R2	Pack expansions in using-declarations	Yes
C++2017-P0212R1	[[maybe_unused]] attribute	Yes
C++2017-P0217R3	Structured Bindings	Polyspace does not support binding by using an rvalue.
C++2017-P0218R1	std::filesystem	No
C++2017-P0220R1	std::any	Yes
C++2017-P0220R1	std::optional	Bug Finder supports the syntax. The semantics are partially supported. Code Prover does not support this feature.
C++2017-P0226R1	Mathematical special functions	No
C++2017-P0245R1	Hexadecimal floating-point literals	Yes
C++2017-P0283R2	Ignore unknown attributes	Yes
C++2017-P0292R2	constexpr if statements	Yes
C++2017-P0298R3	std::byte	Yes
C++2017-P0305R1	init-statements for if and switch	Yes
C++2017-P0386R2	Inline variables	No
C++2017-P0522R0	Invoke partial ordering to determine when a template <i>template-argument</i> is a valid match for a <i>template-parameter</i>	Partially supported

See Also

C++ standard version (-cpp-version)

More About

- “C/C++ Language Standard Used in Polyspace Analysis” (Polyspace Bug Finder)
- “C++11 Language Elements Supported in Polyspace” (Polyspace Bug Finder)
- “C++14 Language Elements Supported in Polyspace” (Polyspace Bug Finder)

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” (Polyspace Bug Finder)

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” (Polyspace Bug Finder).
- Contact MathWorks Technical Support. For more information, see “Contact Technical Support About Issues with Running Polyspace” (Polyspace Bug Finder).
- If you build your code in Cygwin™, you must be using version 2.x or 3.x of Cygwin for Polyspace project creation from your build system (for instance, Cygwin version 2.10 or 3.0).
- 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\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`.

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” (Polyspace Bug Finder).

- 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.

Similar considerations apply to other security applications such as security-related products from CylanceProtect, Avecto and Tanium.

- If your computer hibernates during the build process, Polyspace might not be able to trace your build.
- When creating projects from build commands in the Polyspace User Interface, you might encounter errors such as `libcurl.so.4: version 'CURL_OPENSSL_3' not found`. In such

cases, create the Polyspace project by using the command `polyspace-configure` in the system command line interface, using the build command as the argument. See `polyspace-configure`.

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

See Also

`polyspace-configure`

Related Examples

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

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.

For example, depending on how you define the `sbit` data type, you use these options:

- `sbit ADST = ADCUP^7;`
Use options: `-compiler keil -sfr-type sfr=8`
- `sbit ADST = ADCUP.7;`
Use options: `-compiler iar -sfr-type sfr=8`

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” (Polyspace Bug Finder).

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 & Compiler (Polyspace Bug Finder) 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” (Polyspace Bug Finder)

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” (Polyspace Bug Finder). 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 & Compiler (Polyspace Bug Finder) 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 8.1. 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” (Polyspace Bug Finder)

Approximations Used During Bug Finder Analysis

Inputs in Polyspace Bug Finder

A Bug Finder analysis by default does not return a defect caused by a special value of an unknown input, unless the input is bounded. Polyspace makes no assumption about the value of unbounded inputs when your source code is incomplete. For example, in the following code Bug Finder detects a **division by zero** in `foo_1()`, but not in `foo_2()`:

```
int foo_1(int p)
{
    int x = 0;
    if ( p > -10 && p < 10 ) /* p is bounded by if statement */
        x = 100/p; /* Division by zero detected */

    return x;
}

int foo_2(int p) /* p is unbounded */
{
    int x = 0;
    x = 100/p; /* Division by zero not detected */

    return x;
}
```

To set bounds on your input, add constraints in your code such as `assert` or `if`. At the cost of a possibly longer runtime, you can perform a more exhaustive analysis where all values of function inputs are considered when showing defects. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values” (Polyspace Bug Finder Server).

See Also

“Global Variables in Polyspace Bug Finder” on page 9-3 | “Bug Finder Analysis Assumptions”

Global Variables in Polyspace Bug Finder

When you run a Bug Finder analysis, Polyspace makes certain assumptions about the initialization of global variables. These assumptions depend on how you declare and define global variables. For example, in this code

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

Bug Finder detects a **division by zero** defect with the variable `gvar` in these cases:

- You define `int gvar;` in the source code and provide a `main` function that calls `foo`. Bug Finder follows ANSI standards that state the variable is initialized to zero.
- You define `int gvar;` or declare `extern int gvar;` in the source code. Another function calls `foo` and sets `gvar=0`. Otherwise, when your source files are incomplete and do not contain a `main` function, Bug Finder makes no assumption about the initialization of `gvar`.
- You declare `const int gvar;`. Bug Finder assumes `gvar` is initialized to zero due to the `const` keyword.

At the cost of a possibly longer runtime, you can perform a more exhaustive analysis where all values are considered for each read of a global variable by `foo` or of its callees when showing defects. See “Extend Bug Finder Checkers to Find Defects from Specific System Input Values” (Polyspace Bug Finder Server).

See Also

“Inputs in Polyspace Bug Finder” on page 9-2 | “Bug Finder Analysis Assumptions”

Volatile Variables in Polyspace Bug Finder

You use the `volatile` keyword to inform the compiler that the value of a variable might change at any time without an explicit write operation. When you run an analysis, Polyspace Bug Finder makes these assumptions about volatile variables:

- **Global volatile variables**

- If you declare a global volatile variable as `const`, Polyspace uses the initialization value of the variable or the initialization range if you use the **PERMANENT Init Mode** to constrain the range of the variable externally. Polyspace uses the initialization value or range for every read of the variable. See “External Constraints for Polyspace Analysis” (Polyspace Bug Finder).

For instance, in this code:

```
const volatile volatile_var; // Global variable initialized to 0
const volatile volatile_var_10=10;
const volatile volatile_var_drs=3; // Variable constrained to range [-5 .. 5]

int func(void){
    int i= 10 % volatile_var; // Defect
    int j= 10 % volatile_var_10; // No defect
    int k= 10 % volatile_var_drs; // Defect
    return i+j+k;
}
```

Polyspace detects an **Integer division by zero** defect for `volatile_var` since it is initialized to zero. Polyspace detects an **Integer division by zero** for `volatile_var_drs` because it is externally constrained to the range [-5 .. 5]. All reads of `volatile_var_10` cause no defect.

- For non-`const` global volatile variables, Polyspace ignores the initialization value of the variable, and then considers the input unknown for each read of the variable. If you use the **PERMANENT Init Mode** to constrain the range of the variable externally, Polyspace uses this range for every read of the variable. See “External Constraints for Polyspace Analysis” (Polyspace Bug Finder).

For instance, in this code:

```
volatile volatile_var; // Global variable initialized to 0
volatile volatile_var_drs=3; // Variable constrained to range [-5 .. 5]

int func(void){
    int i= 10 % volatile_var; // No defect
    int j= 10 % volatile_var_drs; // Defect
    return i+j;
}
```

Polyspace detects an **Integer division by zero** defect for `volatile_var_drs` because it is externally constrained to the range [-5 .. 5]. All reads of `volatile_var` cause no defect.

- **Local volatile variables**

Polyspace ignores the initialization value of local volatile variables, and then considers the input unknown for each read of the variable. For example, in this code:

```
int foo(void){
    volatile var=0;
    return 1/var; // No defect
}
```

Polyspace detects no defect. You cannot use external constraints to constrain the range of local variables.

At the cost of a possibly longer runtime, you can perform a more exhaustive analysis where Polyspace considers all values for each read of a volatile variable. See `Run stricter checks considering all values of system inputs (-checks-using-system-input-values)`. When you use this option to analyze all the preceding code examples, Polyspace detects additional **Integer division by zero** defects on the lines labeled with comment `// No defect`, including for the local volatile variable example.

See Also

“Inputs in Polyspace Bug Finder” on page 9-2 | “Bug Finder Analysis Assumptions”

