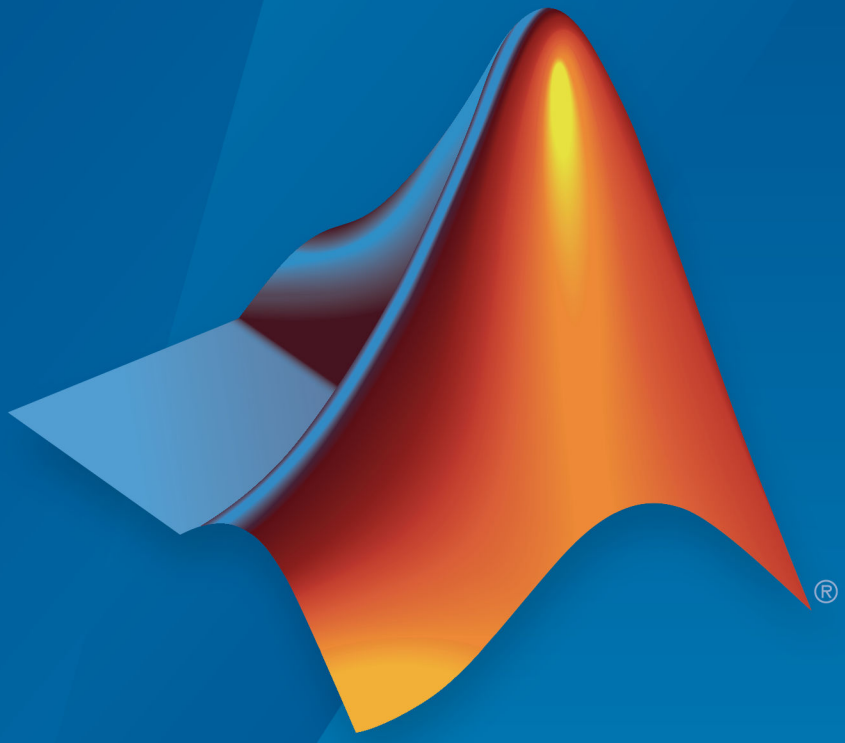
Polyspace® Code Prover™ Access™

User's Guide



R2019a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

Polyspace® Code Prover™ Access™ User's Guide

Revision History

March 2019	Online only	New for Version 2.0 (R2019a)

Contents

1 Interpret Polyspace Code Prover Access Results

2 Fix or Comment Polyspace Results

3 Manage Results

4 Reviewing Checks

5 Coding Rule Sets and Concepts

6 Approximations Used During Verification

Interpret Polyspace Code Prover Access Results

Interpret Polyspace Code Prover Access Results

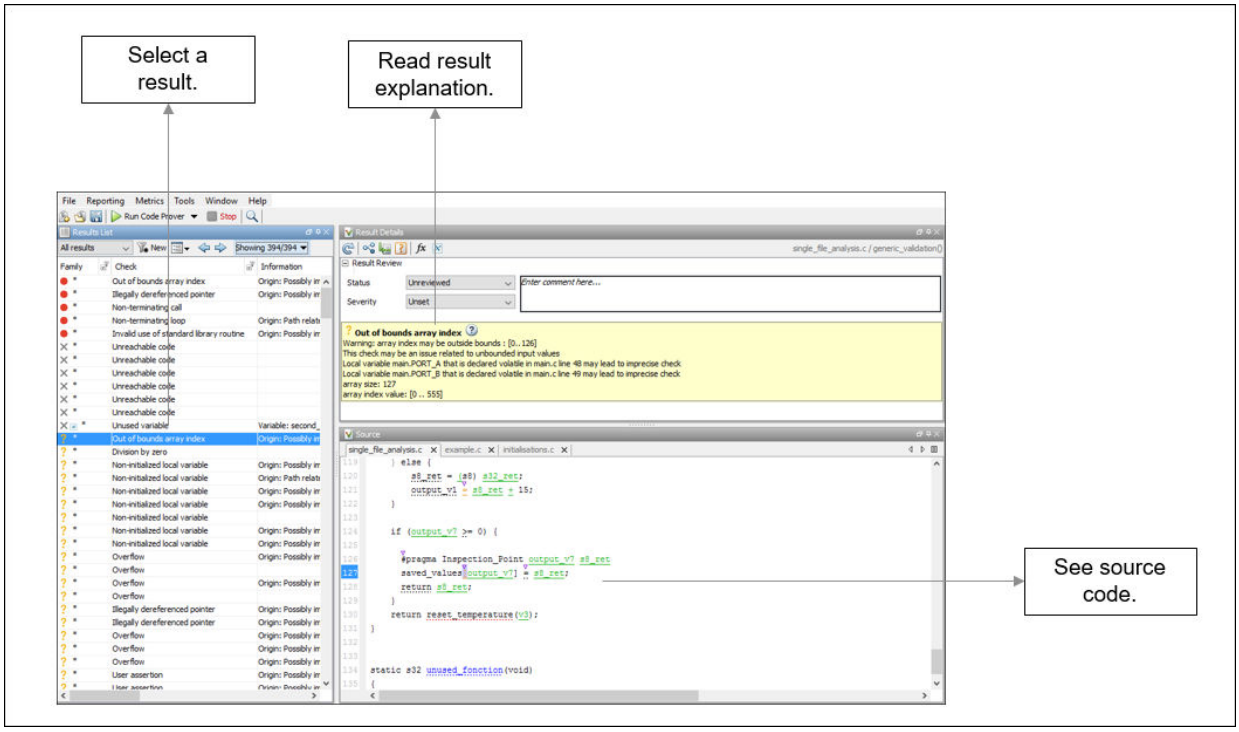
When you open the results of a Polyspace Code Prover analysis, you see a list on the **Results List** pane. The list consists of run-time checks, coding rule violations, code metrics and global variable usage.

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**.

  Because the results of a Code Prover run-time check are dependent on the results of previous checks, it helps to go through run-time checks from the beginning to the end of a function.

See also “Filter and Sort Results” on page 3-2. Once you narrow down the list, you can begin reviewing individual results. This topic describes how to review a result.



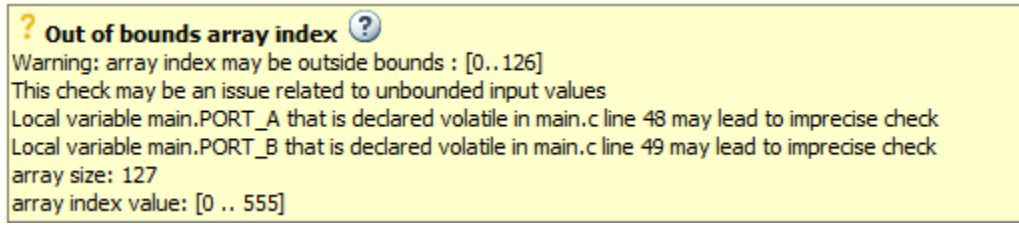
To begin your review, select a result in the list.

Interpret Result

Interpret Message

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

At this point, you might be ready to decide whether to fix the issue.



The message consists of several parts:

- Check color and icon: See “Code Prover Result and Source Code Colors” on page 1-11. In case of checks for run-time errors:
  - : Red indicates a definite error.
  - : Orange indicates a possible error.
  - : Gray indicates unreachable code.
  - : Green indicates that a specific error cannot happen.
- Description of the run-time check.

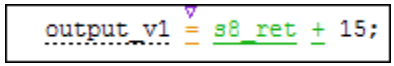
  In the preceding example, the check determines if an array index goes outside the array bounds.
- Values relevant to the run-time check.

  In the example, the message states the array size (127), the array bounds (0..126), and the range of values that the array index variable can take at that point in the code (0..555).
- Relevant sources of imprecision (for orange checks).

  In the example, the message states that two volatile variables might be responsible for the check.

See Variable Ranges in Source Code Tooltips

On the **Source** pane, variables and operations with tooltips are underlined.



In this example, tooltips appear on:

- `s8_ret`: You see its data type and range of values before the + operation.

  If a data type conversion occurs during the + operation, you also see this conversion in the tooltip.
- +: You see the value of the left and right operand, and the result.
- =: You see any data type conversion that occurs during the assignment and the result.

Get Additional Help

Sometimes, you need additional help for certain results. To open a help page for the selected result, click the  icon. See code examples that illustrate the result.

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 possibly a previous `if` or `while` condition that is always false.

Navigate in Source Code

Sometimes, the **Result Details** pane shows one sequence of events that leads to the result. However, in most situations, you have to find your own navigation pathways through the code. Using tooltips on variables, follow the propagation of variable ranges as you navigate through the code.

```
int func (int var) { /* Initial range of var */
     …
```

```
    var -= get ();  /* New range of var */
    …
    set(&var);     /* New range of var */
}
```

Use these quick navigation pathways in the user interface:

- Search for all references to a variable and browse through them.

  Right-click the variable name on the **Source Code** pane and select **Search For All References**. Alternatively, double-click the variable. These options perform more than a string match. The options show only instances of a specific variable and not other variables with the same name in other scopes.
- Navigate from a function call to its definition.

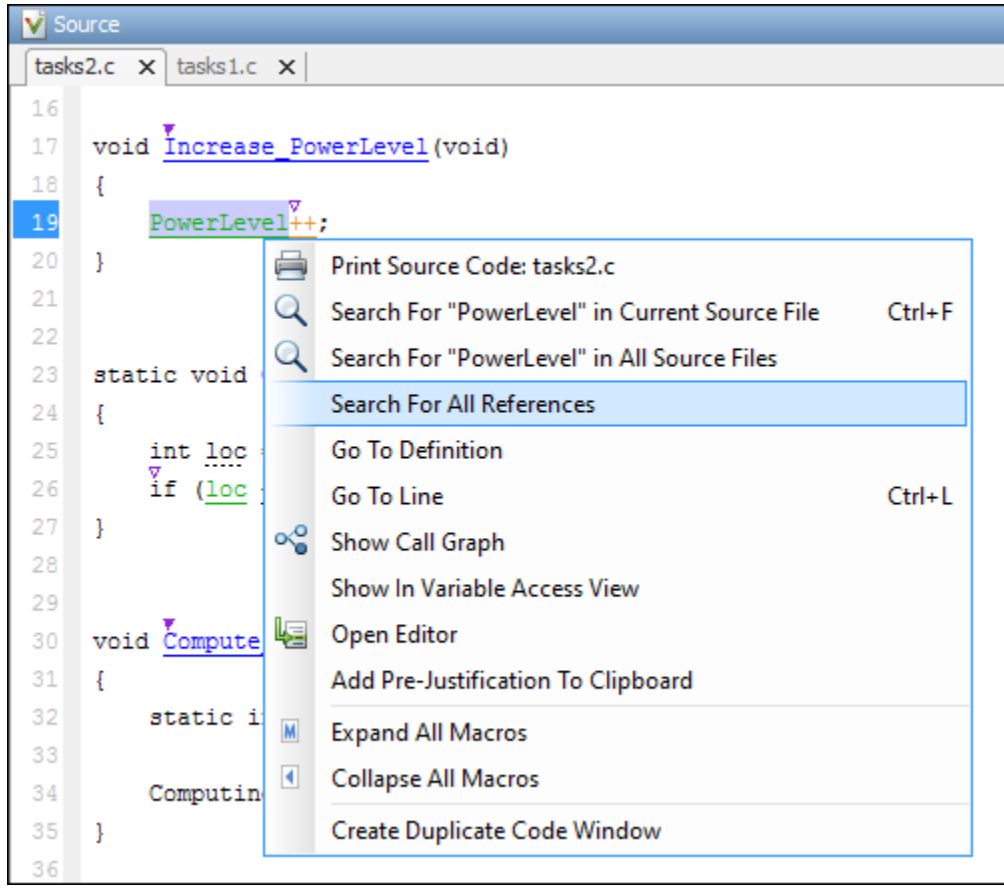
  Right-click the function name on the **Source Code** pane. Select **Go To Definition**.
- Navigate from a function to its callers and callees.

  Click the  icon on the **Result Details** pane. You see the function containing the result, with its callers and callees. Click a caller or callee name to navigate to the call site. Double-click a name to navigate to the definition.

  Alternatively, click the  icon to see a graphical representation of the call sequence leading to the result. To navigate to functions in this sequence, click through nodes in the graph.
- Navigate from a function call or loop keyword to an error in the function or loop body.

  If the error occurs only in a specific function call or specific loop iteration, the function call or loop iteration is highlighted red. Right-click the red function call or loop keyword. Select **Go To Cause** if the option is available.
- Navigate across all instances of a global variable.

  Click the  icon on the **Result Details** pane. See all global variables in the result and read/write operations on them.



Before you begin navigating through pathways in your code, determine what you are looking for and choose the appropriate navigation tool. For instance:

- To investigate a **Non-initialized variable** check, you might want to make sure that the variable is not initialized at all. Look for previous instances of the variable and see if it is initialized.
- 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.`

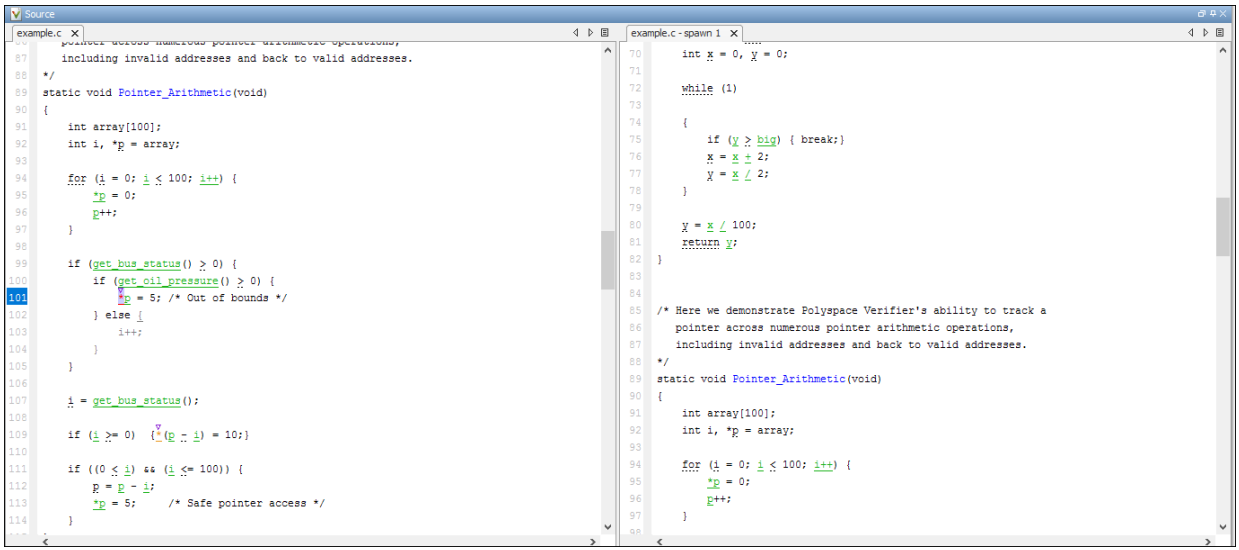
you might want to navigate from a function call to the function definition.

For other examples of what to look for, see “Code Prover Run-Time Checks” on page 1-18. After you navigate away from the current result, use the  icon on the **Result Details** pane to return to that result.

If you click a source code token containing a result, the previous result selection on the **Results List** and details on the **Result Details** pane do not change. You can keep the result in the results list and the result details pinned while navigating in the source code. Sometimes, you might want to see the result associated with a token. To update the result selection and the details, `Ctrl`-click the token or right-click and select **Select Results At This Location**.

Navigate in Separate Window

If reviewing a result requires deeper navigation in your source code, you can create a duplicate source code window that focuses on the result while you navigate in the original source code window.



Right-click in the **Source Code** pane and select **Create Duplicate Code Window**. Right-click the tab showing the duplicate file name (ending with `-spawn 1`) and select **New Vertical Group**.

Perform the navigation steps in the duplicate file window while the defect still appears in the original file window. After the investigation is complete, close the duplicate window.

See Also

More About

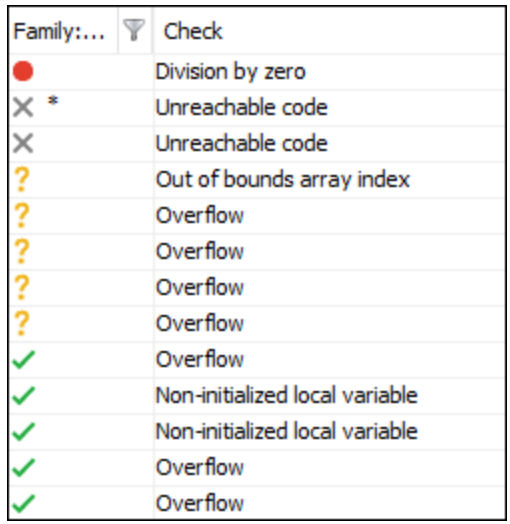
- “Filter and Sort Results” on page 3-2
- “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2

Code Prover Result and Source Code Colors

This topic explains the various colors used in displaying the results of a Polyspace Code Prover analysis.

Result Colors

Polyspace displays the different verification results with specific icons and colors on the **Results List** and **Result Details** pane.



Run-Time Checks

Polyspace Code Prover checks each operation in your code for particular run-time errors. The software assigns a color to the operation based on whether it proved the existence or absence of a run-time error on all or some execution paths.

Check Color	Purpose	Example	Icon
Red	Highlights operations that are proven to cause a particular error on all execution paths*. Polyspace Code Prover verification determines errors with reference to the language standard. Though some of the errors can be acceptable for a particular compilation environment, they violate the language standard. To allow some of the environment-dependent behavior, use appropriate analysis options. For more on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server™.	Red **Overflow** on: `z = x+y;` The operation + overflows for every value of `x` and `y` that the verification considers at that point.	

Gray	Highlights unreachable code.	Gray **Unreachable code** check:
		
		`if(x>0)`
		`{}`
		`else`
		`{}`
		
		The `else` branch is unreachable for all values of `x` that the verification considers at that point.
			

Check Color	Purpose	Example	Icon
Orange	Highlights operations that can cause an error on certain execution paths. For more information, see “Orange Checks in Code Prover” on page 1-57.	Orange **Overflow** on: `z = x+y;` The analysis could not prove whether the operation + overflows. The most common reason is that the operation overflows only for some values of x and y that the verification considers at that point. You can use the tooltips on the variables x and y in the operation to see the range of values that the verification considers.	
Green	Highlights operations that are proven to not cause a particular error on all execution paths*.	Green **Overflow** on: `z = x+y;` The operation + does not overflow for all values of x and y that the verification considers at that point.	

* For most checks, the software terminates an execution path following the first run-time error on the path. Therefore, if it proves a definite error (red) or absence of error (green) on an operation, the proof is valid only for the execution paths that have not yet been terminated at that point in the code. See “Code Prover Analysis Following Red and Orange Checks” on page 1-49.

Other Results

Besides checks for run-time errors, Polyspace Code Prover also displays other results about your code.

Result	Purpose	Icon
Coding rule violations	Indicates violation of predefined or custom coding rules.	for predefined rules and for custom rules.
Code metrics	Indicates code complexity metrics.	for metrics that do not exceed a limit you specified and for metrics that exceed a limit.
Global variables	Indicates global variable declaration.	for shared potentially unprotected variables and for non-shared unused variables

Source Code Colors

Polyspace uses the following color scheme for displaying code on the **Source Code** pane.

- *Lines with checks*:

  For every check on the **Results List** pane, Polyspace assigns the check color to the corresponding section of code.

  - For lines containing macros, if the macro is collapsed, then Polyspace colors the entire line with the color of the most severe check on the line. The severity decreases in this order: red, gray, orange, green.

    This unreachable `for` loop contains a macro `MAX_SIZE`. The entire line is colored gray.

    ```
    for (i = 0; i < MAX_SIZE; i++) {
    ```

    If there is no check in a line containing a macro, Polyspace underlines the line in black when the macro is collapsed.
  - For all other lines, Polyspace colors only the keyword or identifier associated with the check.

    This assignment has three checks: `i` and `used_global` are initialized but the array `tab` can be accessed outside its bounds. The `[` operator is colored orange to indicate the issue.

```
tab[i] = used_global;
```

- *Lines with coding rule violations*:

  For every coding rule violation on the **Results List** pane, Polyspace assigns to the corresponding keyword or identifier:

  - A ▽ (inverted triangle) symbol if the coding rule is a predefined rule. The predefined rules available are MISRA C®, MISRA® AC AGC, MISRA C++, or JSF® C ++.

    This `if` statement and `||` operation violates MISRA rules.

    ```
    if (x < 0 || x > 20) return -1;
    ```

  - A ▼ symbol if the coding rule is a custom rule.

    This function name violates a custom naming convention.

    ```
    int polynomia(int input)
    ```

- *Lines with tooltips*:

  If a tooltip is available for a keyword or identifier on the **Source Code** pane, Polyspace:

  - Uses solid underlining for the keyword or identifier if it is associated with a check.

    This line has both checks and tooltips on `input`, `%` and `used_global`.

    ```
    result = input % used_global;
    ```

  - Uses dashed underlining for the keyword or identifier if it is not associated with a check.

    This line has tooltips on `for` and `<`, but no checks on them.

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

  - Uses dashed red underlining on function calls to indicate that the function body contains a definite run-time error. The tooltip shows the line in the function body that causes the error.

This call to `function_with_red` leads to a run-time error.

```
i = function_with_red(0);
```

- *Function definitions*:

  When a function is defined, Polyspace colors the function name in blue.

```
void task1(void) {
```

- *Lines deactivated due to conditional compilation*:

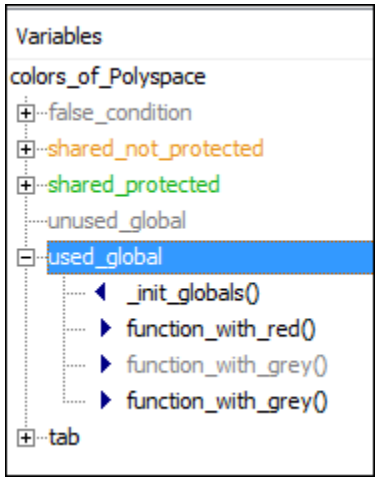
  Polyspace assigns a lighter shade of gray to code deactivated due to conditional compilation. Such code occurs, for instance, in `#ifdef` statements where the macro for a branch is not defined. This code does not affect the verification.

```
#ifdef ACTIVE
        /* this code is not processed! */
        tab[0] = used_global;
```

Global Variable Colors

The **Variable Access** pane shows the global variables in your code along with the read and write operations on the variables.

For instance, `used_global` is a global variable that is accessed four times: once during initialization, once in the function `function_with_red`, and twice in the function `function_with_grey`.



The color scheme is as follows:

- *Variable colors*:
  - Orange: Shared, unprotected global variable (only applicable to multitasking code)
  - Green: Shared, protected global variable (only applicable to multitasking code)
  - Black: Unshared, used global variable
  - Gray: Unshared, unused global variable

  See “Global Variables”.
- *Operation colors*: If an operation occurs in unreachable code, it is grey, otherwise black.

  In the preceding example, one operation in the function `function_with_grey` is unreachable but the other is reachable.

For more information, see “Global Variables” on page 1-40.

Code Prover Run-Time Checks

Polyspace Code Prover checks each operation in your code for certain run-time errors and displays the result as a red, green or orange check. For more information, see “Code Prover Result and Source Code Colors” on page 1-11.

You must review a red or orange check and determine whether to fix your code. The tables below list the checks that Polyspace Code Prover performs and how you can review them.

Data Flow Checks

Check	How to Review	Details
`Function not called`	Investigate why a function does not appear in the call graph starting from the `main` or another entry point function.	“Review and Fix Function Not Called Checks” on page 4-16
`Function not reachable`	Identify the call sites of a function and investigate why they occur in unreachable code.	“Review and Fix Function Not Reachable Checks” on page 4-19
`Non-initialized local variable`	Locate prior variable initializations if any and see if your program can bypass them.	“Review and Fix Non-initialized Local Variable Checks” on page 4-50
`Non-initialized pointer`	Locate prior pointer initializations if any and see if your program can bypass them.	“Review and Fix Non-initialized Pointer Checks” on page 4-54
`Non-initialized variable`	Locate prior initializations of the global variable if any and see if your program can bypass them.	“Review and Fix Non-initialized Variable Checks” on page 4-57
`Return value not initialized`	Identify paths through your function body that do not end in a `return` statement.	“Review and Fix Return Value Not Initialized Checks” on page 4-86

Check	How to Review	Details
`Unreachable code`	Investigate why a conditional statement in your code is redundant, for instance, always true or always false.	“Review and Fix Unreachable Code Checks” on page 4-93

Numerical Checks

Check	How to Review	Details
`Division by zero`	Review prior operations in your code that lead to zero value of a denominator.	“Review and Fix Division by Zero Checks” on page 4-10
`Invalid shift operations`	Review prior operations in your code that lead to a shift amount outside bounds or a negative value being left-shifted.	“Review and Fix Invalid Shift Operations Checks” on page 4-37
`Overflow`	Review prior operations in your code that lead to an operation overflowing.	“Review and Fix Overflow Checks” on page 4-79

Static Memory Checks

Check	How to Review	Details
`Absolute address usage`	Review uses of absolute address in your code and make sure that the addresses are valid.	“Review and Fix Absolute Address Usage Checks” on page 4-3
`Illegally dereferenced pointer`	Review prior operations in your code that lead to a pointer pointing outside its allocated memory buffer.	“Review and Fix Illegally Dereferenced Pointer Checks” on page 4-23
`Out of bounds array index`	Review prior operations in your code that lead to an array index being greater than or equal to array size.	“Review and Fix Out of Bounds Array Index Checks” on page 4-74

Control Flow Checks

Check	How to Review	Details
`Non-terminating call`	Review operations in the function body and find which run-time error occurs because of issues specific to the current function call.	“Review and Fix Non-Terminating Call Checks” on page 4-60
`Non-terminating loop`	Review operations in the loop and determine why the loop does not terminate or why a definite run-time error occurs in one of the loop runs.	“Review and Fix Non-Terminating Loop Checks” on page 4-65

C++ Checks

Check	How to Review	Details
`Invalid C++ specific operations`	Determine root cause of nonpositive array size or incorrect usage of the `typeid` or the `dynamic_cast` operator.	“Review and Fix Invalid C++ Specific Operations Checks” on page 4-34
`Function not returning value`	Identify paths through your function body that do not end in a `return` statement.	“Review and Fix Function Not Returning Value Checks” on page 4-21
`Incorrect object oriented programming`	Investigate why a certain `virtual` member call or `this` pointer usage represents an incorrect pattern of object oriented programming.	“Review and Fix Incorrect Object Oriented Programming Checks” on page 4-31
`Null this-pointer calling method`	Investigate why the pointer to the current object can be `NULL`-valued.	“Review and Fix Null This-pointer Calling Method Checks” on page 4-72

Check	How to Review	Details
`Uncaught exception`	Investigate how an exception can escape uncaught from the function where it is thrown.	“Review and Fix Uncaught Exception Checks” on page 4-90

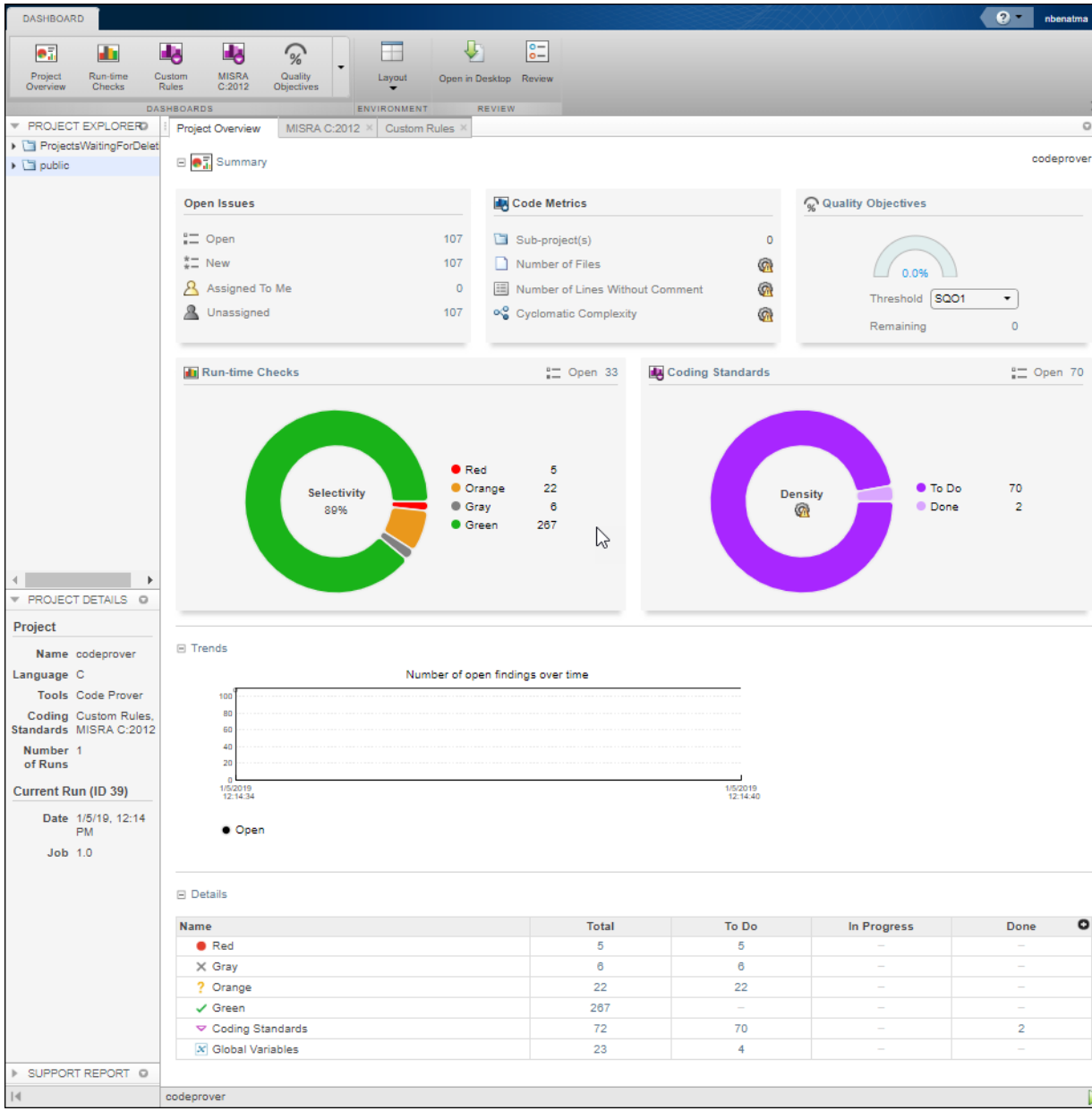
Other Checks

Check	How to Review	Details
`Correctness condition`	Find the root cause of a function pointer misuse, incorrect array conversion or variable values outside specified constraints.	“Review and Fix Correctness Condition Checks” on page 4-4
`Invalid use of standard library routine`	Investigate why the arguments in the current call to the standard library routine are invalid.	“Review and Fix Invalid Use of Standard Library Routine Checks” on page 4-43
`User assertion`	Investigate why the condition in an `assert` statement fails.	“Review and Fix User Assertion Checks” on page 4-99

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.


DASHBOARD
nbenatma
Project Overview
Run-time Checks
Custom Rules
MISRA C:2012
Quality Objectives
Layout
Open in Desktop
Review
DASHBOARDS
ENVIRONMENT
REVIEW
PROJECT EXPLORER
ProjectsWaitingForDelet
public
Project Overview
MISRA C:2012
Custom Rules
Summary
codeprover
Open Issues
Open 107
New 107
Assigned To Me 0
Unassigned 107
Code Metrics
Sub-project(s) 0
Number of Files
Number of Lines Without Comment
Cyclomatic Complexity
Quality Objectives
0.0%
Threshold SQO1
Remaining 0
Run-time Checks
Open 33
Selectivity
89%
Red 5
Orange 22
Gray 6
Green 267
Coding Standards
Open 70
Density
To Do 70
Done 2
Trends
Number of open findings over time
100
80
60
40
20
0
1/5/2019 12:14:34
1/5/2019 12:14:40
Open
Details
Name
Total
To Do
In Progress
Done
Red 5 5
Gray 6 6
Orange 22 22
Green 267
Coding Standards 72 70 2
Global Variables 23 4
PROJECT DETAILS
Project
Name codeprover
Language C
Tools Code Prover
Coding Standards Custom Rules, MISRA C:2012
Number of Runs 1
Current Run (ID 39)
Date 1/5/19, 12:14 PM
Job 1.0
SUPPORT REPORT
codeprover

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” on page 3-2.
- Open the current project findings in the Polyspace desktop interface.
- Manage projects and user authorizations. See “Manage Permissions and View Project Trends”.

Code Metrics Dashboard

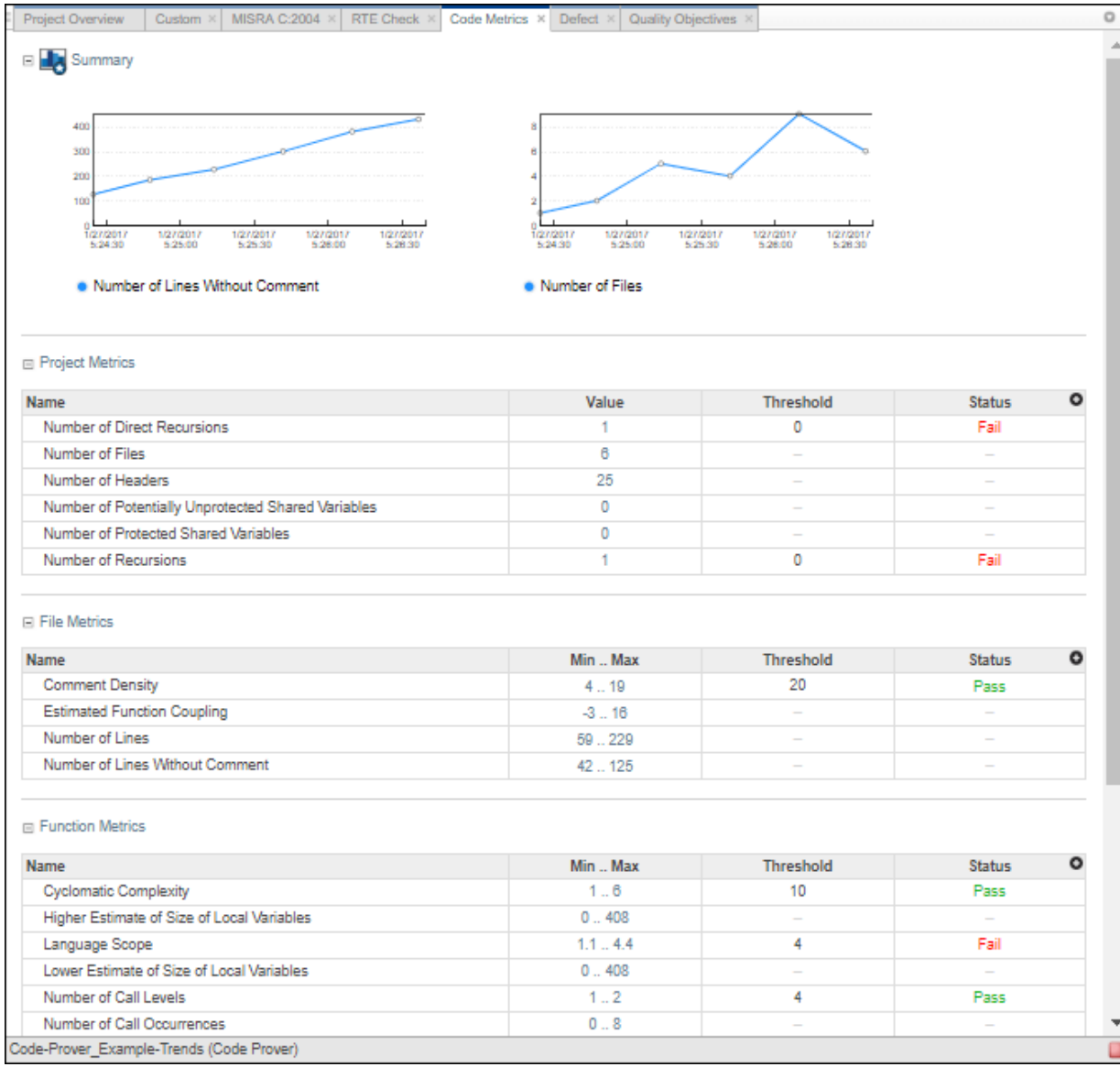
To view the code complexity metrics that Polyspace computes, use the **Code Metrics** dashboard. See “Code Metrics”. Only when you use the option `Calculate code metrics (-code-metrics)` doesPolyspace compute the code complexity metrics during analysis. For more information on analysis options, see the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

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.



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

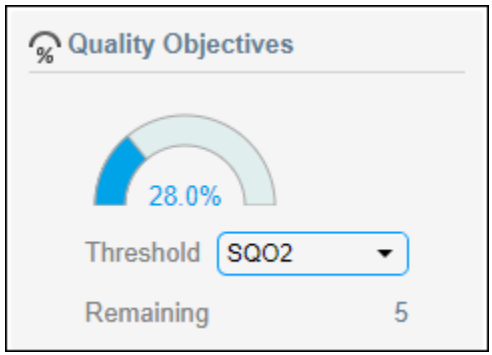
The other sections of the dashboard display tables with the computed value or range of the different project, file, and function metrics. When applicable, the table shows the predefined threshold and pass/fail status for the corresponding code metric. For a list of code complexity metrics thresholds, see “HIS Code Complexity Metrics” on page 1-124. 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

The Quality Objectives dashboard is available only for Code Prover analysis results.

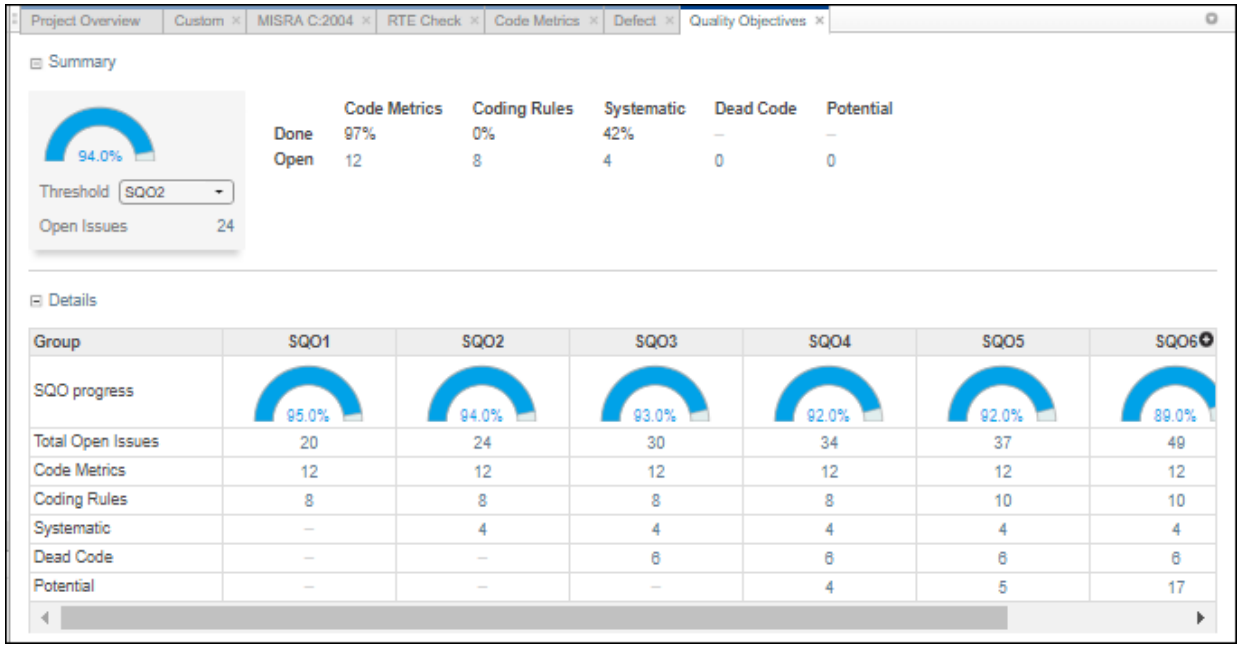
To monitor the quality of your code against predefined “Software Quality Objectives” on page 1-69, use the **Quality Objectives** dashboard.



In the **Project Overview** dashboard, use the **Quality Objectives** card to get a quick overview of your progress in achieving a quality objective threshold. From the **Threshold** drop-down list, select a threshold and view the percentage of findings that you have already addressed to achieve the threshold. The card also displays the number of findings you still 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 you can use the **Threshold** drop-down list to pick a threshold and see the remaining open issues, with a breakdown per category, such as code metrics or coding rules.

In this **Quality Objectives** dashboard, 94% of the findings required to achieve threshold `SQO2` have been addressed. There are 24 open issues, including 12 **Code Metrics**, 8 **Coding Rules**, and 4 **Systematic** issues. Open issues are issues with a review status of `Unreviewed`, `To fix`, `To investigate`, or `Other`.



The 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.

See Also

More About

- “Software Quality Objectives” on page 1-69

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: • For defects: Groups such as static memory, numerical, control flow, concurrency, etc. • For coding rule violations: Groups defined by the coding rule standard. For instance, MISRA C: 2012 defines groups related to code constructs such as functions, pointers and arrays, etc.
Check	Result name, for instance: • For defects: Defect name • For coding rule violations: Coding rule number
Information	Result sub-type when available. • For defects: Impact classification. For coding standards: required or mandatory, rule or recommendation.
Detail	Additional information about a result. The column shows the first line of the **Result Details** pane. For an example of how to use this column, see the result `MISRA C:2012 Dir 1.1`.
File	File containing the instruction where the result occurs

Attribute	Description
Function	Function containing the instruction where the result occurs. If the function is a method of a class, it appears in the format *`class_name`*`::`*`function_name`*.
Status	Review status you have assigned to the result. The possible statuses are: • `Unreviewed` (default status) • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other`
Severity	Level of severity you have assigned to the result. The possible levels are: • `Unset` • `High` • `Medium` • `Low`
Assigned to	User name of reviewer assigned to this result.
Ticket Key	When you create a JIRA issue for a result, this field contains the issue ID. Click the ID to open the issue in the JIRA interface.
Comments	Comments you have entered about the result
Folder	Path to the folder that contains the source file with the result

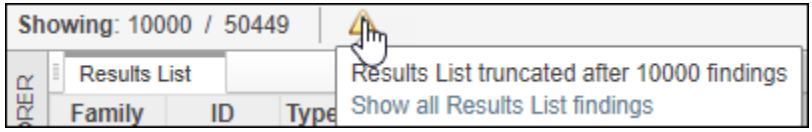
To show or hide any of the columns, click the  icon in the upper-right of the **Results List** pane, then select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the results.

- Organize your result review using filters in the toolstrip or in the context menu. For more information, see “Filter and Sort Results” 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.

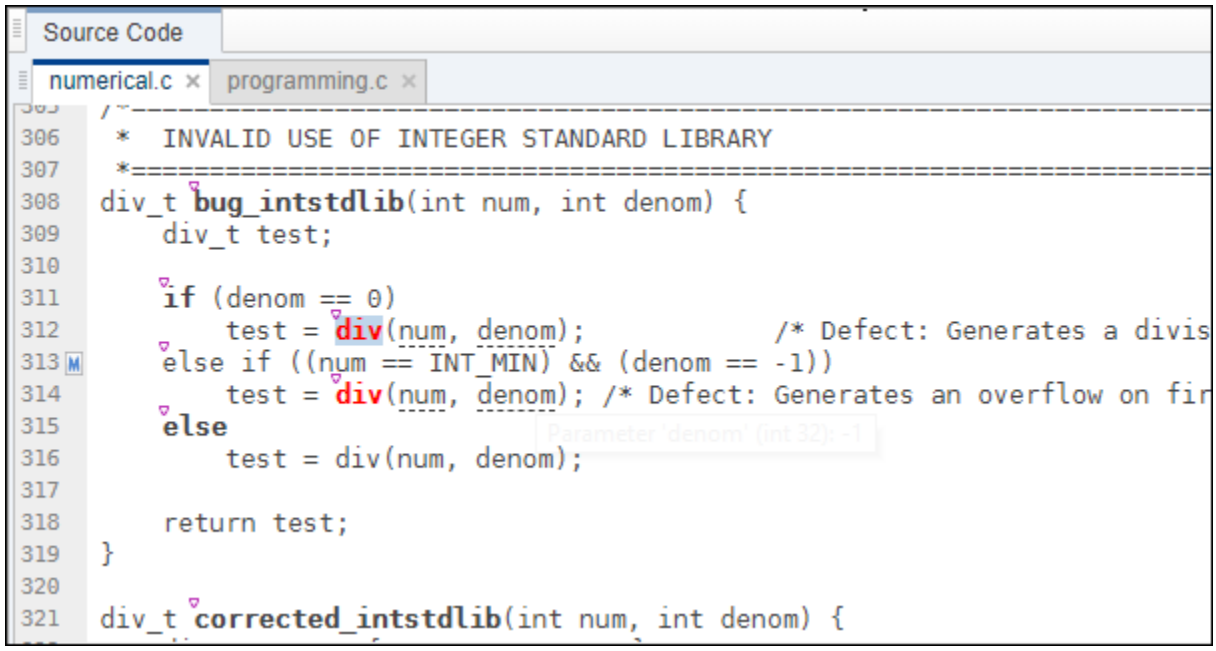
If the **Results List** exceeds 10000 findings, Polyspace Access truncates the list and displays this icon ⚠ in the filters bar. To show all findings, see the contextual help of the icon.



The 10000 findings limit is preset and cannot be changed.

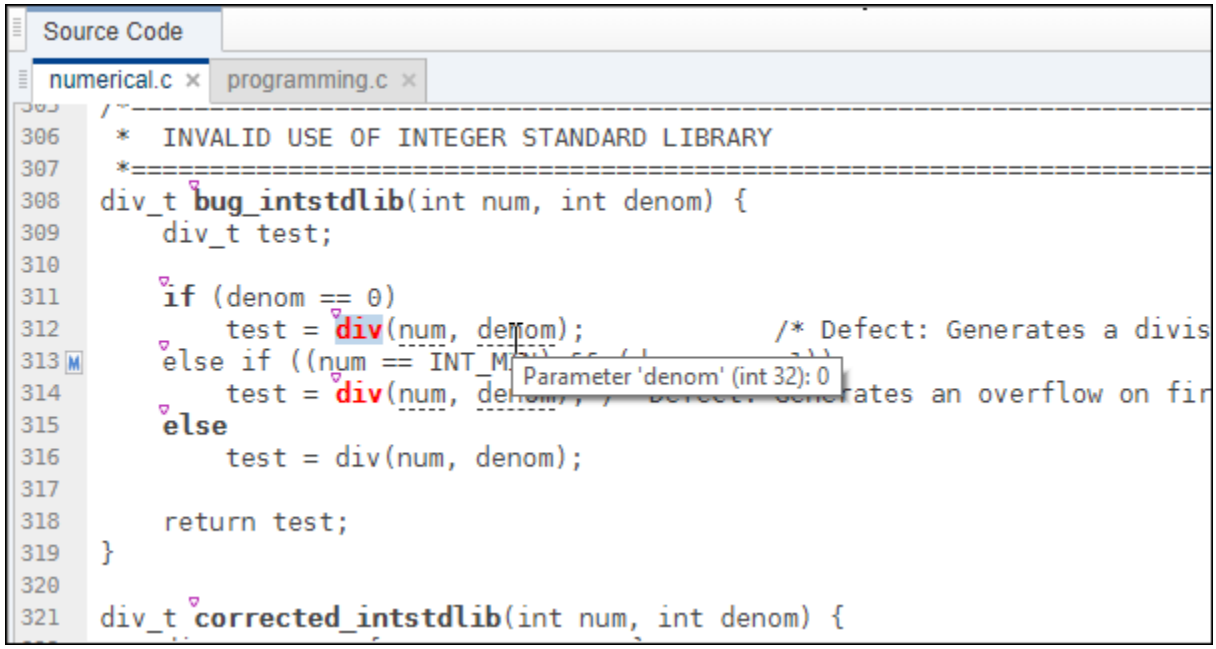
Source Code

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



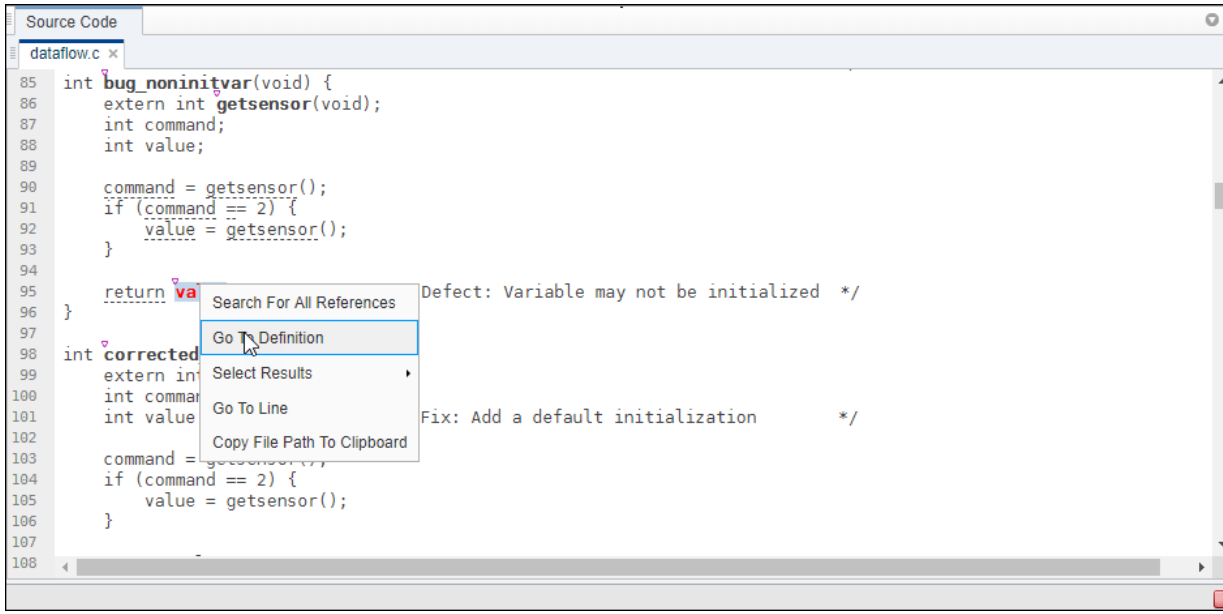
Tooltips

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



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.

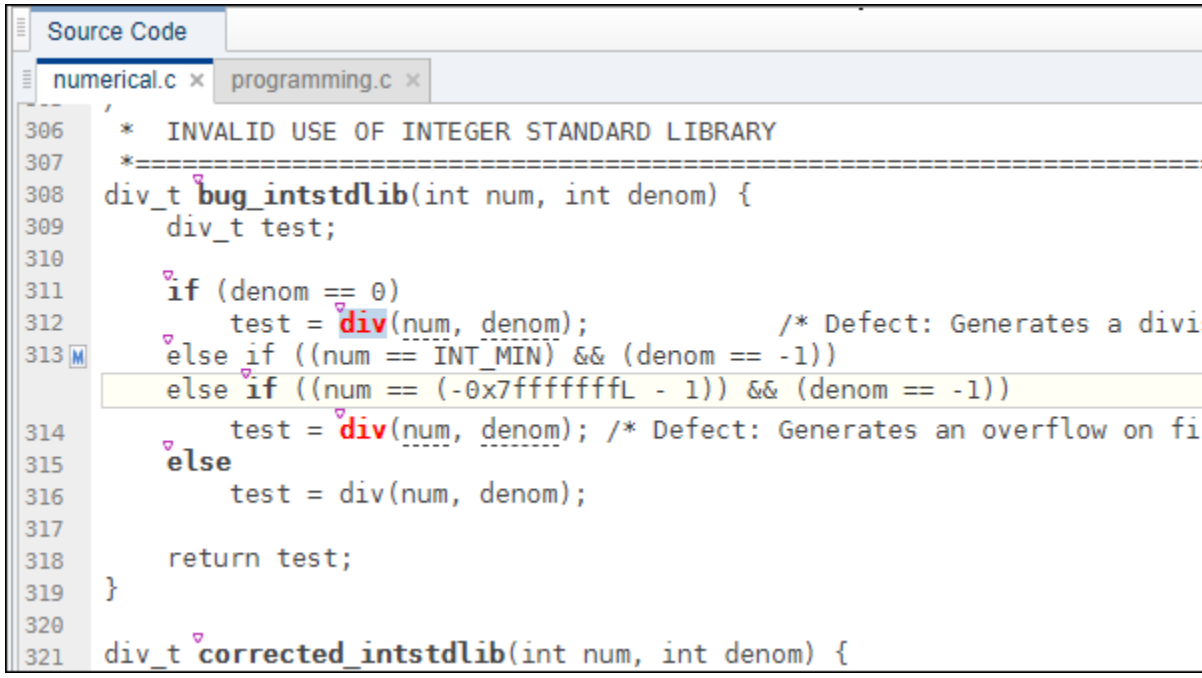
  After you navigate away from the current result, use the  icon on the **Result Details** pane to come back.
- **Go To Line** — Open the Go to line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.

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 M icons that identify source code lines with macros.

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



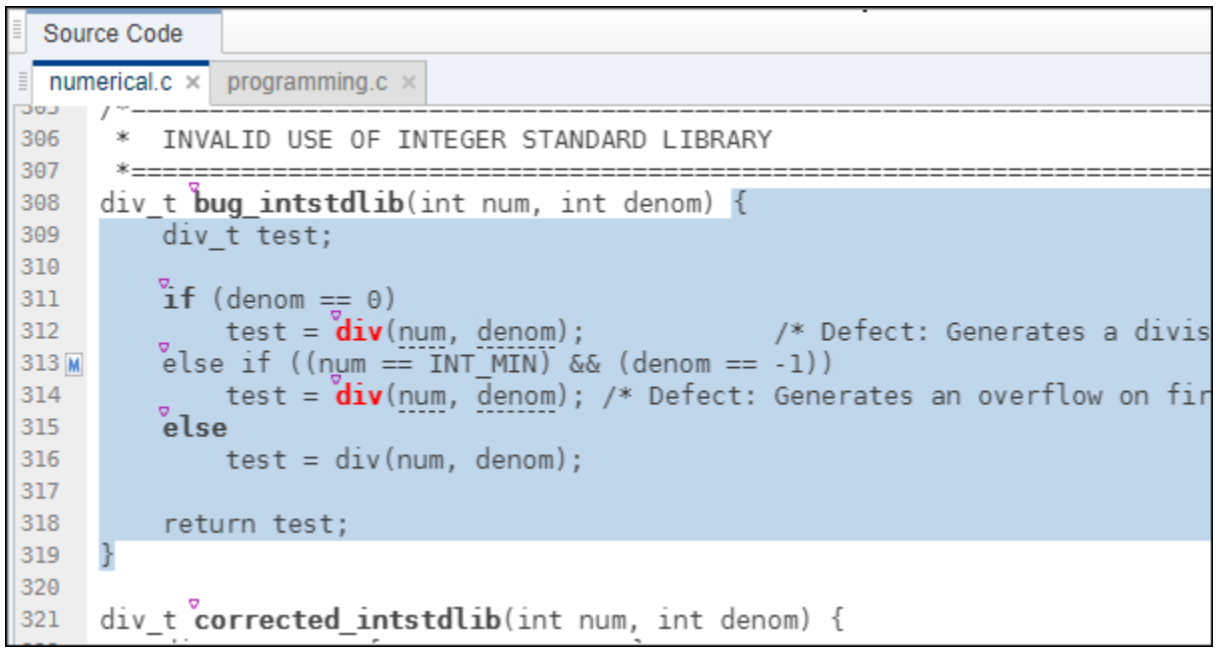
To display the normal source code again, click the icon again.

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.

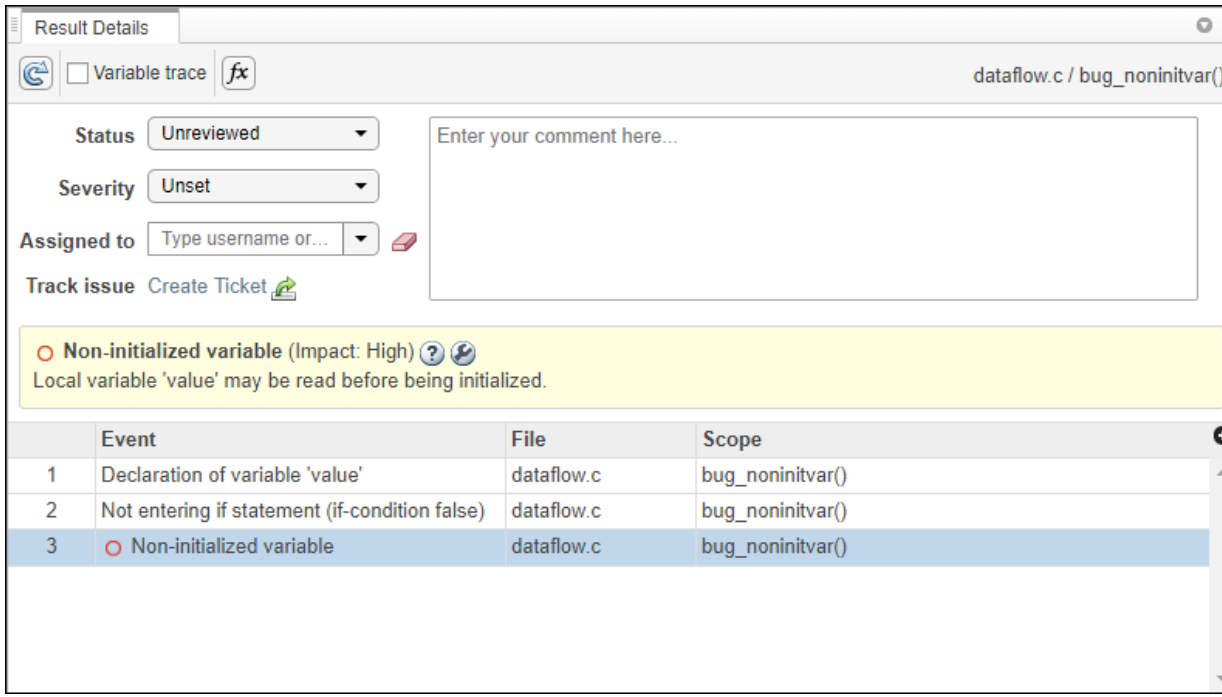


Result Details

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.



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 review to the result. When you assign results 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.

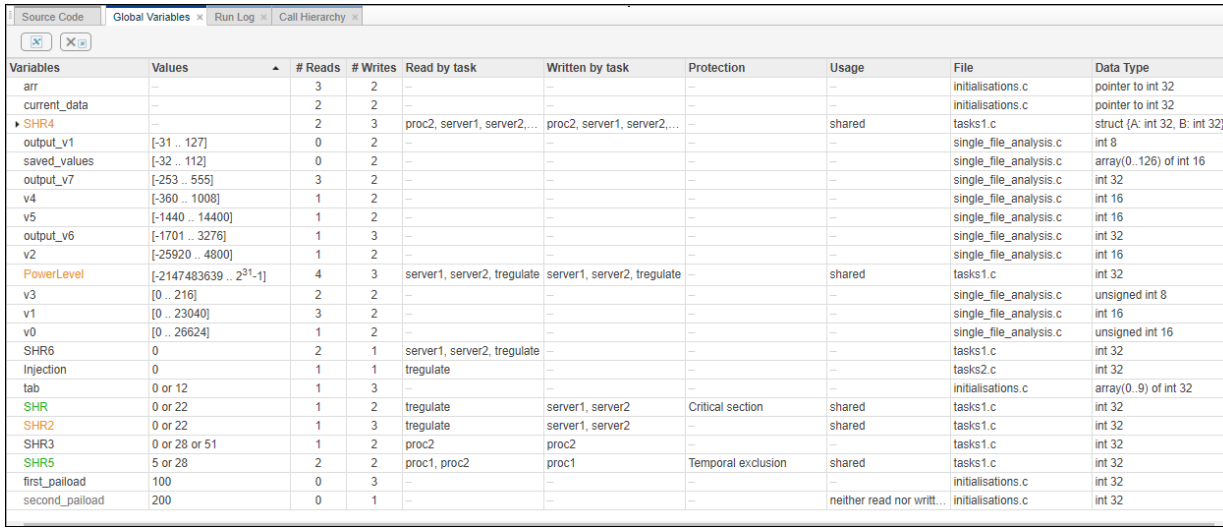
  The **Event** column lists the sequence of code instructions causing the defect. The **Scope** column lists the function containing the instructions. If the instructions are not in a function, the column lists the file containing the instructions.

  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 1-45.

Global Variables

The **Global Variables** pane displays global variables (and local static variables). For each global variable, the pane lists all functions and tasks performing read/write access on the variables, along with their attributes, such as values, read/write accesses and shared usage.



Variables	Values	# Reads	# Writes	Read by task	Written by task	Protection	Usage	File	Data Type
arr	–	3	2	–	–	–	–	initialisations.c	pointer to int 32
current_data	–	2	2	–	–	–	–	initialisations.c	pointer to int 32
SHR4	–	2	3	proc2, server1, server2,...	proc2, server1, server2,...	–	shared	tasks1.c	struct {A: int 32, B: int 32}
output_v1	[-31 .. 127]	0	2	–	–	–	–	single_file_analysis.c	int 8
saved_values	[-32 .. 112]	0	2	–	–	–	–	single_file_analysis.c	array(0..126) of int 16
output_v7	[-253 .. 555]	3	2	–	–	–	–	single_file_analysis.c	int 32
v4	[-360 .. 1008]	1	2	–	–	–	–	single_file_analysis.c	int 16
v5	[-1440 .. 14400]	1	2	–	–	–	–	single_file_analysis.c	int 16
output_v6	[-1701 .. 3276]	1	3	–	–	–	–	single_file_analysis.c	int 32
v2	[-25920 .. 4800]	1	2	–	–	–	–	single_file_analysis.c	int 16
PowerLevel	[-2147483639 .. 2^{31}-1]	4	3	server1, server2, tregulate	server1, server2, tregulate	–	shared	tasks1.c	int 32
v3	[0 .. 216]	2	2	–	–	–	–	single_file_analysis.c	unsigned int 8
v1	[0 .. 23040]	3	2	–	–	–	–	single_file_analysis.c	int 16
v0	[0 .. 26624]	1	2	–	–	–	–	single_file_analysis.c	unsigned int 16
SHR6	0	2	1	server1, server2, tregulate	–	–	–	tasks1.c	int 32
Injection	0	1	1	tregulate	–	–	–	tasks2.c	int 32
tab	0 or 12	1	3	–	–	–	–	initialisations.c	array(0..9) of int 32
SHR	0 or 22	1	2	tregulate	server1, server2	Critical section	shared	tasks1.c	int 32
SHR2	0 or 22	1	3	tregulate	server1, server2	–	shared	tasks1.c	int 32
SHR3	0 or 28 or 51	1	2	proc2	proc2	–	–	tasks1.c	int 32
SHR5	5 or 28	2	2	proc1, proc2	proc1	Temporal exclusion	shared	tasks1.c	int 32
first_paiload	100	0	3	–	–	–	–	initialisations.c	int 32
second_paiload	200	0	1	–	–	–	neither read nor writt...	initialisations.c	int 32

For each variable and each read/write access, the **Global Variables** pane contains the relevant attributes. For the variables, the various attributes are listed in this table.

Attribute	Description
Variables	Name of Variable
Values	Value (or range of values) of variable This column is empty for pointer variables.
# Reads	Number of times the variable is read
# Writes	Number of times the variable is written
Read by task	Name of tasks reading variable
Written by task	Name of tasks writing on variable

Attribute	Description
Protection	Whether shared variable is protected from concurrent access (Filled only when **Usage** column has entry, **Shared**) The possible entries in this column are: • **Critical Section**: If variable is accessed in critical section of code • **Temporal Exclusion**: If variable is accessed in mutually exclusive tasks For more details on these entries, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.
Usage	`Shared`, if variable is shared between tasks; otherwise, blank
File	Source file containing variable declaration
Data Type	Data type of variable (C/C++ data types or structures/classes)

Double-click a variable name to view read/write access operations on the variable in the

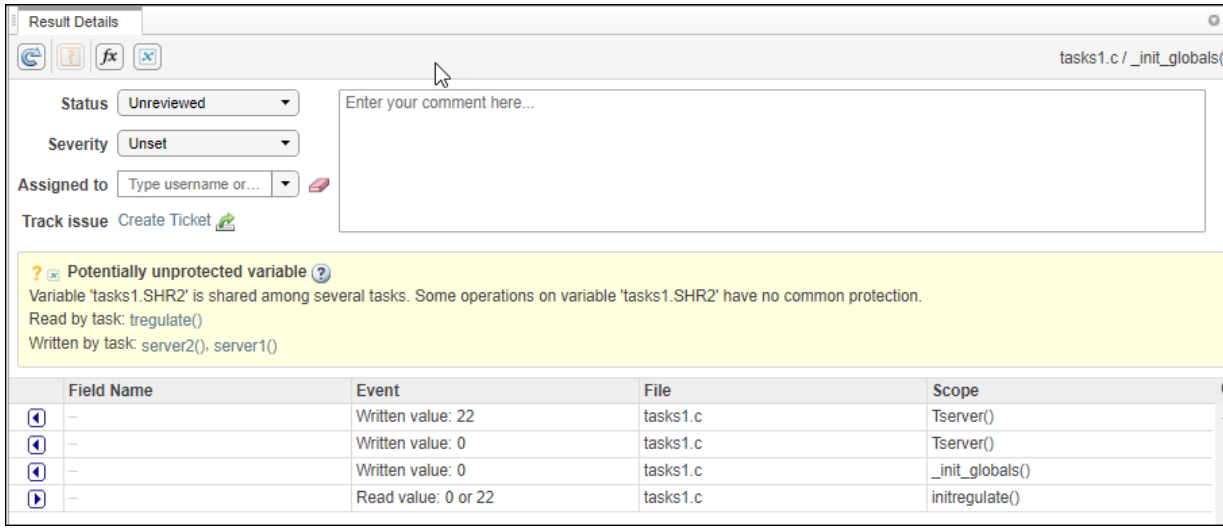
Results Details pane. The arrowhead symbols and in the **Results Details** pane indicate functions performing read and write access respectively on the global variable. For further information on tasks, see analysis option `Tasks (-entry points)` in the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

For access operations on the variables, the various attributes described in the **Global Variables**pane are listed in this table.

Attribute	Description
Values	Value or range of values of variable in the function or task performing read/write access This column is empty for pointer variables.
Written by task	*Only for tasks*: Name of task performing write access on variable
Read by task	*Only for tasks*: Name of task performing read access on variable
File	Source file containing access operation on variable

The **Results Details** pane also lists the **Scope** of the access operation on the variable.

For example, consider the global variable, SHR2:



The function, `Tserver`, in the file, `tasks1.c`, performs two write operations on SHR2. This is indicated in the **Results Details** pane by the two instances of `Tserver()` in the table, marked by . Likewise, the read access by task `initregulate` is also listed in the table and marked by .

The color scheme for variables in the **Global Variables** pane is:

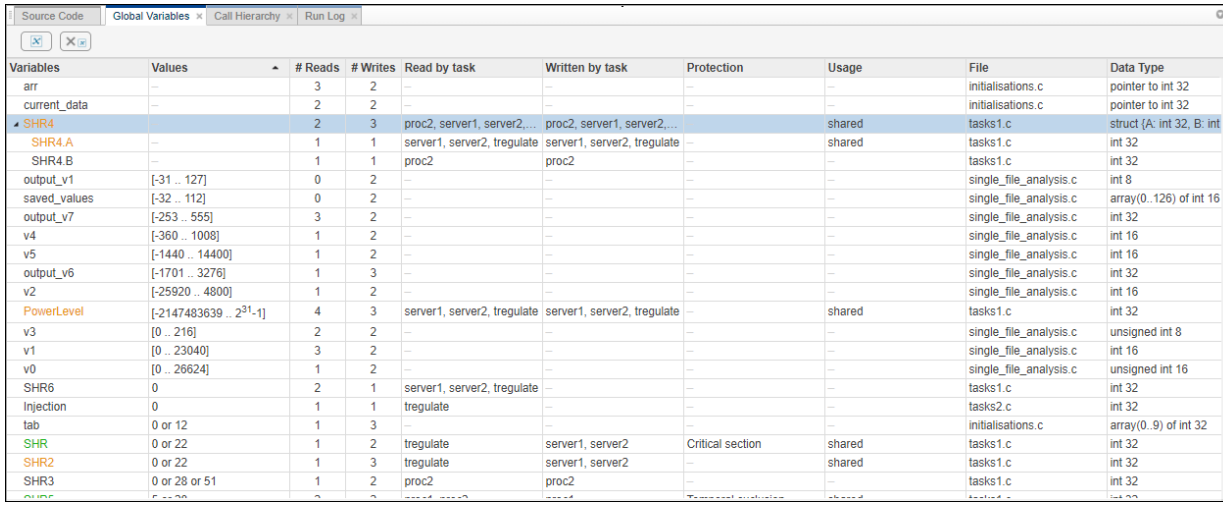
- Black: global variable.
- Orange: global variable, shared between tasks with no protection against concurrent access.
- Green: global variable, shared between tasks and protected against concurrent access.
- Gray: global variable, declared but not used in reachable code.

If a task performs certain operations on a global variable, but the operations are in unreachable code, the tasks are colored gray.

The information about global variables and read/write access operations obtained from the **Global Variables** pane is called the data dictionary.

You can also perform the following actions from the **Global Variables** pane.

- **View Structured Variables**



Variables	Values	# Reads	# Writes	Read by task	Written by task	Protection	Usage	File	Data Type
arr	–	3	2	–	–	–	–	initialisations.c	pointer to int 32
current_data	–	2	2	–	–	–	–	initialisations.c	pointer to int 32
SHR4		2	3	proc2, server1, server2,...	proc2, server1, server2,...		shared	tasks1.c	struct {A: int 32, B: int
SHR4.A	–	1	1	server1, server2, tregulate	server1, server2, tregulate	–	shared	tasks1.c	int 32
SHR4.B	–	1	1	proc2	proc2	–	–	tasks1.c	int 32
output_v1	[-31 .. 127]	0	2	–	–	–	–	single_file_analysis.c	int 8
saved_values	[-32 .. 112]	0	2	–	–	–	–	single_file_analysis.c	array(0..126) of int 16
output_v7	[-253 .. 555]	3	2	–	–	–	–	single_file_analysis.c	int 32
v4	[-360 .. 1008]	1	2	–	–	–	–	single_file_analysis.c	int 16
v5	[-1440 .. 14400]	1	2	–	–	–	–	single_file_analysis.c	int 16
output_v6	[-1701 .. 3276]	1	3	–	–	–	–	single_file_analysis.c	int 32
v2	[-25920 .. 4800]	1	2	–	–	–	–	single_file_analysis.c	int 16
PowerLevel	[-2147483639 .. 2^{31}-1]	4	3	server1, server2, tregulate	server1, server2, tregulate	–	shared	tasks1.c	int 32
v3	[0 .. 216]	2	2	–	–	–	–	single_file_analysis.c	unsigned int 8
v1	[0 .. 23040]	3	2	–	–	–	–	single_file_analysis.c	int 16
v0	[0 .. 26624]	1	2	–	–	–	–	single_file_analysis.c	unsigned int 16
SHR6	0	2	1	server1, server2, tregulate	–	–	–	tasks1.c	int 32
Injection	0	1	1	tregulate	–	–	–	tasks2.c	int 32
tab	0 or 12	1	3	–	–	–	–	initialisations.c	array(0..9) of int 32
SHR	0 or 22	1	2	tregulate	server1, server2	Critical section	shared	tasks1.c	int 32
SHR2	0 or 22	1	3	tregulate	server1, server2	–	shared	tasks1.c	int 32
SHR3	0 or 28 or 51	1	2	proc2	proc2	–	–	tasks1.c	int 32

For structured variables, double click the variable in the **Global Variables** pane to view the individual fields. For example, for the structure, `SHR4`, the pane displays the fields, `SHR4.A` and `SHR4.B`, and the functions performing read/write access on them.

- **Show/Hide Callers and Callees**

  Customize the **Global Variables** pane to show only the shared variables. On the **Global Variables** pane toolbar, click the Non-Shared Variables button to show or hide non-shared variables.

- **Hide Access in Unreachable Code**

  Hide read/write access occurring in unreachable code by clicking the filter button .

- **Limitations**

  You cannot see an addressing operation on a global variable or object (in C++) as a read/write operation in the **Global Variables** pane. For example, consider the following C++ code:

```
class C0
{
public:
  C0() {}
```

```
  int get_flag()
  {
    volatile int rd;
    return rd;
  }
  ~C0() {}
private:
  int a;                /* Never read/written */
};

C0 c0;                  /* c0 is unreachable */

int main()
{
  if (c0.get_flag())    /* Uses address of the method */
    {
      int *ptr = take_addr_of_x();
      return 1;
    }
  else
    return 0;
}
```

You do not see the method call `c0.get_flag()` in the **Global Variables** pane because the call is an addressing operation on the method belonging to the object `c0`.

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.

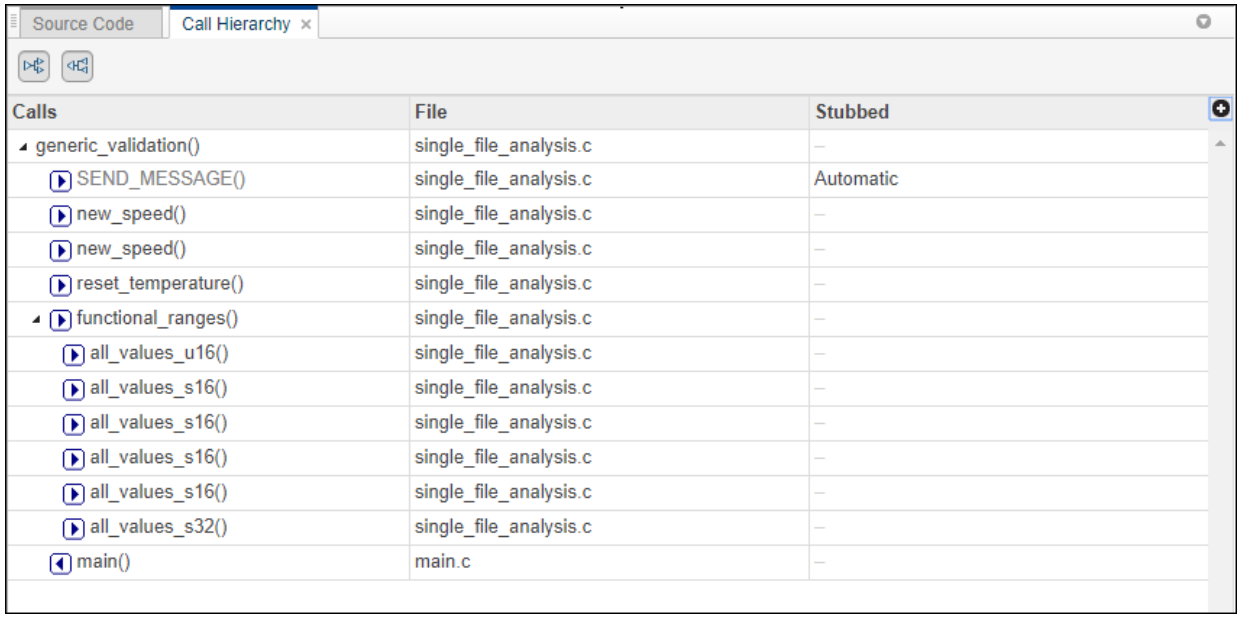
Note In Polyspace Bug Finder™ Access, you might not see all callers or callees of a function, especially for calls through function pointers and dead code.

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 result details. To update the pane:

- You can 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.



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.

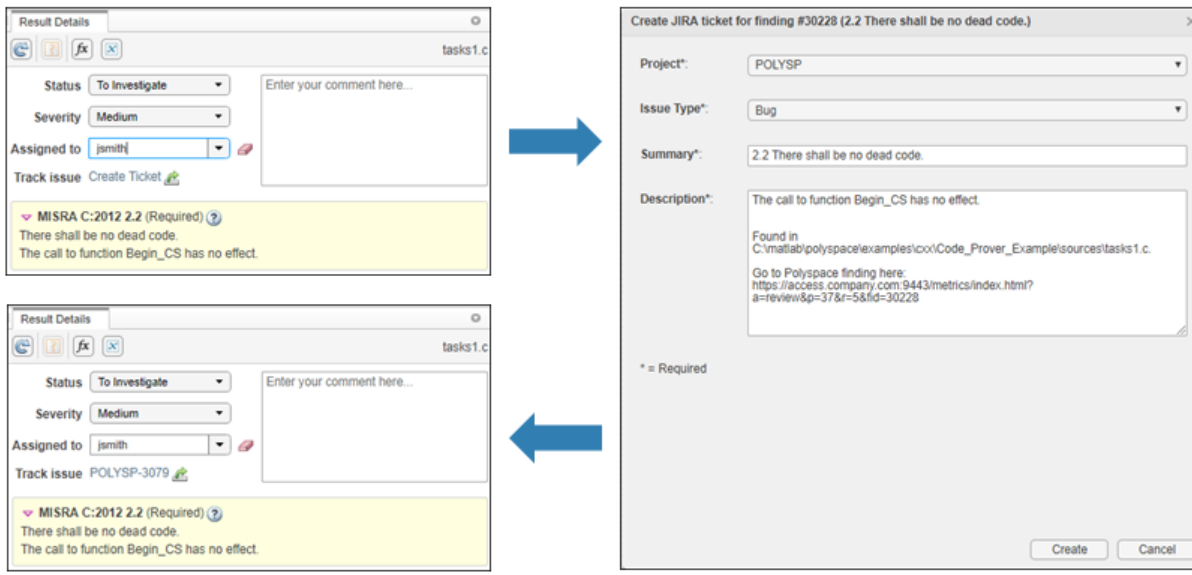
- **Automatic**: Polyspace cannot find the function definition. For instance, you did not provide the file containing the definition.
- **Std library**: The function is a standard library function. You do not provide the function definition explicitly in your Polyspace project.
- **Mapped to std library**: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

Track Issue in Bug Tracking Tool

If you use JIRA as part of your software development process, you can configure Polyspace Access to create JIRA issues for Polyspace findings and add those issues to your JIRA software project. See “Configure the Web Server and Gateway”.

To create a JIRA issue, select a finding and click **Create Ticket** from the **Results Details** pane in Polyspace Access or in the Polyspace desktop interface. In the desktop interface, you can create a JIRA issue only for results that you open from Polyspace Access.

If you are prompted to log in, use your JIRA credentials.



In the **Create JIRA ticket** window, select a **Project** and **Issue Type** from the drop-down lists. The **Description** includes a URL to the Polyspace Access **Results List** filtered down to the finding for which you created the JIRA issue.

After you create a JIRA issue, click the link in the **Results Details** pane to open the issue in JIRA and track the progress in resolving the issue.

Code Prover Analysis Following Red and Orange Checks

Polyspace considers that all execution paths that contain a run-time error terminate at the location of the error. For a given execution path, Polyspace highlights the first occurrence of a run-time error as a red or orange check and excludes that path from consideration. Therefore:

- Following a red check, Polyspace does not analyze the remaining code in the same scope as the check.
- Following an orange check, Polyspace analyzes the remaining code. But it considers only a reduced subset of execution paths that did not contain the run-time error. Therefore, if a green check occurs on an operation *after an orange check*, it means that the operation does not cause a run-time error only for this reduced set of execution paths.

  Exceptions to this behavior can occur. For instance:

  - For an orange overflow, if you specify `warn-with-wrap-around` or `allow` for `Overflow mode for signed integer (-signed-integer-overflows)` or `Overflow mode for unsigned integer (-unsigned-integer-overflows)`, Polyspace wraps the result of an overflow and does not terminate the execution paths.
  - For a subnormal float result, if you specify `warn-all` for `Subnormal detection mode (-check-subnormal)`, Polyspace does not terminate the execution paths with subnormal results.

  For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

The path containing a run-time error is terminated for the following reasons:

- The state of the program is unknown following the error. For instance, following an `Illegally dereferenced pointer` error on an operation `x=*ptr`, the value of `x` is unknown.
- You can review an error as early in your code as possible, because the first error on an execution path is shown in the verification results.
- You do not have to review and then fix or justify the same result more than once. For instance, consider these statements, where the vertical ellipsis represents code in which the variable `i` is not modified.

```
x = arr[i];
.
.
y = arr[i];
```

If an orange `Out of bounds array index` check appears on `x=arr[i]`, it means that `i` can be outside the array bounds. You do not want to review another orange check on `y=arr[i]` arising from the same cause.

Use these two rules to understand your checks. The following examples show how the two rules can result in checks that can be misleading when viewed out of context. Understand the examples below thoroughly to practice reviewing checks in context of the remaining code.

Code Following Red Check

The following example shows what happens after a red check:

```
void red(void)
{
int x;
x = 1 / x ;
x = x + 1;
}
```

When Polyspace verification reaches the division by `x`, `x` has not yet been initialized. Therefore, the software generates a red `Non-initialized local variable` check for `x`.

Execution paths beyond division by `x` are stopped. No checks are generated for the statement `x = x + 1;`.

Green Check Following Orange Check

The following example shows how a green check can result from a previous orange check. An orange check terminates execution paths that contain an error. A green check on an operation after an orange check means that the operation does not cause a run-time error only for the remaining execution paths.

```
extern int Read_An_Input(void);
void propagate(void)
{
    int x;
    int y[100];
    x = Read_An_Input();
```

```
    y[x] = 0;
    y[x] = 0;
}
```

In this function:

- `x` is assigned the return value of `Read_An_Input`. After this assignment, the software estimates the range of `x` as `[-2^31, 2^31-1]`.
- The first `y[x]=0;` shows an `Out of bounds array index` error because `x` can have negative values.
- After the first `y[x]=0;`, from the size of `y`, the software estimates `x` to be in the range `[0,99]`.
- The second `y[x]=0;` shows a green check because `x` lies in the range `[0,99]`.

Gray Check Following Orange Check

The following example shows how a gray check can result from a previous orange check.

Consider the following example:

```
extern int read_an_input(void);

void main(void)
{
    int x;
    int y[100];
    x = read_an_input();
    y[x] = 0;
    y[x-1] = (1 / x) + x ;
    if (x == 0)
        y[x] = 1;
}
```

From the gray check, you can trace backwards as follows:

- The line `y[x]=1;` is unreachable.
- Therefore, the test to assess whether `x = 0` is always false.
- The return value of `read_an_input()` is never equal to 0.

However, `read_an_input` can return any value in the full integer range, so this is not the correct explanation.

Instead, consider the execution path leading to the gray code:

- The orange **Out of bounds array index** check on `y[x]=0;` means that subsequent lines deal with `x` in `[0,99]`.
- The orange **Division by Zero** check on the division by `x` means that `x` cannot be equal to 0 on the subsequent lines. Therefore, following that line, `x` is in `[1,99]`.
- Therefore, `x` is never equal to 0 in the `if` condition. Also, the array access through `y[x-1]` shows a green check.

Red Check Following Orange Check

The following example shows how a red error can reveal a bug which occurred on previous lines.

```
%% file1.c %%

void f(int);
int read_an_input(void);

int main()  {
    int x,old_x;
    x = read_an_input();
    old_x = x;
    if (x<0 || x>10)
      return 1;
    f(x);
    x = 1 / old_x;
    // Red Division by Zero
    return 0;
}
```

```
%% file2.c %%

#include <math.h>

void f(int a) {
    int tmp;
    tmp = sqrt(0-a);
}
```

A red check occurs on `x=1/old_x;` in `file1.c` because of the following sequence of steps during verification:

1. When `x` is assigned to `old_x` in `file1.c`, the verification assumes that `x` and `old_x` have the full range of an integer, that is `[-2^31 , 2^31-1]`.
2. Following the `if` clause in `file1.c`, `x` is in `[0,10]`. Because `x` and `old_x` are equal, Polyspace considers that `old_x` is in `[0,10]` as well.
3. When `x` is passed to `f` in `file1.c`, the only possible value that `x` can have is 0. All other values lead to a run-time exception in `file2.c`, that is `tmp = sqrt(0–a);`.

4 A red error occurs on `x=1/old_x;` in `file1.c` because the software assumes `old_x` to be 0 as well.

Red Checks in Unreachable Code

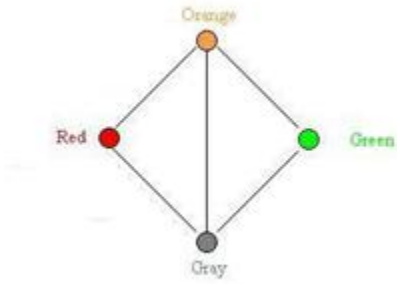
Code Prover can sometimes show red checks in code that is supposed to be unreachable and gray. When propagating variable ranges, Code Prover sometimes makes approximations. In making these approximations, Code Prover might consider an otherwise unreachable branch as reachable. If an error appears in that unreachable branch, it is colored red.

Consider the statement:

```
if (test_var == 5) {
  // Code Section
}
```

If `test_var` does not have the value 5, the `if` branch is unreachable. If Code Prover makes an approximation because of which `test_var` acquires the value 5, the branch is now reachable and can show checks of other colors.

Use this figure to understand the effect of approximations. Because of approximations, a check color that is higher up can supersede the colors below. A check that should be red or green (indicating a definite error or definite absence of error) can become orange because a variable acquires extra values that cannot occur at run time. A check that should be gray can show red, green and orange checks because Code Prover considers an unreachable branch as reachable.



See Also

Related Examples

- “Interpret Polyspace Code Prover Access Results” on page 1-3
- “Order of Code Prover Run-Time Checks” on page 1-55

Order of Code Prover Run-Time Checks

If multiple checks are performed on the same operation, Code Prover performs them in a specific order. The order of checks is important only if one of the checks is not green. If a check is red, the subsequent checks are not performed. If a check is orange, the subsequent checks are performed for a reduced set of values. For details, see “Code Prover Analysis Following Red and Orange Checks” on page 1-49.

A simple example is the order of checks on a pointer dereference. Code Prover first checks if the pointer is initialized and then checks if the pointer points to a valid location. The check `Illegally dereferenced pointer` follows the check `Non-initialized local variable`.

The order of checks can be nontrivial if one of the operands in a binary operation is a floating-point variable. Code Prover checks the operation in this order:

1. `Invalid operation on floats`: If you enable the option to consider non-finite floats, this check determines if the operation can result in NaN.
2. `Overflow`: This check determines if the result overflows.

   If you enable the option to consider non-finite floats, this check determines if the operation can result in infinities.
3. `Subnormal float`: If you enable the subnormal detection mode, this check determines if the operation can result in subnormal values.

For instance, suppose you enable options to forbid infinities, NaNs and subnormal results. In this example, the operation `y = x + 0;` is checked for all three issues. The operation appears red but consists of three checks: an orange **Invalid operation on floats**, an orange **Overflow**, and a red **Subnormal float** check.

```
#include <float.h>
#include <assert.h>

double input();

int main() {
  double x = input();
  double y;
  assert (x != x || x > DBL_MAX  || (x > 0. && x < DBL_MIN));
  y = x + 0.;
  return 0;
}
```

At the + operation, `x` can have three groups of values: `x` is NaN, `x > DBL_MAX`, and `x > 0. && x < DBL_MIN`.

The checks are performed in this order:

1. **Invalid operation on floats**: The check is orange because one execution path considers that `x` can be `NaN`.

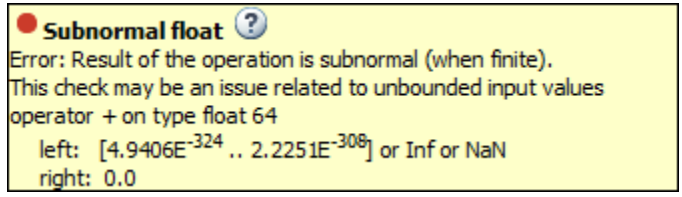
   For the next checks, this path is not considered.

2. **Overflow**: The check is orange because one group of execution paths considers that `x > DBL_MAX`. For this group of paths, the + operation can result in infinity.

   For the next check, this group of paths is not considered.

3. **Subnormal float**: On the remaining group of execution paths, `x > 0. && x < DBL_MIN`. All values of `x` cause subnormal results. Therefore, this check is red.

The message on the **Result Details** pane reflects this reduction in paths. The message for the **Subnormal float** check states `(when finite)` to show that infinite values were removed from consideration.



The values for the left and right operands reflect all values before the operation, and not the reduced set of values. Therefore, the left operand still shows `Inf` and `NaN` even though these values were not considered for the check.

See Also

More About

- “Code Prover Analysis Following Red and Orange Checks” on page 1-49

Orange Checks in Code Prover

In this section...

When Orange Checks Occur

An orange check indicates that Polyspace detects a possible run-time error but cannot prove it. A check on an operation appears orange if both conditions are true:

First condition	Second condition	Example
The operation occurs multiple times on an execution path or on multiple execution paths.	During static verification, the operation fails only a fraction of times or only on a fraction of paths.	The operation occurs in: • A loop with more than one iterations. • A function that is called more than once.
The operation involves a variable that can take multiple values.	During static verification, the operation fails only for a fraction of values.	The operation involves a `volatile` variable.

During static verification, Polyspace can consider more execution paths than the execution paths that occur during run time. If an operation fails on a subset of paths, Polyspace cannot determine if that subset actually occurs during run time. Therefore, instead of a red check, it produces an orange check on the operation.

Orange Checks from Multiple Paths

Consider this example:

```
void main() {
  func(1);
  func(0);
}

double func(int value) {
  return (1.0/value); //Orange check
}
```

`func` is called twice with two arguments. Only one of the calls results in a division by zero in the body of `func`. Code Prover shows this result as an orange **Division by zero** check.

Orange Checks from Multiple Values

Consider this example:

```
double func(int value) {
  int reducedValue = value%21 - 10; // Result in [-10,10]
  return 1.0/reducedValue; //Orange check
}
```

If the call context of `func` is unknown, Code Prover assumes that its argument `value` can take any `int` value. As a result, `reducedValue` can take any value in [-10,10]. One of these values is zero, which causes a division by zero in `func`. Code Prover shows this result as an orange **Division by zero** check.

Why Review Orange Checks

Considering a superset of actual execution paths is a sound approximation because Polyspace does not lose information. If an operation contains a run-time error, Polyspace does not produce a green check on the operation. If Polyspace cannot prove the run-time error because of approximations, it produces an orange check. Therefore, you must review orange checks.

Examples of Polyspace approximations include:

- Approximating the range of a variable at a certain point in the execution path. For instance, Polyspace can approximate the range `{-1} U [0,10]` of a `float` variable by `[-1,10]`.
- Approximating the interleaving of instructions in multitasking code. For instance, even if certain instructions in a pair of tasks cannot interrupt each other, Polyspace verification might not take that into account.

How to Review Orange Checks

To ensure that an operation does not fail during run time:

1. Determine if the execution paths on which the operation fails can actually occur.

   For more information, see “Interpret Polyspace Code Prover Access Results” on page 1-3.
2. If any of the execution paths can occur, fix the cause of the failure.
3. If the execution paths cannot occur, enter a comment in your Polyspace result or source code, describing why they cannot occur. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

   In a later verification, you can import these comments into your results. Then, if the orange check persists in the later verification, you do not have to review it again.

How to Reduce Orange Checks

Polyspace performs approximations because of one of the following.

- Your code does not contain complete information about run-time execution. For example, your code is partially developed or contains variables whose values are known only at run time.

  If you want fewer orange checks, provide the information that Polyspace requires.
- Your code is very complex. For example, there can be multiple type conversions or multiple `goto` statements.

  If you want fewer orange checks, reduce the complexity of your code and follow recommended coding practices.
- Polyspace must complete the verification in reasonable time and use reasonable computing resources.

  If you want fewer orange checks, improve the verification precision. But higher precision also increases verification time.

For more information , see *Provide Context for Verification*, *Follow Coding Rules*, and *Improve Verification Precision* in the documentation for Polyspace Code Prover.

See Also

More About

- “Managing Orange Checks” on page 1-61
- “Critical Orange Checks” on page 1-66

Managing Orange Checks

Polyspace checks every operation in your code for certain run-time errors. Therefore, you can have several orange checks in your verification results. To avoid spending unreasonable time on an orange check review, you must develop an efficient review process.

Depending on your stage of software development and quality goals, you can choose to:

- Review all red checks and critical orange checks.
- Review all red checks and all orange checks.

To see only red and critical orange checks, from the drop-down list in the left of the **Results List** pane toolbar, select **Critical checks**.

Software Development Stage

Development Stage	Situation	Review Process
Initial stage or unit development stage	In initial stages of development, you can have partially developed code or want to verify each source file independently. In that case, it is possible that: • You have not defined all your functions and class methods. • You do not have a `main` function Because of insufficient information in the code, Polyspace makes assumptions that result in many orange checks. For instance, if you use the default configuration, Polyspace assumes full range for inputs of functions that are not called in the code.	In the initial stages of development, review all red checks. For orange checks, depending on your requirements, do one of the following: • You want your partially developed code to be free of errors independent of the remaining code. For instance, you want your functions to not cause run-time errors for any input. If so, review orange checks at this stage. • You might want your partially developed code to be free of errors only in the context of the remaining code. If so, do one of the following: • Ignore orange checks at this stage. • Provide the context and then review orange checks. For instance, you can provide stubs for undefined functions

Development Stage	Situation	Review Process
		to emulate them more accurately. For more information, see *Provide Context for Verification* in the documentation for Polyspace Code Prover or Polyspace Code Prover Server.
Later stage or integration stage	In later stages of development, you have provided all your source files. However, it is possible that your code does not contain all information required for verification. For example, you have variables whose values are known only at run time.	Depending on the time you want to spend, do one of the following: • Review red checks only. • Review red and critical orange checks.

Development Stage	Situation	Review Process
Final stage	• You have provided all your source files. • You have emulated run-time environment accurately through the verification options.	Depending on the time you want to spend, do one of the following: • Review red checks and critical orange checks. • Review red checks and all orange checks. For each orange check: • If the check indicates a run-time error, fix the cause of the error. • If the check indicates a Polyspace approximation, enter a comment in your results or source code. As part of your final release process, you can have one of these criteria: • All red and critical orange checks must be reviewed and justified. • All red and orange checks must be reviewed and justified. To justify a check, assign the **Status** of `No action planned` or `Justified` to the check.

Quality Goals

For critical applications, you must review all red and orange checks.

- If an orange check indicates a run-time error, fix the cause of the error.
- If an orange check indicates a Polyspace approximation, enter a comment in your results or source code.

As part of your final release process, review and justify all red and orange checks. To justify a check, assign the **Status** of `No action planned` or `Justified` to the check.

For noncritical applications, you can choose whether or not to review the noncritical orange checks.

See Also

More About

- “Orange Checks in Code Prover” on page 1-57

Critical Orange Checks

The software identifies a subset of orange checks that are most likely run-time errors. If you select **Critical checks** from the drop-down list in the left of the **Results List** pane toolbar, you can view this subset.

These orange checks are related to path and bounded input values. Here, input values refer to values that are external to the application. Examples include:

- Inputs to functions called by generated main. For more information on functions called by generated main, see `Functions to call (-main-generator-calls)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.
- Global and volatile variables.
- Data returned by a stubbed function. The data can be the value returned by the function or a function parameter modified through a pointer.

Path

The following example shows a path-related orange check that might be identified as a potential run-time error.

Consider the following code.

```
void path(int x) {
   int result;
   result = 1 / (x - 10);
   // Orange division by zero
 }

void main() {
    path(1);
    path(10);
 }
```

The software identifies the orange ZDV check as a potential error. The **Result Details** pane indicates the potential error:

```
...
Warning: scalar division by zero may occur
...
```

This **Division by zero** check on `result=1/(x-10)` is orange because:

- `path(1)` does not cause a division by zero error.
- `path(10)` causes a division by zero error.

Polyspace indicates the definite division by zero error through a **Non-terminating call** error on `path(10)`. If you select the red check on `path(10)`, the **Result Details** pane provides the following information:

```
NTC .... Reason for the NTC: {path.x=10)
```

Bounded Input Values

Most input values can be bounded by data range specifications (DRS). The following example shows an orange check related to bounded input values that might be identified as a potential run-time error.

```
int tab[10];
extern int val;
// You specify that val is in [5..10]

void assignElement(int index) {
   int result;
   result = tab[index];
   // Orange Out of bounds array index
 }
void main(void) {
   assignElement(val);
}
```

If you specify a **PERMANENT** data range of 5 to 10 for the variable `val`, verification generates an orange **Out of bounds array index** check on `tab[index]`. The **Result Details** pane provides information about the potential error:

```
Warning: array index may be outside bounds: [0..9]
This check may be an issue related to bounded input values
Verifying DRS on extern variable val may remove this orange.
  array size: 10
  array index value: [5 .. 10]
```

Unbounded Input Values

The following example shows an orange check related to unbounded input values that might be identified as a potential run-time error:

```
int tab[10];
extern int val;

void assignElement(int index) {
   int result;
   result = tab[index];
   // Orange Out of bounds array index
 }
void main(void) {
   assignElement(val);
}
```

The verification generates an orange **Out of bounds array index** check on `tab[index]`. The **Result Details** pane provides information about the potential error:

```
Warning: array index may be outside bounds: [0..9]
This check may be an issue related to unbounded input values
If appropriate, applying DRS to extern variable val may remove this orange.
 array size: 10
 array index value: [-2^31 .. 2^31-1]
```

Software Quality Objectives

The Software Quality Objectives or SQOs are a set of thresholds against which you can compare your verification results. You can develop a review process based on the Software Quality Objectives. In your review process, you consider only those results that cause your project to fail a certain SQO level.

You can use a predefined SQO level or define your own SQOs. Following are the quality thresholds specified by each predefined SQO.

SQO 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)` • *n1* — Number of different operators • *N1* — Total number of operators • *n2* — Number of different operands • *N2* — Total number of operands	4
Number of recursions	0

Metric	Threshold Value
Number of direct recursions	0
Number of unjustified violations of the following MISRA C:2004 rules: • 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: • 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

Metric	Threshold Value
Number of unjustified violations of the following MISRA C++ rules: • 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

SQO Level 2

In addition to all the requirements of SQO Level 1, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified red checks	0
Number of unjustified `Non-terminating call` and `Non-terminating loop` checks	0

SQO Level 3

In addition to all the requirements of SQO Level 2, this level includes the following thresholds:

Metric	Threshold Value
Number of unjustified gray `Unreachable code` checks	0

SQO Level 4

In addition to all the requirements of SQO Level 3, this level includes the following thresholds:

Metric	Threshold Value
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	`Invalid C++ specific operations`: 50
	`Correctness condition`: 60
	`Division by zero`: 80
	`Uncaught exception`: 50
	`Function not returning value`: 80
	`Illegally dereferenced pointer`: 60
	`Return value not initialized`: 80
	`Non-initialized local variable`: 80
	`Non-initialized pointer`: 60
	`Non-initialized variable`: 60
	`Null this-pointer calling method`: 50
	`Incorrect object oriented programming`: 50
	`Out of bounds array index`: 80
	`Overflow`: 60
	`Invalid shift operations`: 80
	`User assertion`: 60

SQO Level 5

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

Metric	Threshold Value
Number of unjustified violations of the following MISRA C:2004 rules: • 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: • 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: • 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
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	`Invalid C++ specific operations`: 70
	`Correctness condition`: 80
	`Division by zero`: 90
	`Uncaught exception`: 70
	`Function not returning value`: 90
	`Illegally dereferenced pointer`: 70
	`Return value not initialized`: 90
	`Non-initialized local variable`: 90
	`Non-initialized pointer`: 70
	`Non-initialized variable`: 70
	`Null this-pointer calling method`: 70
	`Incorrect object oriented programming`: 70
	`Out of bounds array index`: 90
	`Overflow`: 80

Metric	Threshold Value
	`Invalid shift operations`: 90
	`User assertion`: 80

SQO Level 6

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

Metric	Threshold Value
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	`Invalid C++ specific operations`: 90
	`Correctness condition`: 100
	`Division by zero`: 100
	`Uncaught exception`: 90
	`Function not returning value`: 100
	`Illegally dereferenced pointer`: 80
	`Return value not initialized`: 100
	`Non-initialized local variable`: 100
	`Non-initialized pointer`: 80
	`Non-initialized variable`: 80
	`Null this-pointer calling method`: 90
	`Incorrect object oriented programming`: 90
	`Out of bounds array index`: 100
	`Overflow`: 100
	`Invalid shift operations`: 100
	`User assertion`: 100

Exhaustive

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

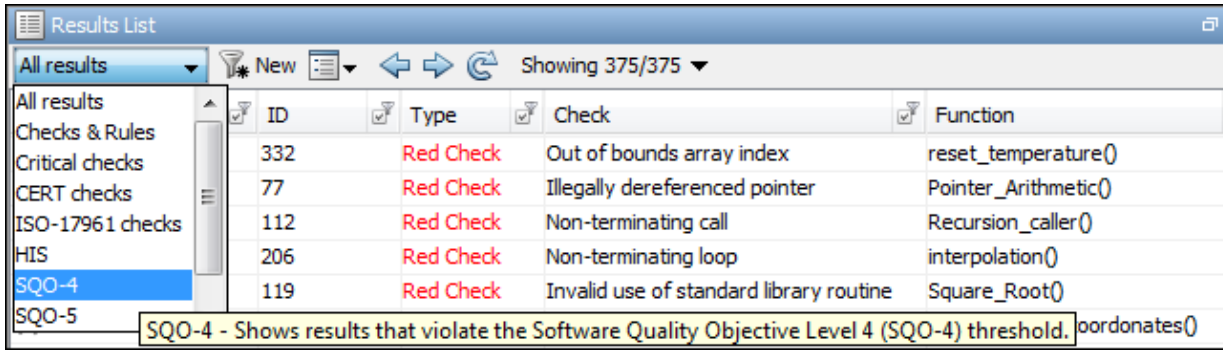
Metric	Threshold Value
Number of unjustified MISRA C and MISRA C++ coding rule violations	0
Number of unjustified red checks	0
Number of unjustified `Non-terminating call` and `Non-terminating loop` checks	0
Number of unjustified gray `Unreachable code` checks	0
Percentage of justified orange checks, calculated as the number of green and justified orange checks divided by the total number of green and orange checks.	100

For information on the rationales behind these levels, see Software Quality Objectives for Source Code.

Comparing Verification Results Against Software Quality Objectives

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

- 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**.

- In the Polyspace Metrics web interface, you can first determine whether your project fails to attain a certain Software Quality Objective. The web interface generates a **Quality Status** of **PASS** or **FAIL** for your project. If your project has a **Quality Status** of **FAIL**, the web interface highlights in red those results that you must fix or justify to attain the Software Quality Objective. You can choose to only download those results to the Polyspace user interface and review them. For more information, see “Quality Objectives Dashboard” on page 1-28.

  You can also generated reports that show the **PASS** or **FAIL** status using the templates `SoftwareQualityObjectives_Summary` and `SoftwareQualityObjectives`. See `Bug Finder and Code Prover report (-report-template)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Note You cannot use the menu in the user interface to suppress red or gray checks. Therefore, you cannot directly compare your project against predefined SQO levels 1, 2 and 3 in the Polyspace user interface. However, in the Polyspace Access web interface, you can compare your project against all predefined SQO levels.

Software Quality Objective Subsets (C:2004)

In this section...

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 *static* 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 *for* statement shall not contain any objects of floating type.
13.5	The three expressions of a *for* statement shall be concerned only with loop control.
14.4	The *goto* statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.

Rule number	Description
16.2	Functions shall not call themselves, either directly or indirectly.
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.
20.4	Dynamic heap memory allocation shall not be used.

Note Polyspace software does not check MISRA rule **18.3**.

Rules in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
6.3	*typedefs* that indicate size and signedness should be used in place of the basic types
8.7	Objects shall be defined at block scope if they are only accessed from within a single function

Rule number	Description
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression
10.5	Bitwise operations shall not be performed on signed integer types
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
12.1	Limited dependence should be placed on C's operator precedence rules in expressions
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits
12.5	The operands of a logical && or || shall be primary-expressions
12.6	Operands of logical operators (&&, || and !) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, || or !)
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
12.10	The comma operator shall not be used

Rule number	Description
12.12	The underlying bit representations of floating-point values shall not be used.
13.1	Assignment operators shall not be used in expressions that yield Boolean values
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a *for* statement shall not contain any objects of floating type.
13.5	The three expressions of a *for* statement shall be concerned only with loop control.
13.6	Numeric variables being used within a "*for*" loop for iteration counting should not be modified in the body of the loop
14.4	The *goto* 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 *switch, while, do while* or *for* statement shall be a compound statement
14.10	All *if else if* constructs should contain a final *else* clause
15.3	The final clause of a *switch* statement shall be the *default* clause
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty

Rule number	Description
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4	Array indexing shall be the only allowed form of pointer arithmetic.
17.5	The declaration of objects should contain no more than 2 levels of pointer indirection.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.3	An area of memory shall not be reused for unrelated purposes.
18.4	Unions shall not be used.
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.
20.4	Dynamic heap memory allocation shall not be used.

Note Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \
return -1 else return 0; }
```

See Also

More About

- “Interpret Polyspace Code Prover Access Results” on page 1-3

Software Quality Objective Subsets (AC AGC)

In this section...

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 *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

Rules in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
6.3	*typedefs* 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 *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
11.5	Type casting from any type to or from pointers shall not be used
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits

Rule number	Description
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned
12.10	The comma operator shall not be used
12.12	The underlying bit representations of floating-point values shall not be used.
14.7	A function shall have a single point of exit at the end of the function.
16.1	Functions shall not be defined with variable numbers of arguments.
16.2	Functions shall not call themselves, either directly or indirectly.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.6	The address of an object with automatic storage shall not be assigned to an object that may persist after the object has ceased to exist.
18.4	Unions shall not be used.
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.
20.3	The validity of values passed to library functions shall be checked.

Note Polyspace software does not check MISRA rule **20.3** directly.

However, you can check this rule by writing manual stubs that check the validity of values. For example, the following code checks the validity of an input being greater than 1:

```
int my_system_library_call(int in) {assert (in>1); if random \
return -1 else return 0; }
```

For more information about these rules, see *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

See Also

More About

- “Interpret Polyspace Code Prover Access Results” on page 1-3

Software Quality Objective Subsets (C:2012)

In this section...

These subsets of MISRA C:2012 guidelines can have a direct or indirect impact on the precision of your Polyspace results. When you set up coding rules checking, you can select these subsets.

Guidelines in SQO-Subset1

The following set of MISRA C:2012 coding guidelines typically reduces the number of unproven results in Polyspace Code Prover.

Rule	Description
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage
8.11	When an array with external linkage is declared, its size should be explicitly specified
8.13	A pointer should point to a const-qualified type whenever possible
11.1	Conversions shall not be performed between a pointer to a function and any other type
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type
11.4	A conversion should not be performed between a pointer to object and an integer type
11.5	A conversion should not be performed from pointer to void into pointer to object
11.6	A cast shall not be performed between pointer to void and an arithmetic type
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type
14.1	A loop counter shall not have essentially floating type

Rule	Description
14.2	A for loop shall be well-formed
15.1	The goto statement should not be used
15.2	The goto statement shall jump to a label declared later in the same function
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement
15.5	A function should have a single point of exit at the end
17.1	The features of <starg.h> shall not be used
17.2	Functions shall not call themselves, either directly or indirectly
18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type
18.5	Declarations should contain no more than two levels of pointer nesting
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist
19.2	The union keyword should not be used
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used

Guidelines in SQO-Subset2

Good design practices generally lead to less code complexity, which can reduce the number of unproven results in Polyspace Code Prover. The following set of coding rules enforce good design practices. The `SQO-subset2` option checks the rules in `SQO-subset1` and some additional rules.

Rule	Description
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage
8.11	When an array with external linkage is declared, its size should be explicitly specified

Rule	Description
8.13	A pointer should point to a const-qualified type whenever possible
11.1	Conversions shall not be performed between a pointer to a function and any other type
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type
11.4	A conversion should not be performed between a pointer to object and an integer type
11.5	A conversion should not be performed from pointer to void into pointer to object
11.6	A cast shall not be performed between pointer to void and an arithmetic type
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer
12.1	The precedence of operators within expressions should be made explicit
12.3	The comma operator should not be used
13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders
13.4	The result of an assignment operator should not be used
14.1	A loop counter shall not have essentially floating type
14.2	A for loop shall be well-formed
14.4	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type
15.1	The goto statement should not be used
15.2	The goto statement shall jump to a label declared later in the same function
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement
15.5	A function should have a single point of exit at the end

Rule	Description
15.6	The body of an iteration- statement or a selection- statement shall be a compound- statement
15.7	All if ... else if constructs shall be terminated with an else statement
16.4	Every switch statement shall have a default label
16.5	A default label shall appear as either the first or the last switch label of a switch statement
17.1	The features of <starg.h> shall not be used
17.2	Functions shall not call themselves, either directly or indirectly
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type
18.5	Declarations should contain no more than two levels of pointer nesting
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist
19.2	The union keyword should not be used
20.4	A macro shall not be defined with the same name as a keyword
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses
20.9	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation
20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used

See Also

More About

- “Interpret Polyspace Code Prover Access Results” on page 1-3

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

The following set of MISRA C++ coding rules will typically improve the number of unproven results in Polyspace Code Prover.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	The One Definition Rule shall not be violated.
3-9-3	The underlying bit representations of floating-point values shall not be used.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.

MISRA C++ Rule	Description
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.

MISRA C++ Rule	Description
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
18-4-1	Dynamic heap memory allocation shall not be used.

SQO Subset 2 – Indirect Impact on Selectivity

Good design practices generally lead to less code complexity, which can improve the number of unproven results in Polyspace Code Prover. The following set of coding rules may help to address design issues in your code. The `SQO-subset2` option checks the rules in `SQO-subset1` and `SQO-subset2`.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.
3-9-3	The underlying bit representations of floating-point values shall not be used.

MISRA C++ Rule	Description
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.
5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.
5-0-13	The condition of an if-statement and the condition of an iteration- statement shall have type bool
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-1	Each operand of a logical && or || shall be a postfix - expression.
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.

MISRA C++ Rule	Description
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
5-2-11	The comma operator, && operator and the || operator shall not be overloaded.
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3	The unary & operator shall not be overloaded.
5-18-1	The comma operator shall not be used.
6-2-1	Assignment operators shall not be used in sub-expressions.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
6-4-2	All if ... else if constructs shall be terminated with an else clause.
6-4-6	The final clause of a switch statement shall be the default-clause.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.
6-6-2	The goto statement shall jump to a label declared later in the same function body.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.
6-6-5	A function shall have a single point of exit at the end of the function.
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

MISRA C++ Rule	Description
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-4	Functions should not call themselves, either directly or indirectly.
8-4-1	Functions shall not be defined using the ellipsis notation.
8-4-3	All exit paths from a function with non- void return type shall have an explicit return statement with an expression.
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.
8-5-2	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
9-5-1	Unions shall not be used.
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.
11-0-1	Member data in non- POD class types shall be private.
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.

MISRA C++ Rule	Description
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-5	A class type exception shall always be caught by reference.
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.
15-5-1	A class destructor shall not exit with an exception.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.
18-4-1	Dynamic heap memory allocation shall not be used.

See Also

More About

- “Interpret Polyspace Code Prover Access Results” on page 1-3

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 on analysis options, see the documentation of Polyspace Code Prover or Polyspace Code Prover Server .

Argument	Purpose
`single-unit-rules`	Check rules that apply only to single translation units. If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the compilation phase.
`system-decidable-rules`	Check rules in the `single-unit-rules` subset and some rules that apply to the collective set of program files. The additional rules are the less complex rules that apply at the integration level. These rules can be checked only at the integration level because the rules involve more than one translation unit. If you detect only coding rule violations and select this subset, a Bug Finder analysis stops after the linking phase.

See also “Interpret Polyspace Code Prover Access Results” on page 1-3.

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	`typedef`s that indicate size and signedness should be used in place of the basic types.
6.4	Bit fields shall only be defined to be of type `unsigned int` or `signed int`.
6.5	Bit fields of type `signed int` shall be at least 2 bits long.

Constants

Rule	Description
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and Definitions

Rule	Description
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
8.4*	If objects or functions are declared more than once their types shall be compatible.
8.5	There shall be no definitions of objects or functions in a header file.
8.6	Functions shall always be declared at file scope.
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.
8.8*	An external object or function shall be declared in one file and only one file.
8.9*	An identifier with external linkage shall have exactly one external definition.
8.10*	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
8.11	The `static` 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 = 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: • 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 • 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 ~ and << are applied to an operand of underlying type `unsigned char` or `unsigned short`, 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 `unsigned` 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 `void`.
11.3	A cast should not be performed between a pointer type and an integral type.
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.
11.5	A cast shall not be performed that removes any `const` or `volatile` 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 `sizeof` operator should not be used on expressions that contain side effects.
12.5	The operands of a logical && or `||` shall be primary-expressions.
12.6	Operands of logical operators (&&, `||` and `!`) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, `||` or `!`).
12.7	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 (++) and decrement (`--`) 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 `for` statement shall not contain any objects of floating type.
13.5	The three expressions of a `for` statement shall be concerned only with loop control.
13.6	Numeric variables being used within a `for` 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 • have at least one side effect however executed, or • cause control flow to change.
14.4	The `goto` statement shall not be used.
14.5	The `continue` statement shall not be used.
14.6	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.
14.8	The statement forming the body of a `switch`, `while`, `do while` or `for` statement shall be a compound statement.
14.9	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 `if else if` constructs should contain a final `else` clause.

Switch Statements

Rule	Description
15.0	Unreachable code is detected between `switch` statement and first `case`.
15.1	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 `break` statement shall terminate every non-empty `switch` clause.
15.3	The final clause of a `switch` statement shall be the `default` clause.
15.4	A `switch` expression should not represent a value that is effectively Boolean.
15.5	Every `switch` statement shall have at least one `case` 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 `void`.
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 &, 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	`#include` 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 `#include` directives.
19.3	The `#include` directive shall be followed by either a <filename> 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 `#define`d and `#undef`d within a block.
19.6	`#undef` 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 # 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.
19.13	The # and ## preprocessor operators should not be used.
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

Rule	Description
19.17	All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` 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 `errno` shall not be used.
20.6	The macro `offsetof`, in library `<stddef.h>`, shall not be used.
20.7	The `setjmp` macro and the `longjmp` function shall not be used.
20.8	The signal handling facilities of `<signal.h>` shall not be used.
20.9	The input/output library `<stdio.h>` shall not be used in production code.
20.10	The library functions `atof`, `atoi` and `atoll` from library `<stdlib.h>` shall not be used.
20.11	The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.
20.12	The time handling functions of library `<time.h>` 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 `/*` and `//` shall not be used within a comment.
3.2	Line-splicing shall not be used in `//` comments.

Character Sets and Lexical Conventions

Rule	Description
4.1	Octal and hexadecimal escape sequences shall be terminated.
4.2	Trigraphs should not be used.

Identifiers

Rule	Description
5.1*	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 `static` 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 `static` 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 `restrict` type qualifier shall not be used.

Initialization

Rule	Description
9.2	The initializer for an aggregate or union shall be enclosed in braces.
9.3	Arrays shall not be partially initialized.
9.4	An element of an object shall not be initialized more than once.
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The Essential Type Model

Rule	Description
10.1	Operands shall not be of an inappropriate essential type.
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
10.5	The value of an expression should not be cast to an inappropriate essential type.
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer Type Conversion

Rule	Description
11.1	Conversions shall not be performed between a pointer to a function and any other type.
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type.
11.4	A conversion should not be performed between a pointer to object and an integer type.
11.5	A conversion should not be performed from pointer to void into pointer to object.
11.6	A cast shall not be performed between pointer to void and an arithmetic type.
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
11.9	The macro `NULL` shall be the only permitted form of integer null pointer constant.

Expressions

Rule	Description
12.1	The precedence of operators within expressions should be made explicit.
12.3	The comma operator should not be used.
12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side Effects

Rule	Description
13.3	A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.
13.4	The result of an assignment operator should not be used.
13.6	The operand of the `sizeof` operator shall not contain any expression which has potential side effects.

Control Statement Expressions

Rule	Description
14.4	The controlling expression of an `if` statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control Flow

Rule	Description
15.1	The `goto` statement should not be used.
15.2	The `goto` statement shall jump to a label declared later in the same function.
15.3	Any label referenced by a `goto` statement shall be declared in the same block, or in any block enclosing the `goto` statement.
15.4	There should be no more than one `break` or `goto` statement used to terminate any iteration statement.
15.5	A function should have a single point of exit at the end
15.6	The body of an iteration-statement or a selection-statement shall be a compound statement.
15.7	All `if … else if` constructs shall be terminated with an `else` statement.

Switch Statements

Rule	Description
16.1	All `switch` statements shall be well-formed.
16.2	A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.
16.3	An unconditional `break` statement shall terminate every `switch`-clause.
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.
16.6	Every `switch` statement shall have at least two `switch`-clauses.
16.7	A `switch`-expression shall not have essentially Boolean type.

Functions

Rule	Description
17.1	The features of `<starg.h>` shall not be used.
17.3	A function shall not be declared implicitly.
17.4	All exit paths from a function with non-`void` return type shall have an explicit return statement with an expression.
17.6	The declaration of an array parameter shall not contain the `static` keyword between the `[ ]`.
17.7	The value returned by a function having non-`void` return type shall be used.

Pointers and Arrays

Rule	Description
18.4	The `+`, `-`, `+=` and `-=` operators should not be applied to an expression of pointer type.
18.5	Declarations should contain no more than two levels of pointer nesting.
18.7	Flexible array members shall not be declared.
18.8	Variable-length array types shall not be used.

Overlapping Storage

Rule	Description
19.2	The `union` keyword should not be used.

Preprocessing Directives

Rule	Description
20.1	`#include` directives should only be preceded by preprocessor directives or comments.
20.2	The `'`, `"`, or `\` characters and the `/*` or `//` character sequences shall not occur in a header file name.
20.3	The `#include` directive shall be followed by either a <filename> or \"filename \" sequence.
20.4	A macro shall not be defined with the same name as a keyword.
20.5	`#undef` 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 `#if` or `#elif` preprocessing directive shall evaluate to 0 or 1.
20.9	All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be `#define`'d before evaluation.
20.10	The # and ## preprocessor operators should not be used.
20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.
20.12	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
20.13	A line whose first token is # shall be a valid preprocessing directive.
20.14	All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related.

Standard Libraries

Rule	Description
21.1	`#define` and `#undef` 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 `<stdlib.h>` shall not be used.
21.4	The standard header file `<setjmp.h>` shall not be used.
21.5	The standard header file `<signal.h>` shall not be used.
21.6	The Standard Library input/output functions shall not be used.
21.7	The `atof`, `atoi`, `atol`, and `atoll` functions of `<stdlib.h>` shall not be used.
21.8	The library functions `abort`, `exit`, `getenv` and `system` of `<stdlib.h>` shall not be used.
21.9	The library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used.
21.10	The Standard Library time and date functions shall not be used.
21.11	The standard header file `<tgmath.h>` shall not be used.
21.12	The exception handling features of `<fenv.h>` 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 Polyspace Code Prover Access Results” on page 1-3

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

Metric	Recommended Upper Limit
Number of return statements	1

See Also

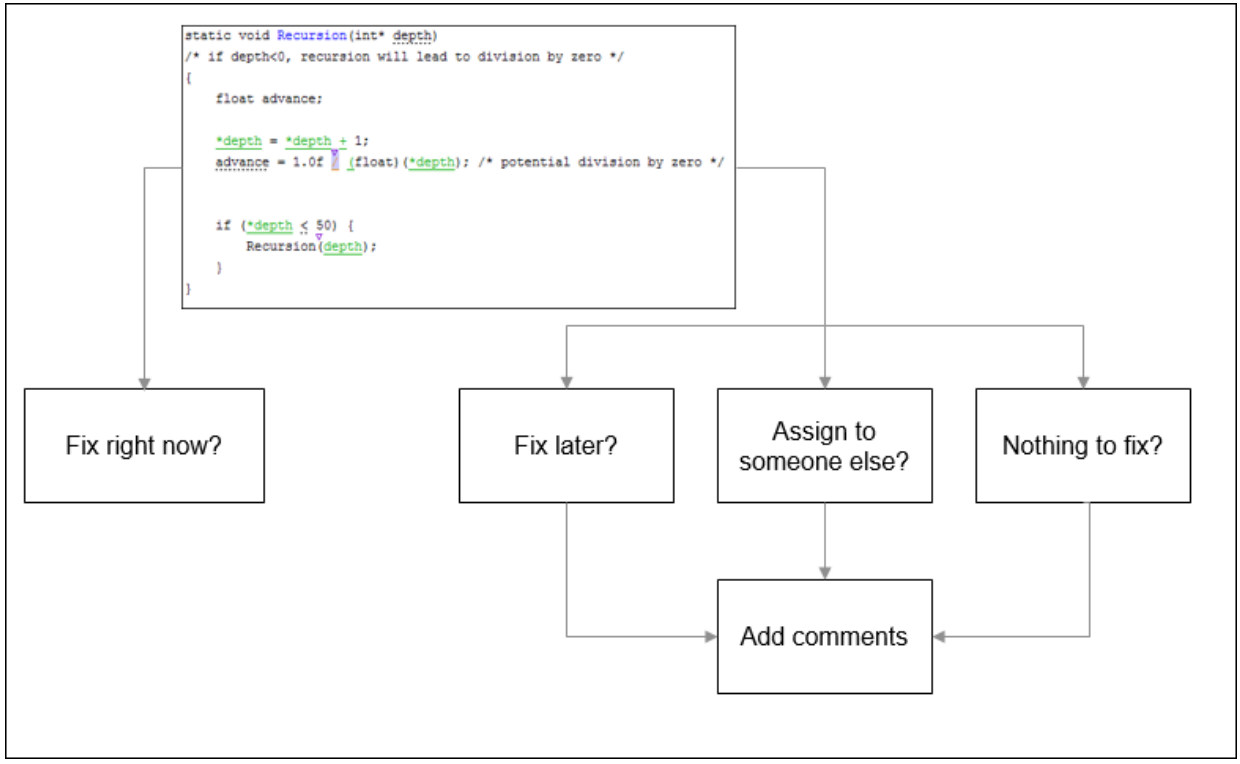
More About

- “Code Metrics”

Fix or Comment Polyspace Results

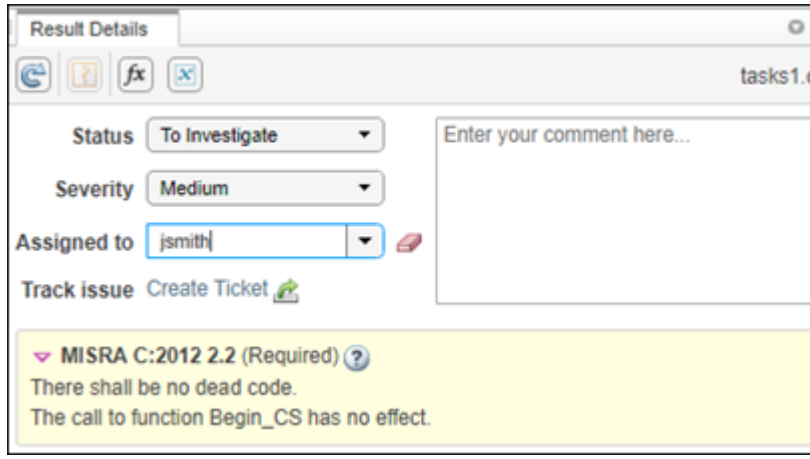
Address Polyspace Results Through Bug Fixes or Comments

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



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

Comment in Result Details pane



Set the **Status**, **Severity**, and comment fields in the **Result Details** pane. The status indicates your response to the Polyspace result. 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).

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 “Annotate Code and Hide Known or Acceptable 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

- “Annotate Code and Hide Known or Acceptable Results” on page 2-5

Annotate Code and Hide Known or Acceptable Results

To facilitate your review workflow, Polyspace Access classifies analysis findings as **To Do**, **In Progress**, or **Done**. In the **DASHBOARD** perspective, open issues are findings that are **To Do** or **In Progress**.

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

Add annotations by typing them directly in your code. For the general workflow, see “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2. This topic shows the annotation syntax. If you annotate a finding in your code, you cannot edit the status, severity, or comment fields in the Polyspace Access interface.

Code Annotation Syntax

To add comments directly to your source file, use the Polyspace annotation syntax. The syntax is not case sensitive, and has this format:

- Annotation for current line of code:

  ```
  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`*, they are ignored.

For example, in this code, the annotation on line 9 is ignored and the block annotation is applied instead.

```
1 /*polyspace-begin MISRA-C:14.9 [High:To fix] "Block annotation"*/
2
3 int main(void) /*polyspace MISRA-C:14.7 "Annotation applied"*/
4 {
5     int x = 1;
6     int y = x / 2;
7
8
9     if (x > y) /*polyspace MISRA-C:14.9 "Annotation ignored"*/
1         return x;
11     return x;
12 }
13 /*polyspace-end MISRA-C:14.9 [High:To fix] "Block annotation"*/
```

If you apply an annotation to multiple lines of code, the annotation does not apply to green checks in the code. When you rerun the analysis these green checks are not considered justified, and their *`Status`* and *`Severity`* in the **Results List** do not change to the *`Status`* and *`Severity`* of the 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
Family	Type of analysis result: • 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 To specify all analysis results, use the asterisk character *:*.

Field	Allowed Value
Result_name	For DEFECT, use short names of checkers. See “Short Names of Bug Finder Defect Checkers” (Polyspace Bug Finder Access). For RTE, use short names of run-time checks. See “Short Names of Code Prover Run-Time Checks” on page 2-12. For CODE-METRICS, use short names of code complexity metrics. See “Short Names of Code Complexity Metrics” on page 2-14. For VARIABLE, the only allowed value is the asterisk character " * ". For coding standard violations, specify the rule number or numbers. To specify all parts of a result name [MISRA2012:17.*] or all result names in a family [DEFECT:*], use the asterisk character.
Status	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: • Unreviewed • To investigate • To fix • Justified • No action planned • Not a defect • Other Polyspace Access removes results annotated with status Justified, No action planned, or Not a defect from the list of **Open Issues** in subsequent analyses.

Field	Allowed Value
Severity	Text to indicate how critical you consider the error in your code. This value populates the **Severity** column in the **Results List** pane as: • `Unset` • `High` • `Medium` • `Low`
Comment	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.

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" */
```

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 2-5 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] */
```

See Also

More About

- “Define Custom Annotation Format” on page 2-22
- “Short Names of Code Prover Run-Time Checks” on page 2-12
- “Short Names of Code Complexity Metrics” on page 2-14

Short Names of Code Prover Run-Time Checks

When annotating your code to justify checks or creating custom software quality objectives, you use short names of Code Prover run-time checks instead of the full names. The following table lists the short names for individual run-time checks.

Check	Acronym
Absolute address	ABS_ADDR
AUTOSAR runnable not implemented	AUTOSAR_NOIMPL
Correctness condition	COR
Division by zero	ZDV
Function not called	FNC
Function not reachable	FNR
Function returns a value	FRV
Illegally dereferenced pointer	IDP
Incorrect object oriented programming	OOP
Invalid C++ specific operations	CPP
Invalid floating point operation	INVALID_FLOAT_OP
Invalid result of AUTOSAR runnable implementation	AUTOSAR_IMPL
Invalid shift operations	SHF
Invalid use of AUTOSAR runtime environment function	AUTOSAR_USE
Invalid use of standard library routine	STD_LIB
Non-initialized local variable	NIVL
Non-initialized pointer	NIP
Non-initialized variable	NIV
Non-terminating call	NTC
Non-terminating loop	NTL

Check	Acronym
Null this-pointer calling method	NNT
Out of bounds array index	OBAI
Overflow	OVFL
Return value not initialized	IRV
Subnormal Float	SUBNORMAL
Uncaught exception	EXC
Unreachable Code	UNR
User assertion	ASRT

See Also

More About

- “Annotate Code and Hide Known or Acceptable 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
Lower Estimate of Local Variable Size	LOCAL_VARS_MIN
Minimum Stack Usage (Code Prover only)	MIN_STACK
Maximum Stack Usage (Code Prover only)	MAX_STACK
Number of Call Levels	LEVEL
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

- “Annotate Code and Hide Known or Acceptable Results” on page 2-5

Annotate Code for Known or Acceptable Results (Not Recommended)

Note Starting R2017b, Polyspace uses a simpler annotation format. See “Annotate Code and Hide Known or Acceptable Results” on page 2-5.

If Polyspace finds defects in your code that you cannot or will not fix, you can add annotations to your code. These annotations are code comments that indicate known or acceptable defects or coding rule violations. By using these annotations, you can:

- Avoid reviewing defects or coding rule violations from previous analyses.
- Preserve review comments and classifications.

Note Source code annotations do not apply to code comments. You cannot annotate these rules:

- MISRA C:2004 Rules 2.2 and 2.3
- MISRA C:2012 Rules 3.1 and 3.2
- MISRA-C++ Rule 2-7-1
- JSF++ Rules 127 and 133

Add Annotations Manually

This method shows you how to enter comments directly into your source files by using the Polyspace code annotation syntax. The syntax is not case-sensitive and applies to the first uncommented line of C/C++ code following the annotation.

1. Open your source file in an editor and locate the line or section of code that you want to annotate.
2. Add one of the following annotations:

   - For a single line of code, add the following text directly before the line of code that you want to annotate.

     ```
     /* polyspace<Type:Kind1[,Kind2] : [Severity] : [Status] > [Notes] */
     ```

- For a section of code, use the following syntax.

```
/* polyspace:begin<Type:Kind1[,Kind2] : [Severity] : [Status] > [Notes] */
... Code section ...
/* polyspace:end<Type:Kind1[,Kind2] : [Severity] : [Status] >  */
```

  If a macro expands to multiple lines, use the syntax for code sections even though the macro itself covers one line. The single-line syntax applies only to results that appear in the first line of the expanded macro.

3. Replace the words *Type*, *Kind1*, *[Kind2]*, *[Severity]*, *[Status]*, and *[Additional text]* with allowed values, indicated in the following table. The text with square brackets *[]* is optional and you can delete it. See “Syntax Examples” on page 2-20.

Word	Allowed Values
Type	The type of results: • `Defect` (Polyspace Bug Finder) • `RTE`, for run-time checks (Polyspace Code Prover) • `VARIABLE`, for global variables (Polyspace Code Prover) • `CODE-METRIC`, for code complexity metrics. • `MISRA-C`, for MISRA C:2004 • `MISRA-AC-AGC` • `MISRA-C3`, for MISRA C:2012 • `MISRA-CPP` • `JSF` • `Custom`, for custom coding rule violations.

Word	Allowed Values
Kind1, [Kind2],...	For defects, run-time checks and code metrics, use the short names of checkers. See: • "Short Names of Bug Finder Defect Checkers" (Polyspace Bug Finder Access) • "Short Names of Code Prover Run-Time Checks" on page 2-12 • "Short Names of Code Complexity Metrics" on page 2-14 For coding rule violations, specify the rule number or numbers. For global variables, the only allowed value is `ALL`. If you want the comment to apply to all possible defects or coding rules, specify `ALL`.
Severity	Text that indicates how critical you consider the defect. Enter one of the following: • `Unset` • `High` • `Medium` • `Low` This text populates the **Severity** column on the **Results List** pane.

Word	Allowed Values
Status	Text that indicates how you intend to correct the error in your code. Enter one of the following or any other text: • `Unreviewed` • `To investigate` • `To fix` • `Justified` • `No action planned` • `Not a defect` • `Other` This text populates the **Status** column on the **Results List** pane. The status is also used in Polyspace Access to determine whether a result is **Done**. To justify a result, use `Justified`, `No action planned` or `Not a defect`.
Notes	Additional comments, such as a keyword or an explanation for the status and severity.

Syntax Examples

- A single defect:

```
/* polyspace<Defect:HARD_CODED_BUFFER_SIZE:Medium:To investigate> Known issue */
int table[100];
```

- A single run-time check:

```
/* polyspace<RTE: ZDV : High : To Fix > Denominator cannot be zero */
y=1/x;
```

- A MISRA C:2012 rule violation:

```
/* polyspace<MISRA-C3: 13.1 : Low : Justified> Known issue */
int arr[2] = {x++,y};
```

- Unused global variable:

```
/* polyspace<VARIABLE: ALL : Low : Justified> Variable to use later*/
int var_unused;
```

- Multiple defects:

```
polyspace<Defect:USELESS_WRITE,DEAD_CODE:Low:No Action Planned> OK issue
```

- Multiple JSF rule violations:

  ```
  polyspace<JSF:9,13:Low:Justified> Known issue
  ```

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.

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*-*)*"
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="RTE" Result_Name="ZDV"/>
<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`. For more on analysis options, see the documentation for Polyspace Bug Finder or Polyspace Bug Finder Server .

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 Expresions. 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.

		`BEGIN`	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.
			
			`//myKeyword 50, 51 Block_on`
			`Code block 1;`
			`...`

Attribute	Use	Value	Example

		END	Stops the application of a rule identifier declared by a corresponding expression with attribute `Mode="BEGIN"`.
			`//myKeyword 50, 51 Block_on`
			`Code block 1;`
			`...`
			`More code;`
			`//myKeyword 50 Block_off`
			Only rule identifier 50 is turned off. Rule identifier 51 still applies.

		END_ALL	Stops all rule identifiers declared by an expression with attribute `Mode="BEGIN"`.
			`//myKeyword 50, 51 Block_on`
			`Code block 1;`
			`...`
			`More code;`
			`//myKeyword Block_off_all`
			Rule identifiers 50 and 51 are turned off.

`Regex`	Required	Regular expression search string	See Regular Expresions. `Regex="myKeyword\s+(\w+(\s*,\s*\w+)*)"` matches these expressions:
			`// myKeyword 50, 51`
			`/* myKeyword ALL_MISRA, 100 */`

Attribute	Use	Value	Example
`Rule_Identifier_Position`	Required, except when you set `Mode="END_ALL"`	Integer	The integer value of this attribute corresponds to the number of opening parentheses in the regular expression before the relevant search expression. `<Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s)(\w+(\s*,\s*\w+)*)" Increment_Position="1" Rule_Identifier_Position="2"/>` **Note** Enter the regex expression on a single line when you edit your XML file. The search expression for the rule identifier `\w+(\s*,\s*\w+)*` is after the second opening parenthesis of the regular expression.

Attribute	Use	Value	Example
`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. `<Expression Mode="GOTO_INCREMENT" Regex="myKeyword\s+(\+\d+\s)(\w+(\s*,\s*\w+)*)" Increment_Position="1" Rule_Identifier_Position="2"/>` **Note** Enter the regex expression on a single line when you edit your XML file. The search expression for the increment `\+\d+\s` is after the first opening parenthesis of the regular expression.
`Status_Position`	Optional	Integer	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.
`Severity_Position`	Optional	Integer	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.

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	See the mapping section of `annotations_description.xml`
`Family`	Required	Corresponds to Polyspace results family. For a list of allowed values, see allowed values on page 2-5.	See the mapping section of `annotations_description.xml`
`Result_Name`	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 2-5.	See the mapping section of `annotations_description.xml`

See Also

“Annotation Description Full XML Template” on page 2-31

More About

- “Annotate Code and Hide Known or Acceptable Results” on page 2-5

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 2-22.

Element	Attribute	Use	Value
`Annotations`	`Group`	Required	User defined string. For example, "Custom Annotations"
`Expressions`	`Search_For_Keywords`	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"
	`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, " "

Element	Attribute	Use	Value
	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
			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.

Element	Attribute	Use	Value
	`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.
	`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.

Element	Attribute	Use	Value
	`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.

Element	Attribute	Use	Value
	`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.
`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 2-5.
	`Result_Name`	Required	Corresponds to Polyspace result names. For a list of allowed values, see allowed values on page 2-5.

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

- “Annotate Code and Hide Known or Acceptable Results” on page 2-5

Justify Coding Rule Violations Using Code Prover Checks

Coding rules are good practices that you observe for safe and secure code. Using the Polyspace coding rule checkers, you find instances in your code that violate a coding rule standard such as MISRA. If you run Code Prover, you also see results of checks that find run-time errors or prove their absence. In some cases, the two kinds of results can be used together for efficient review. For instance, you can use a green Code Prover check as rationale for not fixing a coding rule violation (justification).

If you run MISRA checking in Code Prover, some of the checkers use Code Prover static analysis under the hood to find MISRA violations. The MISRA checker in Code Prover is more rigorous compared to Bug Finder because Code Prover keeps precise track of the data and control flow in your code. For instance:

- `MISRA C:2012 Rule 9.1`: The rule states that the value of an object with automatic storage duration shall not be read before it has been set. Code Prover uses the results of a `Non-initialized local variable` check to determine the rule violations.
- MISRA C:2004 Rule 13.7: The rule states that the Boolean operations whose results are invariant shall not be permitted. Code Prover uses the results of an `Unreachable code` check to identify conditions that are always true or false.

In some other cases, the MISRA checkers do not suppress rule violations even though corresponding green checks indicate that the violations have no consequence. You have the choice to do one of these:

- Strictly conform to the standard and fix the rule violations.
- Manually justify the rule violations using the green checks as rationale.

  Set a status such as `No action planned` to the result and enter the green check as rationale in the result comments. You can later filter justified results using that status.

The following sections show examples of situations where you can justify MISRA violations using green Code Prover checks.

Rules About Data Type Conversions

In some cases, implicit data type conversions are okay if the conversion does not cause an overflow.

In the following example, the line `temp = var1 - var2;` violates `MISRA C:2012 Rule 10.3`. The rule states that the value of an expression shall not be assigned to an object of

a different essential type category. Here, the difference between two `int` variables is assigned to a `char` variable. You can justify this particular rule violation by using the results of a Code Prover `Overflow` check.

```
int func (int var1, int var2) {
    char temp;
    temp = var1 - var2;
    if (temp > 0)
        return -1;
    else
        return 1;
}

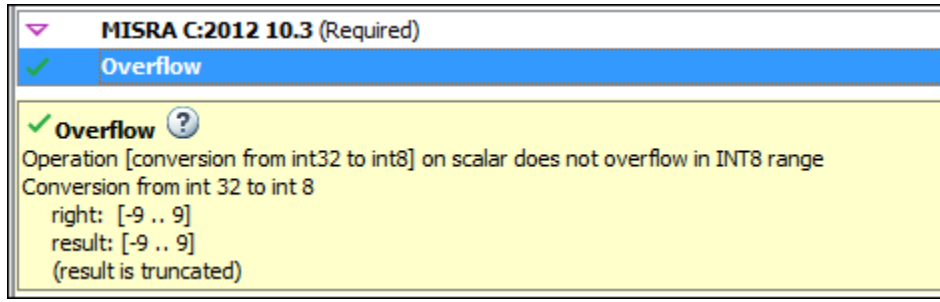
double read_meter1(void);
double read_meter2(void);

int main(char arg, char* argv[]) {
    int meter1 = (read_meter1()) * 10;
    int meter2 = (read_meter2()) * 999;
    int tol = 10;
    if((meter1 - meter2)> -tol && (meter1 - meter2) < tol)
        func(meter1, meter2);
    return 0;
}
```

Consider the rationale behind this rule. The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision. For a conversion from `int` to `char`, a loss of sign or precision cannot happen. The only issue is a potential loss of value if the difference between the two `int` variables overflows.

Run Code Prover on this code. On the **Source Code** pane, click the = in `temp = var1 - var2;`. You see the expected violation of MISRA C:2012 Rule 10.3, but also a green **Overflow** check.

The green check indicates that the conversion from `int` to `char` does not overflow.



You can use the green overflow check as rationale to justify the rule violation.

Rules About Pointer Arithmetic

Pointer arithmetic on nonarray pointers are okay if the pointers stay within the allowed buffer.

In the following example, the operation `ptr++` violates MISRA C:2004 Rule 17.4. The rule states that array indexing shall be the only allowed form of pointer arithmetic. Here, a pointer that is not an array is incremented.

```
#define NUM_RECORDS 3
#define NUM_CHARACTERS 6

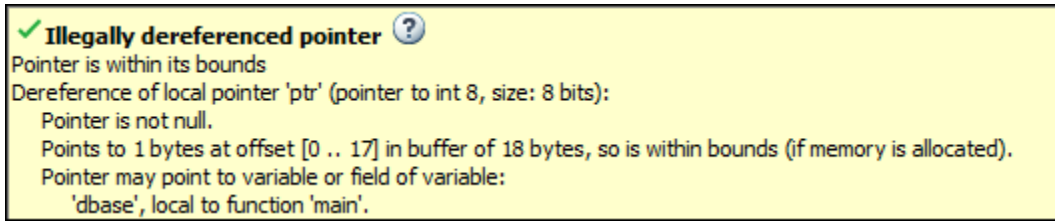
void readchar(char);

int main(int argc, char* argv[]) {
    char dbase[NUM_RECORDS][NUM_CHARACTERS] = { "r5cvx", "a2x5c", "g4x3c"};
    char *ptr = &dbase[0][0];
    for (int index = 0; index < NUM_RECORDS * NUM_CHARACTERS; index++) {
        readchar(*ptr);
        ptr++;
    }
    return 0;
}
```

Consider the rationale behind this rule. After an increment, a pointer can go outside the bounds of an allowed buffer (such as an array) or even point to an arbitrary location. Pointer arithmetic is fine as long as the pointer points within an allowed buffer. You can justify this particular rule violation by using the results of a Code Prover `Illegally dereferenced pointer` check.

Run Code Prover on this code. On the **Source Code** pane, click the ++ in `ptr++`. You see the expected violation of MISRA C:2004 Rule 17.4.

Click the * on the operation `readchar(*ptr)`. You see a green **Illegally dereferenced pointer** check. The green check indicates that the pointer points within allowed bounds when dereferenced.



You can use the green check to justify the rule violation.

See Also

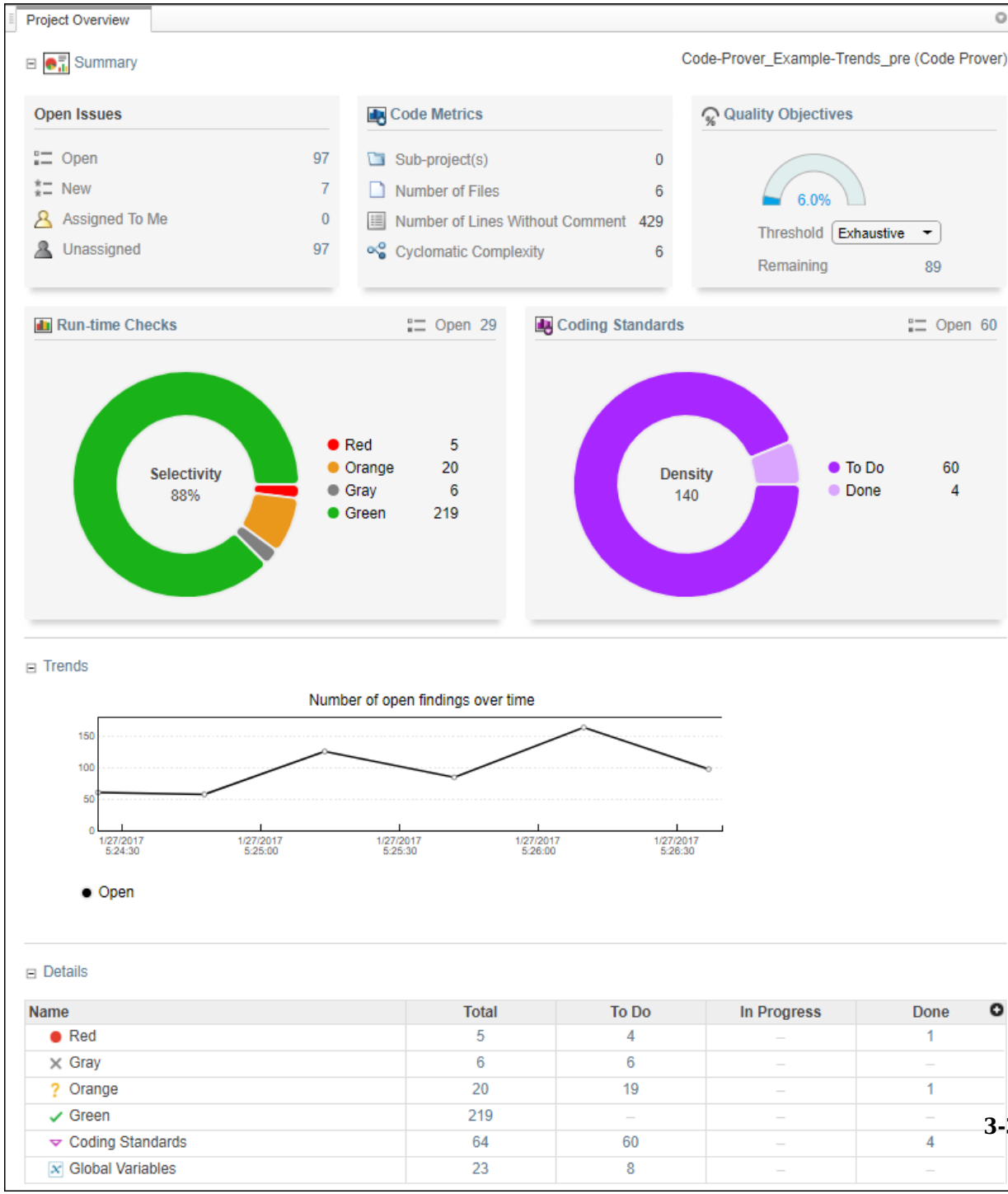
Related Examples

- “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2

Manage Results

Filter and Sort Results

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 defects (Bug Finder), run-time checks (Code Prover), coding rule violations or other results. To organize your review, you can narrow down the list or group results by file or result type.


Project Overview
Summary
Code-Prover_Example-Trends_pre (Code Prover)
Open Issues
Open 97
New 7
Assigned To Me 0
Unassigned 97
Code Metrics
Sub-project(s) 0
Number of Files 6
Number of Lines Without Comment 429
Cyclomatic Complexity 6
Quality Objectives
6.0%
Threshold Exhaustive
Remaining 89
Run-time Checks
Open 29
Selectivity
88%
Red 5
Orange 20
Gray 6
Green 219
Coding Standards
Open 60
Density
140
To Do 60
Done 4
Trends
Number of open findings over time
150
100
50
0
1/27/2017 5:24:30
1/27/2017 5:25:00
1/27/2017 5:25:30
1/27/2017 5:26:00
1/27/2017 5:26:30
Open
Details
Name
Total
To Do
In Progress
Done
Red 5 4 – 1
Gray 6 6 – –
Orange 20 19 – 1
Green 219 – – –
Coding Standards 64 60 – 4
Global Variables 23 8 – –

Some of the ways you can use filtering are:

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

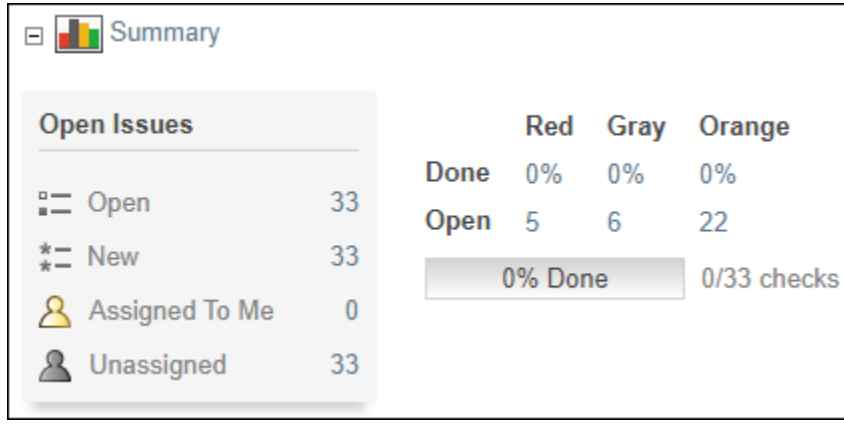
  For instance, for a Bug Finder analysis, you can display only high-impact defects. See "Classification of Defects by Impact" (Polyspace Bug Finder Access).
- You can display only new results found since the last analysis.
- You can display only the results that you have not justified. Results that are not justified are considered **Open**. They are results with status `Unreviewed`, `To Investigate`, `To Fix`, or `Other`.

  For information on justification, see"Address Polyspace Results Through Bug Fixes or Comments" on page 2-2 .

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

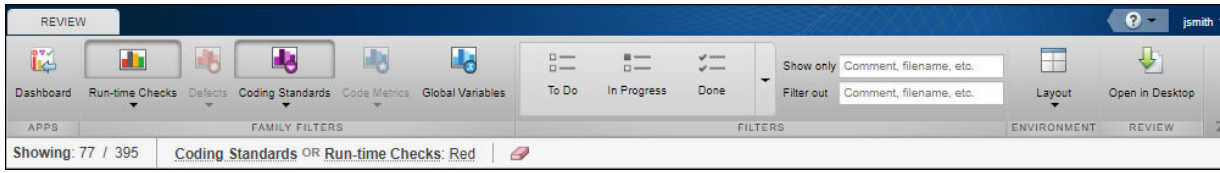


In the **DASHBOARD** view, you can open dashboards for different families of results, then click a number to open a list filtered to the corresponding set of results. For instance:

- 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.

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 using the buttons in the toolstrip. The filter bar underneath the toolstrip shows how many finding 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 out by the content of a column in the “Results List” on page 1-30, right-click the content of the column for that result.

If you do not want to filter by the exact contents of a column, you can use the **Show only** and **Filter out** text filters instead. For instance, you want to filter out subfolders of a

specific folder. Instead of filtering out the full folder path using the right-click menu, then sorting the **Folder** and **Filter out** filters.

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

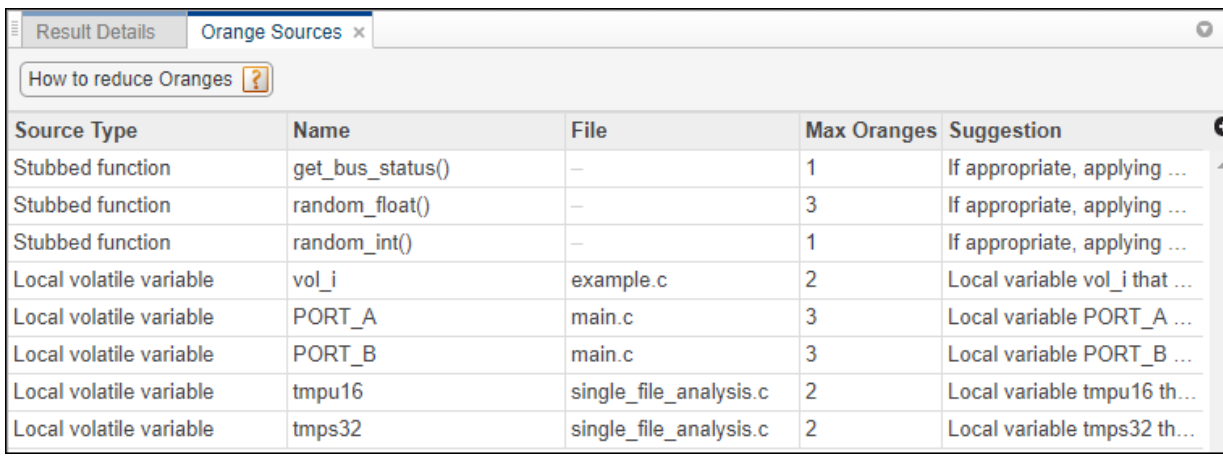
Filter Using Orange Sources

An orange source can cause multiple orange checks in Code Prover. You can display all orange checks from the same source and review them together.

For instance, in this code, the unknown value `input` can cause an overflow and a division by zero. The variable `input` is an orange source that causes two orange checks.

```
void func (int input) {
int val1;
double val2;
val1 = input++;
val2 = 1.0/input;
}
```

To begin, in the **REVIEW** view, select **Layout > Show/Hide View > Orange Sources**. You see the list of orange sources. Select an orange source to see all orange checks coming from this source.



Source Type	Name	File	Max Oranges	Suggestion
Stubbed function	get_bus_status()	–	1	If appropriate, applying ...
Stubbed function	random_float()	–	3	If appropriate, applying ...
Stubbed function	random_int()	–	1	If appropriate, applying ...
Local volatile variable	vol_i	example.c	2	Local variable vol_i that ...
Local volatile variable	PORT_A	main.c	3	Local variable PORT_A ...
Local volatile variable	PORT_B	main.c	3	Local variable PORT_B ...
Local volatile variable	tmpu16	single_file_analysis.c	2	Local variable tmpu16 th...
Local volatile variable	tmps32	single_file_analysis.c	2	Local variable tmps32 th...

See Also

More About

- “Prioritize Check Review” on page 3-8

Prioritize Check Review

This example shows how to prioritize your check review. Try the following approach. You can also develop your own procedure for organizing your orange check review.

Tip For easier review, run Polyspace Bug Finder on your source code first. Once you address the defects that Polyspace Bug Finder finds, run Polyspace Code Prover on your code.

Before beginning your check review, you can check the following:

- See the **Run Log**,by going to **Layout > Show/Hide View** in the **REVIEW** view. Use `CTRL-F` to search the log for warning and error messages, or the string `failed compilation`. If there are warnings or errors, or files failed to compile, identify why Polyspace could not analyze all of your source files.

  To check for some common *Reasons for Unchecked Code*, see the documentation for Polyspace Code Prover.
- See if you have used the right configuration. The configuration options are listed in the **Run Log** under the strings `Options used with Verifier:` and `User:`.

  Sometimes, especially if you are switching between multiple configurations, you can accidentally use the wrong configuration for the verification.

1. From the **Project Overview** dashboard, click the number next to **Open** on the **Run-time Checks** card.

   This action opens the **Results List** pane with only unreviewed red, gray and orange checks. You can also filter for these results from the toolstrip in the **REVIEW** view by clicking **Run-time Checks** and **To Do**.
2. Select and review the first check.

   For more information, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

   Continue going through the list until you have reviewed all of the checks.
3. Before reviewing orange checks, review red and gray checks.
4. Prioritize your orange check review by:

- For easier review, begin your orange check review from files with *fewer* orange checks.

  To sort files by number of orange checks, in the **Details** section of the **Run-time Checks** dashboard, click **View by File**, then click the head of the **Orange** column to sort it. Click an entry from this column to open the corresponding list of orange checks.

- Check type: Review orange checks in the following order. Checks are more difficult to review as you go down this order.

Review Order	Checks
First	• `Out of bounds array index` • `Non-initialized local variable` • `Division by zero` • `Invalid shift operations`
Second	• `Overflow` • `Illegally dereferenced pointer`
Third	Remaining checks

- Orange check sources: Review all orange checks caused by a single variable or function. Orange checks often arise from variables whose values cannot be determined from the code or functions that are not defined.

  To review the sources of orange checks, select an orange check from the **Results List** pane then click  in the **Results Details** pane. You can also open the **Orange Sources** pane by going to **Layout > Show/Hide View**. For more information, see “Filter Using Orange Sources” on page 3-6.

- Result details: Review all results that originate from the same cause. Sometimes, the **Detail** column on the **Results List** pane shows additional information about a result. For instance, if multiple issues trigger the same coding rule violation, this column shows the issue. Click the column header so that results that originate from the same type of issue are grouped together. Review the results in one go.

5 To see what percentage of checks you have justified, go to the **DASHBOARD** view and see the **Summary** section of the **Run-time Checks** dashboard.

See Also

Related Examples

- “Filter and Sort Results” on page 3-2

Reviewing Checks

Review and Fix Absolute Address Usage Checks

Follow one or more of these steps until you determine a fix for the **Absolute address usage** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Absolute address usage`.

Tip This check is green by default. To reduce the number of orange checks, if you trust that all absolute addresses in your code are valid, you can retain this default behavior.

For best use of this check, leave this check green by default during initial stages of development. During integration stage, use the option `-no-assumption-on-absolute-addresses` and detect all uses of absolute memory addresses. Browse through them and make sure that the addresses are valid. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

1. Select the check on the **Results List** pane.

   The **Source** pane displays the code operation containing the absolute address.
2. If you determine that the address is valid, add a comment and justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

Review and Fix Correctness Condition Checks

Follow one or more of these steps until you determine a fix for the **Correctness condition** check. There are multiple ways to fix a red or orange check. For a description of the check and code examples, see `Correctness condition`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

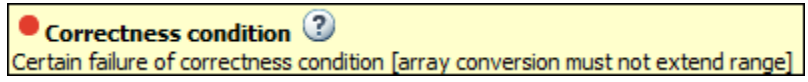
For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

On the **Results List** pane, select the check. View the cause of check on the **Result Details** pane. The following list shows some of the possible causes:

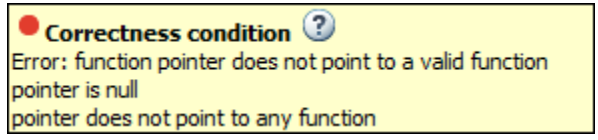
- An array is converted to another array of larger size.

  In the following example, a red check occurs because an array is converted to another array of larger size.

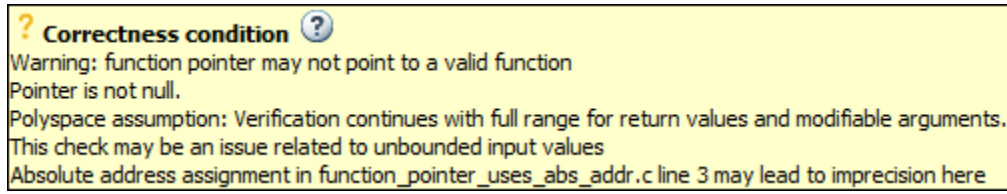
- When dereferenced, a function pointer has value `NULL`.

  In the following example, a red check occurs because, when dereferenced, a function pointer has value `NULL`.

- When dereferenced, a function pointer does not point to a function.

  In the following example, an orange check occurs because Polyspace cannot determine if a function pointer points to a function when dereferenced. This situation can occur if, for instance, you assign an absolute address to the function pointer.

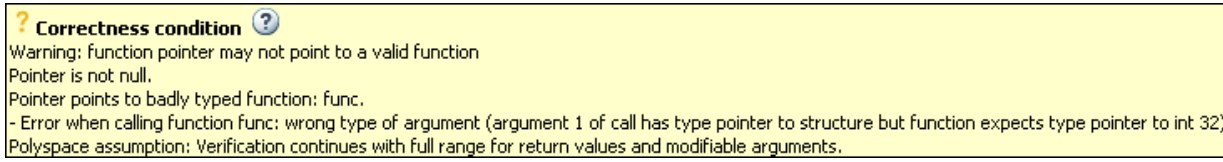


- A function pointer points to a function, but the argument types of the pointer and the function do not match. For example:

  ```
  typedef int (*typeFuncPtr) (complex*);
  int func(int* x);
  .
  .
  typeFuncPtr funcPtr = &func;
  ```

  In the following example, a red check occurs because:

  - The function pointer points to a function `func`.
  - `func` expects an argument of type `int`, but the corresponding argument of the function pointer is a structure.

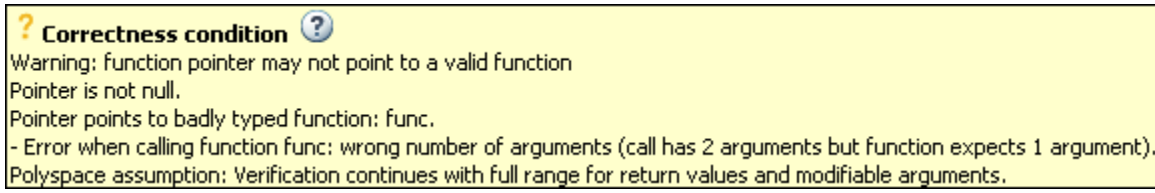


- A function pointer points to a function, but the argument numbers of the pointer and the function do not match. For example:

  ```
  typedef int (*typeFuncPtr) (int, int);
  int func(int);
  .
  .
  typeFuncPtr funcPtr = &func;.
  ```

  In the following example, a red check occurs because:

  - The function pointer points to a function `func`.
  - `func` expects one argument but the function pointer has two arguments.

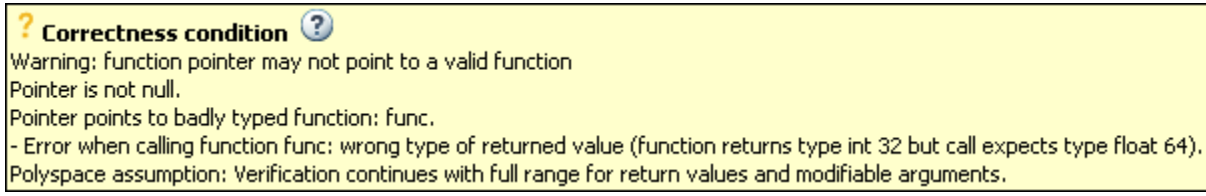


- A function pointer points to a function, but the return types of the pointer and the function do not match. For example:

  ```
  typedef double (*typeFuncPtr) (int);
  int func(int);
  .
  .
  typeFuncPtr funcPtr = &func;
  ```

  In the following example, a red check occurs because:

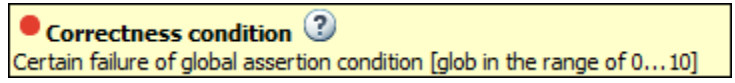
  - The function pointer points to a function `func`.
  - `func` returns an `int` value, but the return type of the function pointer is `double`.



- The value of a variable falls outside the range that you specify through the **Global Assert** mode. See the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

  In the following example, a red check occurs because:

  - You specify a range 0...10 for the variable `glob`.
  - The value of the variable falls outside this range.



Step 2: Determine Root Cause of Check

Based on the check information on the **Result Details** pane, perform further steps to determine the root cause. You can perform the following steps in the Polyspace user interface only.

Check Information	How to Determine Root Cause
An array is converted to another array of larger size.	1 To determine the array sizes, see the definition of each array variable. Right-click the variable and select **Go To Definition**. 2 If you dynamically allocate memory to an array, it is possible that their sizes are not available during definition. Browse through all instances of the array variable to find where you allocate memory to the array. a Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. b On the **Search** pane, select the previous instances.

Check Information	How to Determine Root Cause
Issues when dereferencing a function pointer: • The function pointer has value `NULL` when dereferenced. • The function pointer does not point to a function when dereferenced. • The function pointer points to a function, but the argument types of the pointer and the function do not match. • The function pointer points to a function, but the argument numbers of the pointer and the function do not match. • The function pointer points to a function, but the return types of the pointer and the function do not match.	**1** Find the location where you assign the function pointer to a function. **a** Right-click the function pointer. Select **Search For All References**. All instances of the function pointer appear on the **Search** pane with the current instance highlighted. **b** On the **Search** pane, select the previous instances. **2** Determine the argument and return types of the function pointer type and the function. Identify if there is a mismatch between the two. For instance, in the following example, determine the argument and return types of `typeFuncPtr` and `func`. `typeFuncPtr funcPtr = func;` **a** Right-click the function pointer type and select **Go To Definition**. **b** Right-click the function and select **Go To Definition**. If the definition does not exist, this option shows the function stub definition instead. In this case, find the function declaration. **3** Sometimes, you assign a function pointer to a function with matching signature, but the assignment is unreachable. Check if this is the case.

Check Information	How to Determine Root Cause
The value of a variable falls outside the range that you specify through the **Global Assert** mode.	Browse through all previous instances of the global variable. Identify a suitable point to constrain the variable. 1 Right-click the variable. Select **Show In Variable Access View**. 2 On the **Variable Access** pane, select each instance of the variable.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

Review and Fix Division by Zero Checks

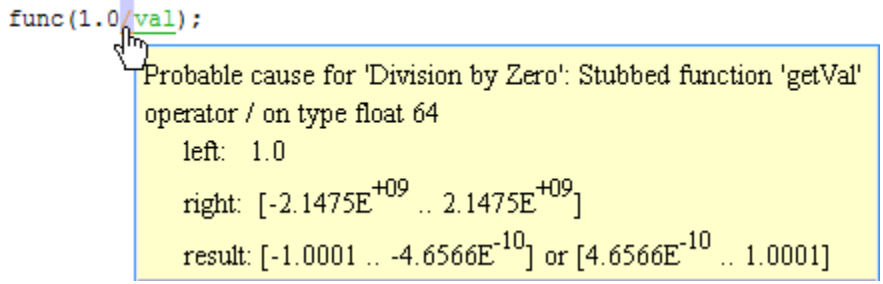
Follow one or more of these steps until you determine a fix for the **Division by zero** check. There are multiple ways to fix a red or orange check. For a description of the check and code examples, see `Division by zero`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

Place your cursor on the `/` or `%` operation that causes the **Division by zero** error.



Obtain the following information from the tooltip:

- The values of the right operand (denominator).

  In the preceding example, the right operand, `val`, has a range that contains zero.

  Possible fix: To avoid the division by zero, perform the division only if `val` is not zero.

Integer	Floating-point

`if(val != 0)`
`    func(1.0/val);`
`else`
`    /* Error handling */`
	`#define eps 0.0000001`
	`.`
	`.`
	`if(val < -eps || val > eps)`
	`    func(1.0/val);`
	`else`
	`    /* Error handling */`

- The probable root cause for division by zero, if indicated in the tooltip.

  In the preceding example, the software identifies a stubbed function, `getVal`, as probable cause.

  Possible fix: To avoid the division by zero, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `1..10`. To specify constraints, use the analysis option `Constraint setup (-data-range-specifications)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Step 2: Determine Root Cause of Check

Before a `/` or `%` operation, test if the denominator is zero. Provide appropriate error handling if the denominator is zero.

Only if you do not expect a zero denominator, determine root cause of check. Trace the data flow starting from the denominator variable. Identify a point where you can specify a constraint to prevent the zero value.

In the following example, trace the data flow starting from `arg2`:

```
void foo() {
    double time = readTime();
    double dist = readDist();
    .
    .
  bar(dist,time);
}

void bar(double arg1, double arg2) {
    double vel;
    vel=arg1/arg2;
}
```

You might find that:

1. `bar` is called with full-range of values.

   Possible fix: Call `bar` only if its second argument `time` is greater than zero.

2. `time` obtains a full-range of values from `readTime`.

   Possible fix: Constrain the return value of `readTime`, either in the body of `readTime` or through the Polyspace Constraint Specification interface, if you do not have the definition of `readTime`. For more information, see “Stubbed Functions” on page 6-9.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find the previous write operation on the operand variable.

     Example: The value of `arg2` is written from the value of `time` in `bar`.

  2. At the previous write operation, identify a new variable to trace back.

     Place your cursor on the variables involved in the write operation to see their values. The values help you decide which variable to trace.

     Example: At `bar(dist,time)`, you find that `time` has a full-range of values. Therefore, you trace `time`.

  3. Find the previous write operation on the new variable. Continue tracing back in this way until you identify a point to specify your constraint.

     Example: The previous write operation on `time` is `time=readTime()`. You can choose to specify your constraint on the return value of `readTime`.

Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with  and read operations with  .
Function return value `ret=func();`	1 Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. 2 In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 4-104.

Step 3: Look for Common Causes of Check

Look for common causes of the **Division by zero** check.

- For a variable that you expect to be non-zero, see if you test the variable in your code to exclude the zero value.

  Otherwise, Polyspace cannot determine that the variable has non-zero values. You can also specify constraints outside your code. See the documentation for Polyspace Code Prover or Polyspace Code Prover Server.
- If you test the variable to exclude its zero value, see if the test occurs in a reduced scope compared to the scope of the division.

  For example, a statement `assert(var !=0)` occurs in an `if` or `while` block, but a division by `var` occurs outside the block. If the code does not enter the `if` or `while` block, the `assert` does not execute. Therefore, outside the `if` or `while` block, Polyspace assumes that `var` can still be zero.

  Possible fix:

  - Investigate why the test occurs in a reduced scope. In the above example, see if you can place the statement `assert(var !=0)` outside the `if` or `for` block.
  - If you expect the `if` or `while` block to always execute, investigate when it does not execute.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

For instance, you are using a volatile variable in your code. Then:

1. Polyspace assumes that the variable is full-range at every step in the code. The range includes zero.
2. A division by the variable can cause **Division by zero** error.
3. If you know that the variable takes a non-zero value, add a comment and justification describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Disabling This Check

You can effectively disable this check. If your compiler supports infinities and NaNs from floating-point operations, you can enable a verification mode that incorporates infinities and NaNs. See `Consider non finite floats (-allow-non-finite-floats)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Review and Fix Function Not Called Checks

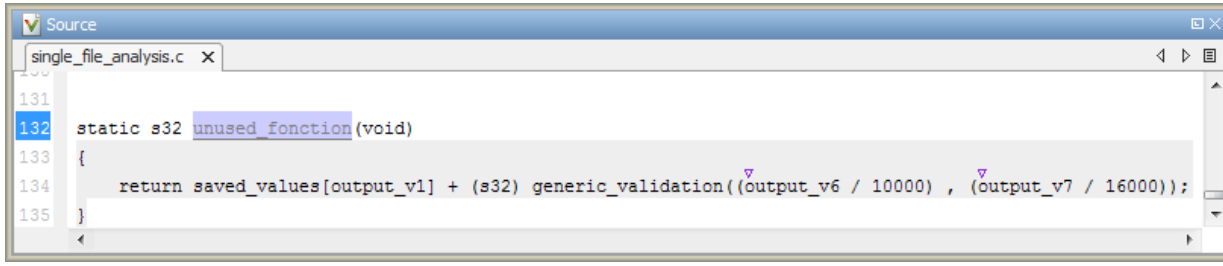
Follow one or more of these steps until you determine a fix for the **Function not called** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Function not called`.

If you determine that the check represents defensive code or a function that is part of a library, add a comment and justification in your result or code explaining why you did not change your code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

Note This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see `Detect uncalled functions (-uncalled-function-checks)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Step 1: Interpret Check Information

On the **Results List** pane, select the check. On the **Source** pane, the body of the function is highlighted in gray.



Step 2: Determine Root Cause of Check

1 Search for the function name and see if you can find a call to the function in your code.

On the **Search** pane, enter the function name. From the drop-down list beside the search field, select **Source**.

Possible fix: If you do not find a call to the function, determine why the function definition exists in your code.

2. If you find a call to the function, see if it occurs in the body of another uncalled function.

   Possible fix: Investigate why the latter function is not called.

3. See if you call the function indirectly, for example, through function pointers.

   If the indirection is too deep, Polyspace sometimes cannot determine that a certain function is called.

   Possible fix: If Polyspace cannot determine that you are calling a function indirectly, you must verify the function separately. You do not need to write a new `main` function for this other verification. Polyspace can generate a `main` function if you do not provide one in your source. You can change the `main` generation options if needed. For more information on the options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Step 3: Look for Common Causes of Check

Look for the following common causes of the **Function not called** check.

- Determine if you intended to call the function but used another function instead.
- Determine if you intended to replace some code with a function call. You wrote the function definition, but forgot to replace the original code with the function call.

  If this situation occurs, you are likely to have duplicate code.
- See if you intend to call the function from yet unwritten code. If so, retain the function definition.
- For code intended for multitasking, see if you have specified all your entry point functions.

  To see the options used for the result, select the link **View configuration for results** on the **Dashboard** pane.

  For more information, see `Tasks (-entry-points)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

- For code intended for multitasking, see if your `main` function contains an infinite loop. Polyspace Code Prover requires that your `main` function must complete execution before the other entry points begin.

Review and Fix Function Not Reachable Checks

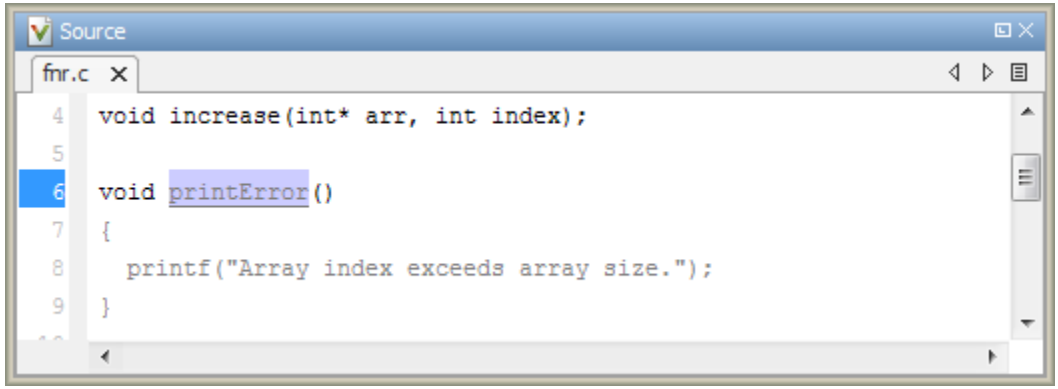
Follow one or more of these steps until you determine a fix for the **Function not reachable** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Function not reachable`.

If you determine that the check represents defensive code, add a comment and justification in your result or code explaining why you did not change your code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

Note This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see `Detect uncalled functions (-uncalled-function-checks)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. On the **Source** pane, you can see the function definition in gray.



Step 2: Determine Root Cause of Check

Determine where the function is called and review why all the function call sites are unreachable. You can perform the following steps in the Polyspace user interface only.

1. Select the check on the **Results List** pane.
2. On the **Result Details** pane, click the  button.

   On the **Call Hierarchy** pane, you see the callers of the function denoted by  .
3. On the **Call Hierarchy** pane, select each caller.

   This action takes you to the function call on the **Source** pane.
4. See if the caller itself is called from unreachable code. If the caller definition is entirely in gray on the **Source** pane, it is called from unreachable code. Follow the same investigation process, starting from step 1, for the caller.
5. Otherwise, investigate why the section of code from which you call the function is unreachable.

   The code can be unreachable because it follows a red check or because it contains the gray **Unreachable code** check.

   - If a red check occurs, fix your code to remove the check.
   - If a gray **Unreachable code** check occurs, review the check and determine if you must fix your code. See “Review and Fix Unreachable Code Checks” on page 4-93.

Note If you do not see a caller name on the **Call Hierarchy** pane, determine if you are calling the function indirectly, for example through a function pointer. Determine if a mismatch occurs between the function pointer declaration and the function call through the pointer.

Polyspace places a red or orange **Correctness condition** check on the indirect call if a mismatch occurs. To detect a mismatch in indirect function calls, look for the **Correctness condition** check on the **Results List** pane. For more information, see `Correctness condition`.

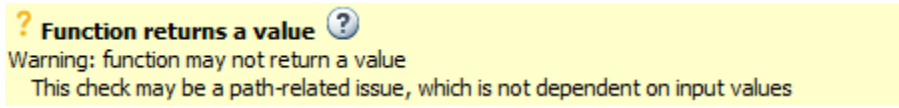
Review and Fix Function Not Returning Value Checks

Follow one or more of these steps until you determine a fix for the **Function not returning value** check. For a description of the check and code examples, see `Function not returning value`.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. The **Result Details** pane displays further information about the check.



You can see:

- The immediate cause of the check.

  In this example, the software has identified that a function with a non-`void` return type might not have a `return` statement.
- The probable root cause of the check, if indicated.

  In this example, the software has identified that the check is possibly path-related. More than one call to the function exists, and the check is green on at least one call.

Step 2: Determine Root Cause of Check

Determine why a `return` statement does not exist on certain execution paths.

1. Browse the function body for `return` statements.
2. If you find a `return` statement:

   a. See if the `return` statement occurs in a block inside the function.

      For instance, the `return` statement occurs in an `if` block. An execution path that does not enter the `if` block bypasses the `return` statement.

b See if you can identify the execution paths that bypass the `return` statement.

   For instance, an `if` block that contains the `return` statement is bypassed for certain function inputs.

c If the function is called multiple times in your code, you can identify which function call led to bypassing of the `return` statement. Use the option `Sensitivity context (-context-sensitivity)` to determine the check color for each function call. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Possible fix: If the return type of the function is incorrect, change it. Otherwise, add a `return` statement on all execution paths. For instance, if only a fraction of branches of an `if-else if-else` condition have a `return` statement, add a `return` statement in the remaining branches. Alternatively, add a `return` statement outside the `if-else if-else` condition.

Review and Fix Illegally Dereferenced Pointer Checks

Follow one or more of these steps until you determine a fix for the **Illegally dereferenced pointer** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Illegally dereferenced pointer`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

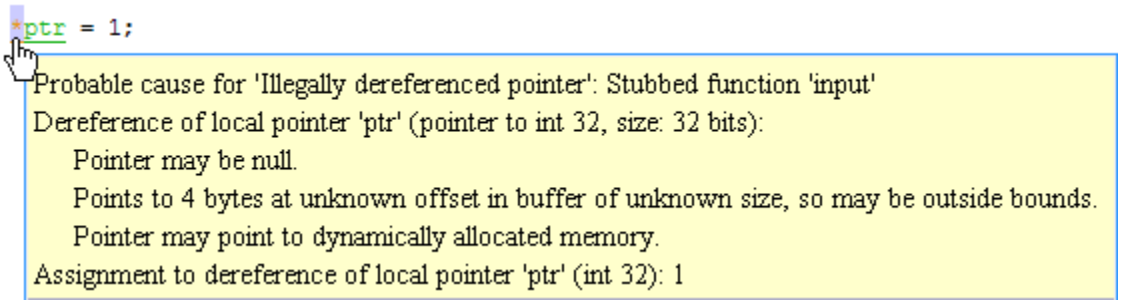
Step 1: Interpret Check Information

Place your cursor on the dereference operator.

Obtain the following information from the tooltip:

- Whether the pointer can be `NULL`.

  In the following example, `ptr` can be `NULL` when dereferenced.

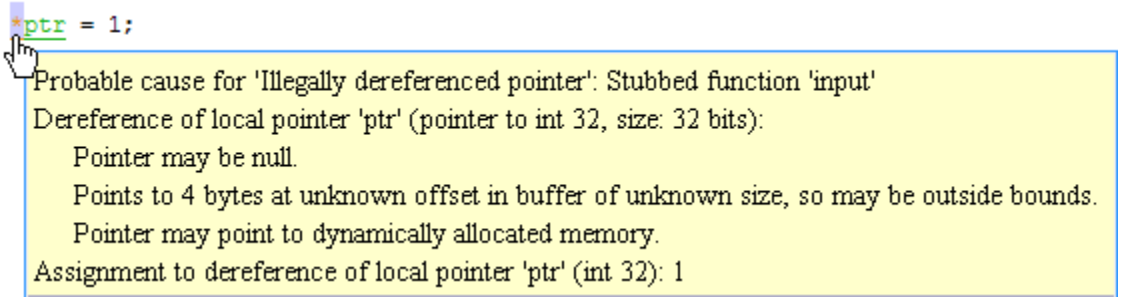


  Possible fix: Dereference `ptr` only if it is not `NULL`.

```
if(ptr !=NULL)
    *ptr = 1;
else
    /* Alternate action */
```

- Whether the pointer points to dynamically allocated memory.

  In the following example, `ptr` can point to dynamically allocated memory. It is possible that the dynamic memory allocation operator returns `NULL`.



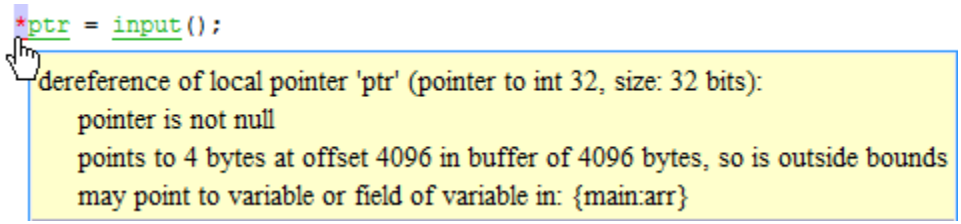
  Possible fix: Check the return value of the memory allocation operator for `NULL`.

```
ptr = (char*) malloc(i);
if(ptr==NULL)
  /* Error handling*/
else {
  .
    .
    *ptr=0;
    .
    .
}
```

- Whether pointer points outside allowed bounds. A pointer points outside bounds when the sum of pointer size and offset is greater than buffer size.

  In the following example, the offset size (4096 bytes) together with pointer size (4 bytes) is greater than the buffer size (4096 bytes). If the pointer points to an array:

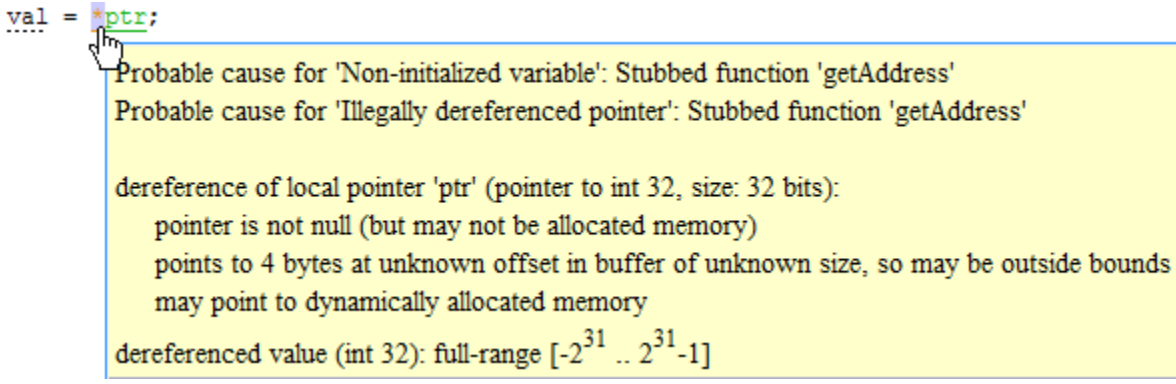
  - The buffer size is the array size.
  - The offset is the difference between the beginning of the array and the current location of the pointer.



Possible fix: Investigate why the pointer points outside the allowed buffer.

- Whether pointer can point outside allowed bounds because buffer size is unknown.

  In the following example, the buffer size is unknown.

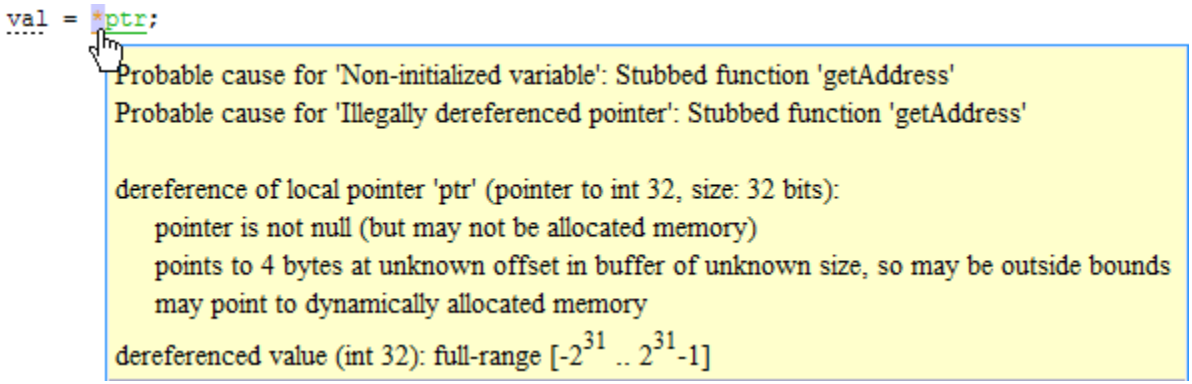


Possible fix: Investigate whether the pointer is assigned:

  - The return value of an undefined function.
  - The return value of a dynamic memory allocation function. Sometimes, Polyspace cannot determine the buffer size from the dynamic memory allocation.
  - Another pointer of a different type, for instance, `void*`.

- The probable root cause for illegal pointer dereference, if indicated in the tooltip.

  In the following example, the software identifies a stubbed function, `getAddress`, as probable cause.



Possible fix: To avoid the illegally dereferenced pointer, constrain the return value of `getAddress`. For instance, specify that `getAddress` returns a pointer to a 10-element array. For more information, see “Stubbed Functions” on page 6-9.

Step 2: Determine Root Cause of Check

Select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction and trace back to the root cause.
- If the **Result Details** pane shows the line number of probable cause for the check, in the Polyspace user interface, right-click the **Source** pane. Select **Go To Line**.
- Otherwise, based on the nature of the error, use one of the following methods to find the root cause. You can perform the following steps in the Polyspace user interface only.

Error	How to Find Root Cause

Pointer can be `NULL`.	Find an execution path where the pointer is assigned the value NULL or not assigned a definite address.
	1 Right-click the pointer and select **Search For All References**.
	2 Find each previous instance where the pointer is assigned an address.
	3 For each instance, on the **Source** pane, place your cursor on the pointer. The tooltip indicates whether the pointer can be `NULL`.
	Possible fix: If the pointer can be `NULL`, place a check for `NULL` immediately after the assignment.
	`if(ptr==NULL)`
	`  /* Error handling*/`
	`else {`
	`  .`
	`  .`
	`  }`
	4 If the pointer is not `NULL`, see if the assignment occurs only in a branch of a conditional statement. Investigate when that branch does not execute.
	Possible fix: Assign a valid address to the pointer in all branches of the conditional statement.

Pointer can point to dynamically allocated memory.	Identify where the allocation occurs. **1** Right-click the pointer and select **Search For All References**. **2** Find the previous instance where the pointer receives a value from a dynamic memory allocation function such as `malloc`. *Possible fix*: After the allocation, test the pointer for `NULL`.

Error	How to Find Root Cause
Pointer can point outside bounds allowed by the buffer.	1 Find the allowed buffer. a On the **Search** tab, enter the name of the variable that the pointer points to. You already have this name from the tooltip on the check. b Search for the variable definition. Typically, this is the first search result. If the variable is an array, note the array size. If the variable is a structure, search for the structure type name on the **Search** tab and find the structure definition. Note the size of the structure field that the pointer points to. 2 Find out why the pointer points outside the allowed buffer. a Right-click the pointer and select **Search For All References**. b Identify any increment or decrement of the pointer. See if you intended to make the increment or decrement. *Possible fix*: Remove unintended pointer arithmetic. To avoid pointer arithmetic that takes a pointer outside allowed buffer, use a reference pointer to store its initial value. After every arithmetic operation on your pointer, compare it with the reference pointer to see if the difference is outside the allowed buffer.

Step 3: Look for Common Causes of Check

Look for common causes of the **Illegally dereferenced pointer** check.

- If you use pointers for moving through an array, see if you can use an array index instead.

To avoid use of pointer arithmetic in your code, look for violations of MISRA C: 2004 rule 17.4 or MISRA C: 2012 rule 18.4.

- See if you use pointers for moving through the fields of a structure.

  Polyspace does not allow the pointer to one field of a structure to point to another field. To allow this behavior, use the option `Enable pointer arithmetic across fields (-allow-ptr-arith-on-struct)`.
- See if you are dereferencing a pointer that points to a structure but does not have sufficient memory for all its fields. Such a pointer usually results from type-casting a pointer to a smaller structure.

  Polyspace does not allow such dereference. To allow this behavior, use the option `Allow incomplete or partial allocation of structures (-size-in-bytes)`.
- If an orange check occurs in a function body, see if you are passing arrays of different sizes in different calls to the function.

  See if one particular call causes the orange check.
- See if you are performing a cast between two pointers of incompatible sizes.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

For instance, the pointer receives an address from an undefined function. Then:

1. Polyspace assumes that the function can return `NULL`.

   Therefore, the pointer dereference is orange.
2. Polyspace also assumes an allowed buffer size based on the type of the pointer.

   If you increment the pointer, you exceed the allowed buffer. The pointer dereference that follows the increment is orange.

3 If you know that the function returns a non-`NULL` value or if you know the true allowed buffer, add a comment and justification in your code describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Review and Fix Incorrect Object Oriented Programming Checks

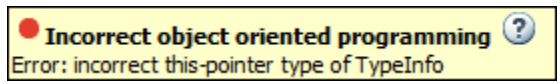
In this section...

Follow one or more of these steps until you determine a fix for the **Incorrect object oriented programming** check. For a description of the check and code examples, see `Incorrect object oriented programming`.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

On the **Results List** pane, select the check. The **Result Details** pane displays further information about the check.



You can see:

- The immediate cause of the check. For instance:
  - You dereference a function pointer that has the value `NULL` or points to an invalid member function.

    The member function is invalid if its argument or return type does not match the pointer argument or return type.
  - You call a pure `virtual` member function of a class from the class constructor or destructor.
  - You call a member function using an incorrect `this` pointer.

    To see why the `this` pointer can be incorrect, see `Incorrect object oriented programming`.

- The probable root cause of the check, if indicated.

Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, use navigation shortcuts in the user interface to navigate to the root cause.

Based on the specific error, use one of the following methods to find the root cause.

Error	How to Find Root Cause
You dereference a function pointer that has the value `NULL`.	Right-click the function pointer and select **Search For All References**. Find the instance where you assign `NULL` to the function pointer.
You dereference a function pointer that points to an invalid member function.	Compare the argument and return types of the function pointer and the member function that it points to. **1** Right-click the function pointer on the **Source** pane and select **Search For All References**. Find the instances where you: • Define the function pointer. • Assign the address of a member function to the function pointer. **2** Find the member function definition. Right-click the member function name on the **Source** pane and select **Go To Definition**.

Error	How to Find Root Cause
You call a pure `virtual` member function from a constructor or destructor.	Find the member function declaration and determine whether you intended to declare it as `virtual` or pure `virtual`. Alternatively, determine if you can replace the call to the pure `virtual` function with another operation, for instance, a call to a different member function. **1** Right-click the function name on the **Source** pane and select **Search for *`function_name`* in All Source Files**. **2** Find the function declaration from the search results. A pure `virtual` function has a declaration such as: `virtual void func() = 0;`

You call a member function using an incorrect `this` pointer.	Determine why the `this` pointer is incorrect.
	For instance, if a red **Incorrect object oriented programming** check appears on a function call `ptr->func()` and the message indicates that the `this` pointer is incorrect, trace the data flow for `ptr`.
	• Right-click the function pointer on the **Source** pane and select **Search For All References**.
	• Browse through all write operations on the pointer. Look for the following issues:
	  • Cast between pointers of unrelated types.
	  • Pointer arithmetic that takes a pointer outside its allowed buffer, for instance, the bounds of an array.
	If a red **Incorrect object oriented programming** check appears on a function call `obj.func()`, trace the data flow for `obj`. See if `obj` is not initialized previously.

Review and Fix Invalid C++ Specific Operations Checks

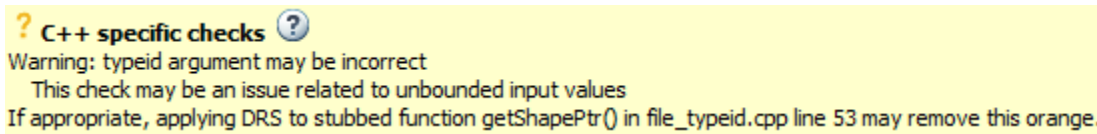
Follow one or more of these steps until you determine a fix for the **Invalid C++ specific operations** check. There are multiple ways to fix a red or orange check. For a description of the check and code examples, see `Invalid C++ specific operations`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

On the **Results List** pane, select the check. The **Result Details** pane displays further information about the check.



You can see:

- The immediate cause of the check. For instance:
  - The size of an array is not strictly positive.

    For instance, you create an array using the statement `arr = new char [num]`. `num` is possibly zero or negative.

    Possible fix: Use `num` as an array size only if it is positive.
  - The `typeid` operator dereferences a possibly `NULL` pointer.

    Possible fix: Before using the `typeid` operator on a pointer, test the pointer for `NULL`.
  - The `dynamic_cast` operator performs an invalid cast.

Possible fix: The invalid cast results in a `NULL` return value for pointers and the `std::bad_cast` exception for references. Try to avoid the invalid cast. Otherwise, if the invalid cast is on pointers, make sure that you test the return value of `dynamic_cast` for `NULL` before dereference. If the invalid cast is on references, make sure that you catch the `std::bad_cast` exception in a `try-catch` statement.

- The probable root cause of the check, if indicated.

Step 2: Determine Root Cause of Check

If you cannot determine the root cause based on the check information, use navigation shortcuts in the user interface to navigate to the root cause.

Based on the nature of the error, use one of the following methods to find the root cause.

Error	How to Find Root Cause
An array size is nonpositive.	**1** Trace the data flow for the size variable. Follow the same root cause investigation steps as for a **Division by Zero** check. See “Review and Fix Division by Zero Checks” on page 4-10. **2** Identify a point where you can constrain the array size variable to positive values.
The `typeid` operator dereferences a possibly `NULL` pointer.	**1** Trace the data flow for the pointer variable. Follow the same root cause investigation steps as for an **Illegally dereferenced pointer** check. See “Review and Fix Illegally Dereferenced Pointer Checks” on page 4-23. **2** Identify a point where you can test the pointer for `NULL`.
The `dynamic_cast` operator performs an invalid cast.	Navigate to the definitions of the classes involved. Determine the inheritance relationship between the classes. **1** On the **Source** pane in the Polyspace user interface, right-click the class name. **2** Select **Go To Definition**.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

For instance, you obtain the array size variable from a stubbed function `getSize`. Then:

1. Polyspace assumes that the return value of `getSize` is full-range. The range includes nonpositive values.
2. Using the variable as array size in dynamic memory allocation causes orange **Invalid C++ specific operations**.
3. If you know that the variable takes a positive value, add a comment and justification explaining why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Review and Fix Invalid Shift Operations Checks

Follow one or more of these steps until you determine a fix for the **Invalid shift operations** check. There are multiple ways to fix the check. For a description of the check and code examples, see `Invalid shift operations`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

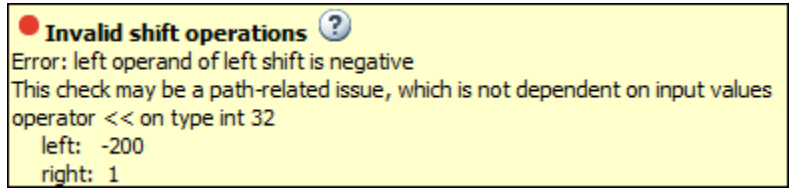
Step 1: Interpret Check Information

Select the red or orange **Invalid shift operations** check. Obtain the following information from the **Result Details** pane:

- The reason for the check being red or orange. Possible reasons:
  - The shift amount can be outside allowed bounds.

    The software also states the allowed range for the shift amount.
  - Left operand of left shift can be negative.

  In the example below, a red error occurs because the shift amount is outside allowed bounds. The allowed range for the shift amount is 0 to 31.

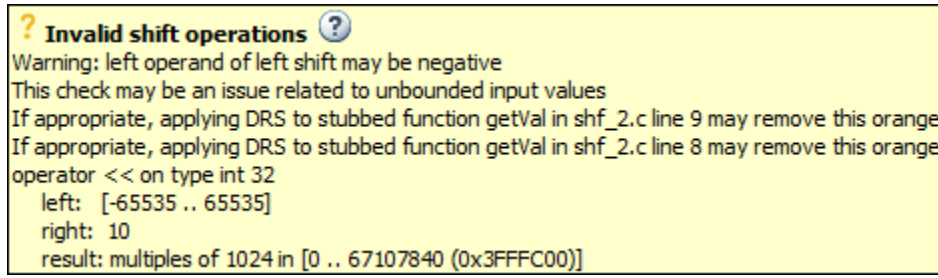


  Possible fix: To avoid the red or orange check, perform the shift operation only if the shift amount is within bounds.

```
if(shiftAmount < (sizeof(int) * 8))
  /* Perform the shift */
```

```
else
  /* Error handling */
```

- Probable root cause for the check, if the software provides this information.



  In the preceding example, the software identifies a stubbed function, `getVal` as probable cause.

  Possible fix: To avoid the orange check, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `0..10`. For more information, see “Stubbed Functions” on page 6-9.

Step 2: Determine Root Cause of Check

- If the shift amount is outside bounds, trace the data flow for the shift variable. Identify a suitable point where you can constrain the shift variable.

  In the following example, trace the data flow for `shiftAmount`.

```
void func(int val) {
    int shiftAmount = getShiftAmount();
    int res = val >> shiftAmount;
}
```

  You might find that `getShiftAmount` returns full-range of values.

  Possible fix:

  - Perform the shift operation only if `shiftAmount` is between 0 and `(sizeof(int))*8 - 1`.
  - Constrain the return value of `getShiftAmount`, in the body of `getShiftAmount` or through the Polyspace Constraint Specification interface, if you do not have the

definition of `getShiftAmount`. For more information, see “Stubbed Functions” on page 6-9.

- If the left operand of a left shift operation can be negative, trace the data flow for the left operand variable. Identify a suitable point where you can constrain the left operand variable.

  In the following example, trace the data flow for `shiftAmount`.

  ```
  void func(int shiftAmount) {
      int val = getVal();
      int res = val << shiftAmount;
  }
  ```

  You might find that `getVal` returns full-range of values.

  Possible fix:

  - Perform the shift operation only if `val` is positive.
  - Constrain the return value of `getVal`, in the body of `getVal` or through the Polyspace Constraint Specification interface, if you do not have the definition of `getVal`. For more information, see “Stubbed Functions” on page 6-9.
  - If you want Polyspace to allow the operation, use the analysis option **Allow negative operand for left shifts**. See `Allow negative operand for left shifts (-allow-negative-operand-in-shift)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find the previous write operation on the variable you want to trace.
  2. At the previous write operation, identify a new variable to trace back.

     Place your cursor on the variables involved in the write operation to see their values. The values help you decide which variable to trace.

3. Find the previous write operation on the new variable. Continue tracing back in this way until you identify a point to specify your constraint.

Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1. Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2. On the **Search** pane, select the previous instances. • Browse the source code. 1. Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2. Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1. Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2. On this pane, select the previous instances of the variable. Write operations on the variable are indicated with  and read operations with  .

Variable	How to Find Previous Instances of Variable
Function return value `ret=func();`	1 Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. 2 In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 4-104.

Step 3: Look for Common Causes of Check

Look for common causes of the **Invalid Shift Operations** check.

- See if you have specified the right target processor type. The target processor type determines the number of bits allowed for a certain variable type.

  To determine the number of bits allowed:

  1. Navigate to the variable definition. Note the variable type.

     Right-click the variable and select **Go To Definition**, if the option exists.
  2. See the number of bits allowed for the type.

     In the configuration used for your results, select the **Target & Compiler** node. Click the **Edit** button next to the **Target processor type** list.
- For left shifts with a negative operand, see if you intended to perform a right shift instead.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

For instance, you obtain a variable from an undefined function and perform a left shift on it. Then:

1. Polyspace assumes that the function can return a negative value.
2. The left shift operation can occur on a negative value and therefore there is an orange check on the operation.
3. If you know that the function returns a positive value, add a comment and justification describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Review and Fix Invalid Use of Standard Library Routine Checks

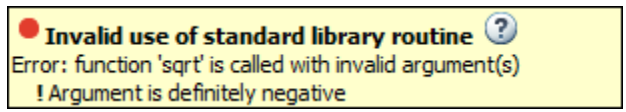
Follow one or more of these steps until you determine a fix for the **Invalid use of standard library routine** check. For a description of the check and code examples, see `Invalid use of standard library routine`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. View further information about the check on the **Result Details** pane. The check is red or orange because of invalid function arguments.



The cause of a red or orange check depends on the standard library function that you use. The following table shows the possible causes for some of the commonly used functions.

Function	Cause of Red or Orange Check
`islower`, `isdigit`, and other character-handling functions in `ctype.h`	The value of the argument can be outside the range allowed for an `unsigned char` variable. Note that you can use the macro `EOF` as argument.

Function	Cause of Red or Orange Check	
Functions in `math.h`	The software checks for multiple kinds of errors in sequence. The software performs each check only for those execution paths where the previous check passes. Some examples are given below. For more information and a list of functions, see “Invalid Use of Standard Library Floating Point Routines” on page 4-46.	
	`sqrt`	The value of the argument can be negative.
	`pow`	The first argument can be negative while the second argument is a non-integer.
	`exp`, `exp2`, or the hyperbolic functions	The argument can be so large that the result exceeds the value allowed for a `double`.
	`log`	The argument can be zero or negative.
	`asin` or `acos`	The argument can be outside the range [-1,1].
	`tan`	The argument can have the value `HALF_PI`.
	`acosh`	The argument can be less than 1.
	`atanh`	The argument can be greater than 1 or less than -1.
`fprintf`, `fscanf`, and other file handling functions	The file pointer argument can be non-readable. For example, it can be `NULL`.	
Functions that take string arguments	The string argument can be an invalid string. For example, it does not end with a terminating `'\0'`.	

Function	Cause of Red or Orange Check
`memmove` or `memcpy`	The third argument of this function specifies the number of bytes to copy from the second to the first argument. This number can exceed the memory allocated to the first or second argument.

Step 2: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

For instance, you obtain a value from an undefined function and perform the `sqrt` operation on it. Then:

1. Polyspace assumes that the function can return a negative value.
2. Therefore, the software produces an orange **Invalid Use of Standard Library Routine** check on the `sqrt` function call.
3. If you know that the function returns a positive value, to avoid the orange, you can specify a constraint on the return value of your function. See “Stubbed Functions” on page 6-9. Alternately, add a comment and justification describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Invalid Use of Standard Library Floating Point Routines

Polyspace Code Prover performs the **Invalid Use of Standard Library Routine** check on standard library routines to determine if their arguments are valid. The check works differently for memory routines, floating-point routines or string routines because their arguments can be invalid in different ways. This topic describes how the check works for standard library floating-point routines.

For more information on the check, see `Invalid use of standard library routine`.

What the Check Looks For

The **Invalid Use of Standard Library Routine** check sequentially looks for the following issues in use of floating-point routines.

- Domain error: A domain error occurs if the arguments of the function are invalid. The definition of invalid argument varies based on whether you allow non-finite floats or not. If you allow non-finite floats but:

  - Specify that you must be warned about NaN results, a domain error occurs if the function returns NaN and the arguments themselves are not NaN.
  - Specify that NaN results must be forbidden, a domain error occurs if the function returns NaN or the arguments themselves are NaN.

  For details, see `NaNs (-check-nan)`.

  The check works in almost the same way as the check `Invalid operation on floats`. The **Invalid Use of Standard Library Routine** check works on standard library functions while the **Invalid Operation on Floats** check works on numerical operations involving floating-point variables.
- Overflow error: An overflow error occurs if the result of the function overflows. The definition of overflow varies based on whether you allow non-finite floats and based on the rounding modes you specify. If you allow non-finite floats but specify that you must be warned about infinite results, an overflow error occurs if the function returns infinity and the arguments themselves are not infinity. For details, see `Infinities (-check-infinite)`.

The check works in the same way as the check `Overflow`. The **Invalid Use of Standard Library Routine** check works on standard library functions while the **Overflow** check works on numerical operations involving floating-point variables.

- Invalid pointer argument: For functions such as `frexp` that take pointer arguments, the verification checks if it is valid to dereference the pointer. For instance, the pointer is not NULL or does not point outside allowed bounds.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

The check looks for these errors in sequence.

- If the check finds a definite domain error, it does not look for the overflow error.
- If the check finds a possible domain error, it looks for the overflow error only for the execution paths where the domain error does not occur.

The check for each error itself can consist of multiple conditions, which are also checked in sequence. Each check is performed only for those execution paths where the previous check passes.

Single-Argument Functions Checked

The **Invalid Use of Standard Library Routine** check covers the following routines, their single-precision versions with suffix `f` (if they have one) and their long double versions with suffix `l`. The check works in exactly the same way for C and C++ code.

- `acos`
- `acosh`
- `asin`
- `asinh`
- `atan`
- `atanh`
- `ceil`
- `cos`
- `cosh`
- `exp`

- exp2
- expm1
- fabs
- floor
- log
- log10
- log1p
- logb
- round
- sin
- sinh
- sqrt
- tan
- tanh
- trunc
- cbrt

Functions with Multiple Arguments

The **Invalid Use of Standard Library Routine** check covers the following routines, their single-precision versions with suffix `f` (if they have one) and their long double versions with suffix `l`. The check works in exactly the same way for C and C++ code.

- atan2
- fdim
- fma
- fmax
- fmin
- fmod
- frexp
- hypot
- ilogb

- `ldexp`
- `modf`
- `nextafter`
- `nexttoward`
- `pow`
- `remainder`

See Also

Review and Fix Non-initialized Local Variable Checks

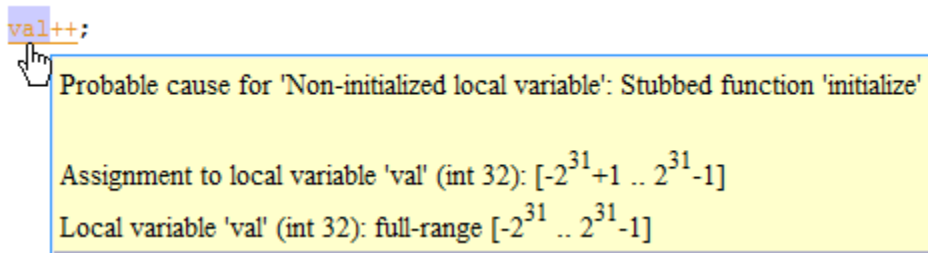
Follow one or more of these steps until you determine a fix for the **Non-initialized local variable** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Non-initialized local variable`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

Place your cursor on the variable on which the **Non-initialized local variable** error appears.



Obtain the probable root cause for the variable being non-initialized, if indicated in the tooltip.

In the preceding example, the software identifies a stubbed function, `initialize`, as probable cause.

Possible fix: To avoid the check, you can specify that `initialize` writes to its arguments. For more information, see “Stubbed Functions” on page 6-9.

Step 2: Determine Root Cause of Check

You can perform the following steps in the Polyspace user interface only.

1. Search for the variable definition. See if you initialize the variable when you define it.

   Right-click the variable and select **Go To Definition**, if the option exists.

2. If you do not want to initialize the variable during definition, browse through all instances of the variable. Determine if you initialize the variable in any of those instances.

   Do one of the following:

   - On the **Source** pane, double-click the variable.

     Previous instances of the variable are highlighted. Scroll up to find them.
   - On the **Source** pane, right-click the variable. Select **Search For All References**.

     Select the previous instances on the **Search** pane.

   Possible fix: If you do not initialize the variable, identify an instance where you can initialize it.

3. If you find an instance where you initialize the variable, determine if you perform the initialization in the scope where the **Non-initialized local variable** error appears.

   For instance, you initialize the variable only in some branches of an `if ... elseif ... else` statement. If you use the variable outside the statement, the variable can be non-initialized.

   Possible fix:

   - Perform the initialization in the same scope where you use it.

     In the preceding example, perform the initialization outside the `if ... elseif ... else` statement.
   - Perform the initialization in a block with smaller scope but make sure that the block always executes.

     In the preceding example, perform the initialization in all branches of the `if ... elseif ... else` statement. Make sure that one branch of the statement always executes.

Step 3: Look for Common Causes of Check

Look for common causes of the **Non-initialized local variable** check.

- See if you pass the variable to another function by reference or pointers before using it. Determine if you initialize the variable in the function body.

  To navigate to the function body, right-click the function and select **Go To Definition**, if the option exists.
- Determine if you initialize the variable in code that is not reachable.

  For instance, you initialize the variable in code that follows a `break` or `return` statement.

  Possible fix: Investigate the unreachable code. For more information, see “Review and Fix Unreachable Code Checks” on page 4-93.
- Determine if you initialize the variable in code that can be bypassed during execution.

  For instance, you initialize the variable in a loop inside a function. However, for certain function arguments, the loop does not execute.

  Possible fix:

  - Initialize the variable during declaration.
  - Investigate when the code can be bypassed. Determine if you can avoid bypassing of the code.
- If the variable is an array, determine if you initialize all elements of the array.
- If the variable is a structured variable, determine if you initialize all fields of the structure.

  If you do not initialize a certain field of the structure, see if the field is unused.

  Possible fix: Initialize a field of the structure if you use the field in your code.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

For instance, you pass a variable to a function by pointer or reference. You intend to initialize the variable in the function body, but you do not provide the function body during verification. Then:

- Polyspace assumes that the function might not initialize the variable.
- If you use the variable following the function call, Polyspace considers that the variable can be non-initialized. It produces an orange **Non-initialized local variable** check on the variable.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Disabling This Check

You can disable this check. If you disable this check, Polyspace assumes that at declaration, variables have full-range of values allowed by their type. For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Review and Fix Non-initialized Pointer Checks

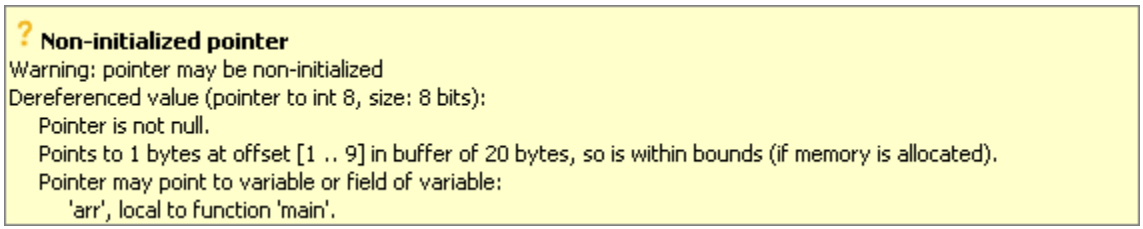
Follow one or more of these steps until you determine a fix for the **Non-initialized pointer** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Non-initialized pointer`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. On the **Result Details** pane, obtain further information about the check.



Step 2: Determine Root Cause of Check

Right-click the pointer variable and select **Go To Definition**. Initialize the variable when you define it. If you do not want to initialize during definition, identify a suitable point to initialize the variable before you read it.

For orange checks, determine why the pointer is non-initialized on certain execution paths.

1. Find previous instances where write operations are performed on the pointer.
2. For each write operation, determine if the operation occurs:

- Before the read operation containing the orange **Non-initialized pointer** check.

  Possible fix: If the write operation occurs after the read operation, see if you intended to perform the operations in reverse order.
- In an unreachable code block.

  Possible fix: Investigate why the code block is unreachable. See “Review and Fix Unreachable Code Checks” on page 4-93.
- In a code block that is not reached on certain execution paths. For example, the operation occurs in an `if` block in a function. The `if` block is not entered for certain function inputs.

  Possible fix: Perform a write operation on all the execution paths. In the preceding example, perform the write operation in all branches of the `if ... elseif ... else` statement.

Depending on the nature of the variable, use the appropriate method to find previous operations on the variable. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Operations on Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 On the **Source** pane, double-click the variable. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.

Variable	How to Find Previous Operations on Variable
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. The current instance of the variable is shown on the **Variable Access** pane. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with ◀. Read operations are indicated with ▶.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

Disabling This Check

You can disable the check in two ways:

- You can disable the check only for non-local pointers. Polyspace considers global pointer variables to be initialized to `NULL` according to ANSI® C standards. For more information, see `Ignore default initialization of global variables (-no-def-init-glob)`.
- You can disable the check completely along with other initialization checks. If you disable this check, Polyspace assumes that at declaration, pointers can be `NULL` or point to memory blocks at an unknown offset. For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Review and Fix Non-initialized Variable Checks

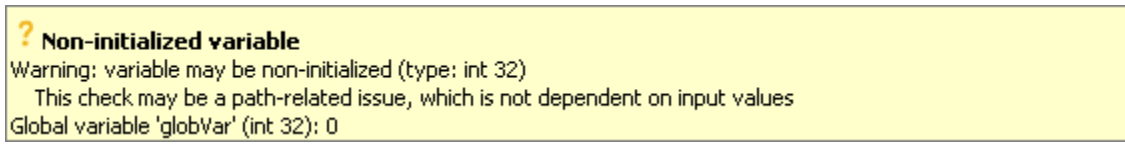
Follow one or more of these steps until you determine a fix for the **Non-initialized variable** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Non-initialized variable`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

On the **Results List** pane, select the check. On the **Result Details** pane, obtain further information about the check.



Obtain the following information:

- Probable cause of check, if described on the **Result Details** pane.

  In the preceding example, there is an orange **Non-initialized variable** check on the global variable `globVar`.

  The software detects that the check is potentially a path-related issue. Therefore, the global variable can be non-initialized only on certain execution paths. For example, you initialized the global variable in an `if` block, but did not initialize it in the corresponding `else` block.

  Possible fix: Determine along which paths the global variables can be non-initialized.
- Value of global variable, if initialized.

  In the preceding example, when initialized, the global variable `globVar` has value 0.

Step 2: Determine Root Cause of Check

You can perform the following steps in the Polyspace user interface only.

Right-click the variable and select **Go To Definition**. Initialize the variable when you define it. If you do not want to initialize during definition, identify a suitable point to initialize the variable before you read it.

If the check is orange, determine why the variable is non-initialized on certain execution paths.

1. Right-click the variable. Select **Show In Variable Access View**.
2. On the **Variable Access** pane, select each write operation on the variable.

   Write operations are indicated with ◂ and read operations with ▸.
3. Determine if the write operation occurs:

   - Before the read operation containing the orange **Non-initialized variable** check.

     Possible fix: If the write operation occurs after the read operation, see if you intended to perform the operations in reverse order.
   - In an unreachable code block.

     Possible fix: Investigate why the code block is unreachable. See “Review and Fix Unreachable Code Checks” on page 4-93.
   - In a code block that is not reached on certain execution paths. For example, the operation occurs in an `if` block in a function. The `if` block is not entered for certain function inputs.

     Possible fix: Perform a write operation on all the execution paths. In the preceding example, perform the write operation in all branches of the `if ... elseif ... else` statement.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

Disabling This Check

You can disable this check in two ways:

- You can specify that global variables must be considered as initialized. Polyspace considers global variables to be initialized according to ANSI C standards. The default values are:

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

  For more information, see `Ignore default initialization of global variables (-no-def-init-glob)`.
- You can disable the check along with other initialization checks. If you disable this check, Polyspace assumes that at declaration, variables have the full range of values allowed by their type. For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Review and Fix Non-Terminating Call Checks

Follow one or more of these steps until you determine a fix for the **Non-terminating call** check. There are multiple ways to fix the check. For a description of the check and code examples, see `Non-terminating call`.

For the general workflow on reviewing checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

A red **Non-terminating call** check on a function call indicates one of the following:

- An operation in the function body failed for that particular call. Because there are other calls to the same function that do not cause a failure, the operation failure typically appears as an orange check in the function body.
- The function does not return to its calling context for other reasons. For example, a loop in the function body does not terminate.

Step 1: Determine Root Cause of Check

Determine the root cause of the check in the function body. You can perform the following steps in the Polyspace user interface only.

1. Navigate to the function definition.

   Right-click the function call containing the red check. Select **Go To Definition**, if the option exists.

2. In the function body, determine if there is a loop where the termination condition is never satisfied.

   Possible fix: Change your code or the function arguments so that the termination condition is satisfied.

3. Otherwise, in the function body, identify which orange check caused the red **Non-terminating call** check on the function call.

   If you cannot find the orange check by inspection, rerun verification using the analysis option **Sensitivity context**. Provide the function name as option argument. The software marks the orange check causing the red **Non-terminating call** check as a dark orange check.

For more information, see `Sensitivity context (-context-sensitivity)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

For a tutorial on using the option, see “Identify Function Call with Run-Time Error” on page 4-63.

Possible fix: Investigate the cause of the orange check. Change your code or the function arguments to avoid the orange check.

Step 2: Look for Common Causes of Check

To trace a **Non-terminating call** check on a function call to an orange check in the function body, try the following:

- If the function call contains arguments, in the function body, search for all instances of the function parameters. See if you can find an orange check related to the parameters. Because other calls to the same function do not cause an operation failure, it is likely that the failure is related to the function parameter values in the red call.

  In the following example, in the body of `func`, search for all instances of `arg1` and `arg2`. Right-click the variable name and select **Search For All References**.

  ```
  void func(int arg1, double arg2) {
    .
    .
  }

  void main() {
    int valInt1,valInt2;
    double valDouble1, valDouble2;
    .
    .
    func(valInt1, valDouble1);
    func(valInt2, valDouble2);
  }
  ```

  Because `valInt1` and `valDouble1` do not cause an operation failure in `func`, the failure might be due to `valInt2` or `valDouble2`. Because `valInt2` and `valDouble2` are copied to `arg1` and `arg2`, the orange check must occur in an operation related to `arg1` or `arg2`.

- If the function call does not contain arguments, identify what is different between various calls to the function. See if you can relate the source of this difference to an orange check in the function body.

  For instance, if the function reads a global variable, different calls to the function can operate on different values of the global variable. Determine if the function body contains an orange check related to the global variable.

Identify Function Call with Run-Time Error

This tutorial shows how to identify the function call that causes a run-time error in the function body.

If a function contains two different colors on the same operation for two different calls, the software combines the call contexts and displays an orange check on the operation. For example, when some function calls cause a red or orange check on an operation in the function body and other calls cause a green check, in your verification results, the operation is orange.

You have to distinguish orange checks that arise from combination of call contexts because an orange check can arise from other causes. Using the option `Sensitivity context (-context-sensitivity)`, make this distinction by storing individual call contexts for a function. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

In this tutorial, a function is called twice. You identify which function call causes a run-time error in the function body.

1. Run analysis on this code and open the results.

   ```
   void func(int arg) {
       int loc_var = 0;
       loc_var = 1/arg;
   }

   void main(void) {
       int num = 1;
       func(num + 10);
       func(num - 1);
   }
   ```

   A red **Non-terminating call** check appears on the second call to `func`. In the body of `func`, there is an orange **Division by zero** check on the `/` operation.

2. Specify that you want to store individual call context information for the function `func`.

   a. In your project configuration, select the **Precision** node.

   b. Select `custom` for **Sensitivity context**.

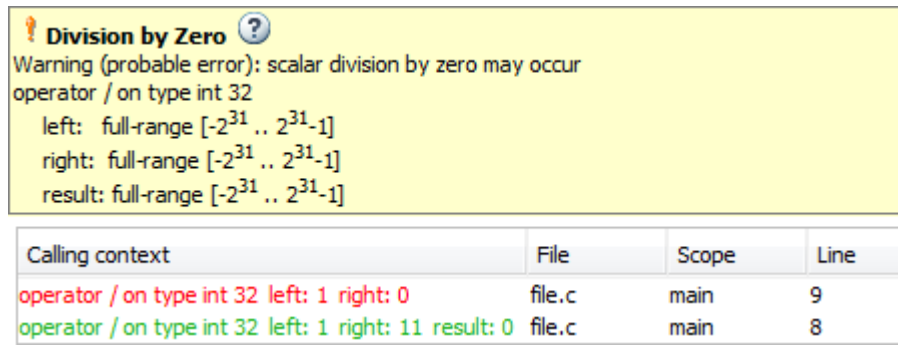
   c. Click  to add a new field. Enter `func`.

3 Run verification and open the results.

An orange **Division by zero** check still appears in the body of `func`. However, this orange check is marked on the **Results List** pane as a dark orange check and is denoted by a mark. Instead of a red **Non-terminating call** check, a dashed, red line appears on the second call to `func`.

4 Select the orange check.

The **Result Details** pane shows the call contexts for the check. You can see that one call produces a red check on the `/` operation and the other call produces a green check. You can click each call to navigate to it in your source code.



See Also

`Non-terminating call`

Related Examples

- “Review and Fix Non-Terminating Call Checks” on page 4-60

More About

- “Orange Checks in Code Prover” on page 1-57

Review and Fix Non-Terminating Loop Checks

Follow one or more of these steps until you determine a fix for the **Non-terminating loop** check. There are multiple ways to fix the check. For a description of the check and code examples, see `Non-terminating loop`.

For the general workflow on reviewing checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

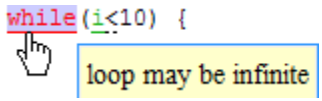
Step 1: Interpret Check Information

Place your cursor on the loop keyword such as `for` or `while`.

Obtain the following information from the tooltip:

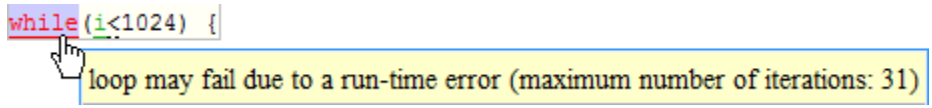
- Whether the loop is infinite or contains a run-time error.

  In the following example, it is likely that the loop is infinite.

- If the loop contains a run-time error, the number of loop iterations. Because Polyspace considers that execution stops when a run-time error occurs, from this number, you can determine which loop iteration contains the error.

  In the following example, it is likely that the loop contains a run-time error. The error is likely to occur on the 31st loop iteration.

Step 2: Determine Root Cause of Check

- If the loop is infinite, determine why the loop-termination condition is never satisfied.

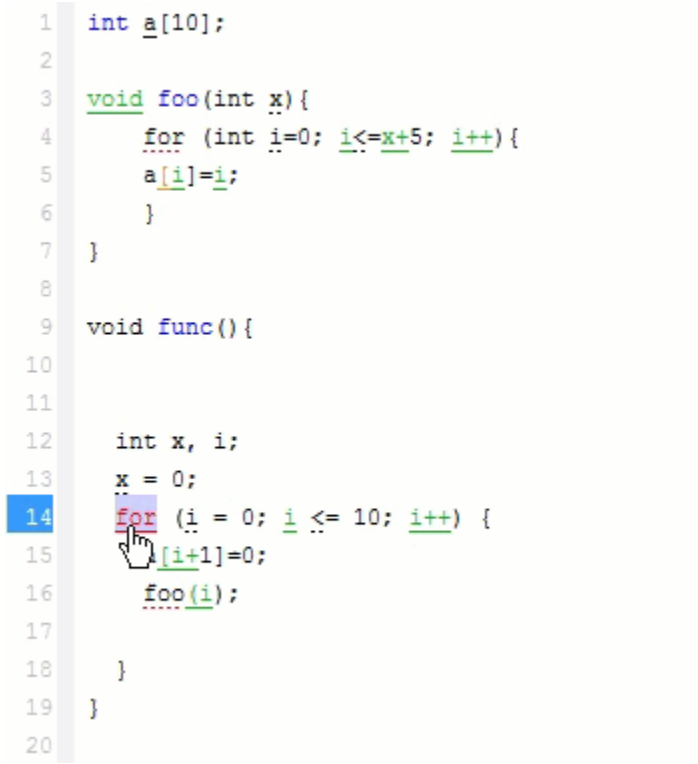
  If you deliberately have an infinite loop in your code, such as for cyclic applications, you can add a comment and justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

- If the loop contains a run-time error, identify the error that causes the **Non-terminating loop** check. Fix the error.

  In the loop body, the run-time error typically occurs as an orange check of another type on an operation. The check is orange and not red because the operation typically passes the check in the first few loop iterations and fails only in a later iteration. However, because the failure occurs every time the loop runs, the **Non-terminating loop** check on the loop keyword is red.

  For loops with few iterations, you can navigate directly from the loop keyword to the operation causing the run-time error.

  - To find the source of error, on the **Source** pane, select the red loop keyword. The **Result Details** pane shows the full history leading to the operation that causes the run-time error.
  - Navigate to the source of error in the loop body. Right-click the loop keyword and select **Go to Cause** if the option exists.



```
1  int a[10];
2
3  void foo(int x){
4      for (int i=0; i<=x+5; i++){
5      a[i]=i;
6      }
7  }
8
9  void func(){
10
11
12   int x, i;
13   x = 0;
14   for (i = 0; i <= 10; i++) {
15       [i+1]=0;
16     foo(i);
17
18   }
19 }
20
```

For a tutorial, see “Identify Loop Operation with Run-Time Error” on page 4-69.

Step 3: Look for Common Causes of Check

- If the loop is infinite:
  - Check your loop-termination condition.
  - Inside the loop body, see if you change at least one of the variables involved in the loop-termination condition.

    For instance, if the loop-termination condition is `while (count1 + count2 < count3)`, see if you are changing at least one of `count1`, `count2`, or `count3` in the loop.
  - If you are changing the variables involved in the loop-termination condition, see if you are changing them in the right direction.

For instance, if the loop termination condition is `while(i<10)` and you decrement `i` in the loop, the loop does not terminate. You must increment `i`.

- If the loop contains a run-time error:
  - If the loop control variable is an array index, see if you have an orange **Out of bounds array index** error in the loop body.
  - If the loop control variable is passed to a function, see if you can relate the red **Non-terminating loop** error to an orange error in the function body.

Identify Loop Operation with Run-Time Error

This tutorial shows how to interpret Polyspace Code Prover results that highlight a run-time error inside a loop.

If an error occurs in a loop, the error shows in the analysis results as a red **Non-terminating loop** check on the loop keyword (`for`, `while`, and so on).

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

The operation causing the error shows as an orange check in the loop. To distinguish this orange check from other orange checks in the loop, navigate directly from the red loop keyword to the operation responsible for the run-time error.

In this tutorial, a function is called in a loop. The function body contains another loop, which has an operation causing a run-time error. You trace from the first loop to the operation causing the run-time error.

1. Run verification on this code and open the results:

```
int a[100];

int f (int i);

void main() {
  int i,x=0;
  for (i = 0; i <= 10; i++) {
    x += f(i);
  }
}

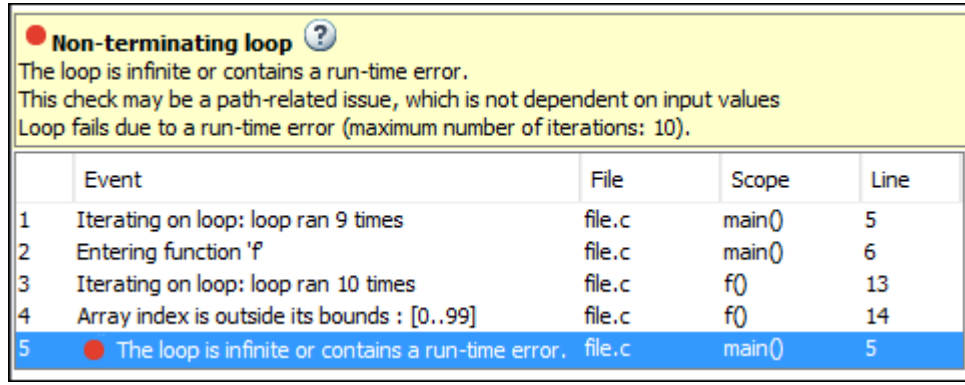
int f (int i) {
  int j, x;
  x = 0;
  for (j = 0; j <= 10; j++) {
    x += a[10 + (i * j)];
  }
  return x;
}
```

2. Select the red **Non-terminating loop** result. The **Source** pane highlights the `for` loop in `main`.

3 Trace from the `for` loop in `main` to the operation causing the error. The operation is `x+= a[10 + (i*j)]`. An **Out of bounds array index** error occurs when `i` is 9 and `j` is 10. The error shows in orange on the `[` operator.

To trace from the red `for` keyword to the orange array access operation:

- Navigate directly to the operation. Right-click the `for` keyword and select **Go to Cause**.
- See the full history from the `for` keyword to the array access operation. Select the red `for` keyword. The **Result Details** pane shows the history.



You can read the event history in sequence. The outer loop loop runs nine times without error. On the tenth iteration (`i=9`), the loop enters the function `f`. Inside `f`, the inner loop runs ten times without error. On the eleventh iteration (`j=10`), the array access causes an error.

You can use this information to determine how to fix the run-time error on the array access operation.

Note You can navigate directly to the root cause of an error for loops with only a small number of iterations.

See Also

`Non-terminating loop`

Related Examples

- “Review and Fix Non-Terminating Loop Checks” on page 4-65

More About

- “Orange Checks in Code Prover” on page 1-57

Review and Fix Null This-pointer Calling Method Checks

In this section...

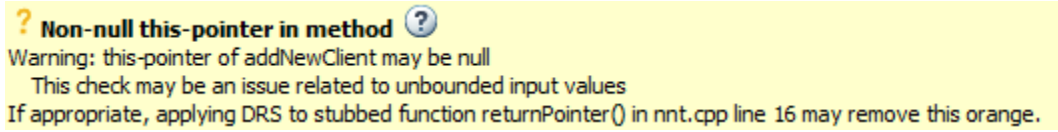
Follow one or more of these steps until you determine a fix for the **Null this-pointer calling method** check. For a description of the check and code examples, see `Null this-pointer calling method`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. The **Result Details** pane displays further information about the check.



You can see:

- The immediate cause of the check.

  In this example, the pointer used to call a method `addNewClient` can be `NULL`.
- The probable root cause of the check, if indicated.

  In this example, the check can be related to a stubbed function `returnPointer`.

Step 2: Determine Root Cause of Check

Find an execution path where the pointer is either assigned the value `NULL` or assigned values from an undefined function or unknown function inputs. In the latter case, the software assumes that the pointer can be `NULL`.

Select the check on the **Results List** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction and trace back to the root cause.
- If the **Result Details** pane shows the line number of probable cause for the check, in the Polyspace user interface, right-click the **Source** pane. Select **Go To Line**.
- If the **Result Details** pane does not lead you to the root cause, using the **Source** pane in the Polyspace user interface, find how the pointer used to call the method can be `NULL`.

  1. Right-click the pointer and select **Search For All References**.
  2. Find each previous instance where the pointer is assigned an address.
  3. For each instance, on the **Source** pane, place your cursor on the pointer. The tooltip indicates whether the pointer can be `NULL`.

     Possible fix: If the pointer can be `NULL`, place a check for `NULL` immediately after the assignment.

     ```
     if(ptr==NULL)
       /* Error handling*/
     else {
       .
       .
         }
     ```
  4. If the pointer is not `NULL`, see if the assignment occurs only in a branch of a conditional statement. Investigate when that branch does not execute.

     Possible fix: Assign a valid address to the pointer in all branches of the conditional statement.

Review and Fix Out of Bounds Array Index Checks

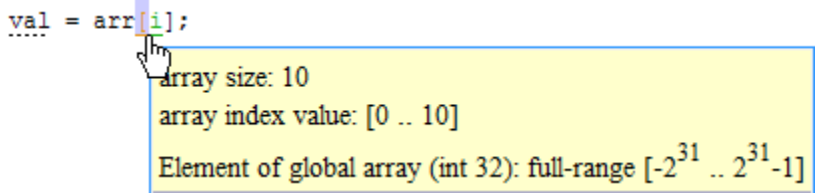
Follow one or more of these steps until you determine a fix for the **Out of bounds array index** check. There are multiple ways to fix the check. For a description of the check and code examples, see `Out of bounds array index`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

Place your cursor on the `[` symbol.



Obtain the following information from the tooltip:

- Array size. The allowed range for array index is 0 to (array size - 1).
- Actual range for array index

In the preceding example, the array size is 10. Therefore, the allowed range for the array index is 0 to 9. However, the actual range is 0 to 10.

Possible fix: To avoid the out of bounds array index, access the array only if the index is between 0 and (array size - 1).

```
#define SIZE 100
int arr[SIZE];
.
.
if(i<SIZE)
```

```
    val = arr[i]
else
    /*Error handling */
```

Step 2: Determine Root Cause of Check

If you want to know or change the array size, right-click the array variable and select **Go To Definition**, if the option exists. Otherwise, trace the data flow starting from the array index variable. Identify a point where you can constrain the index variable.

To trace the data flow, select the check, and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find previous instances of the array index variable.
  2. Browse through those instances. Find the instance where you constrain the array index variable to (array size - 1).

     Possible fix: If you do not find an instance where you constrain the index variable, specify a constraint for the variable. For example:

     ```
     if(index<SIZE)
         read(array[index]);
     ```
  3. Determine if the constraint applies to the instance where the **Out of bounds array index** error occurs.

     For example, you can constrain the index variable in a `for` loop using `for(index=0; index<SIZE; index++)`. However, you access the array outside the loop where the constraint does not apply.

     Possible fix: Investigate why the constraint does not apply. See if you have to constrain the index variable again.
  4. If the index variable is obtained from another variable, trace the data flow for the second variable. Determine if you have constrained that second variable to (array size - 1).

Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with  and read operations with  .
Function return value `ret=func();`	1 Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. 2 In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 4-104.

Step 3: Look for Common Causes of Check

Look for common causes of the **Out of bounds array index** check.

- See if you are starting the array index variable from 0.
- In the condition that constrains your array index variable, see if you use <= when you intended to use <.
- If a check occurs in or immediately after a `for` or `while` loop, determine if you have to reduce the number of runs of the loop.
- If you use the `sizeof` function to constrain your array, see if you are dividing `sizeof(array)` by `sizeof(array[0])` to obtain the array size.

  `sizeof(array)` returns the array size in bytes.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

For instance, you constrain the array index using a function whose definition you do not provide. Then:

1. Polyspace cannot determine that the array index variable is constrained.
2. When you use this variable as array index, an **Out of bounds array index** error can occur.
3. If you know that the variable is constrained to the array size, add a comment and justification describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

For instance, constraining a global variable in one function and using it as array index in a second function causes vulnerabilities in your code. If a new function is called between the previous two functions and modifies your global variable, the constraint no longer applies.

Review and Fix Overflow Checks

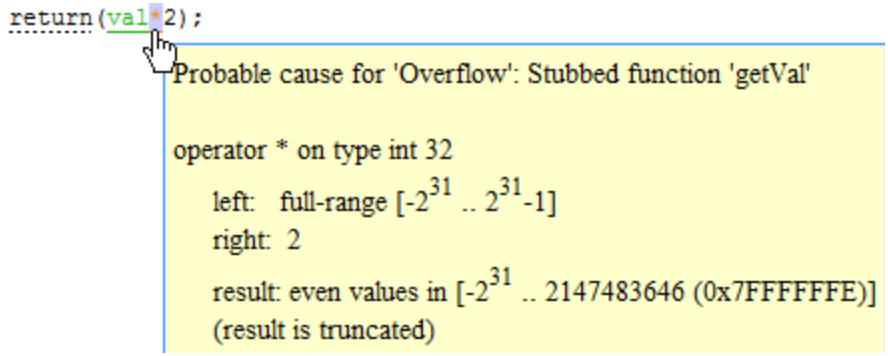
Follow one or more of these steps until you determine a fix for the **Overflow** check. There are multiple ways to fix the check. For a description of the check and code examples, see `Overflow`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

Place your cursor on the operation that overflows.



Obtain the following information from the tooltip:

- The operand variable you can constrain to avoid the overflow.

  In the preceding example, the left operand, `val`, has full range of values.

  Possible fix: To avoid the overflow, perform the multiplication only if `val` is in a certain range.

  ```
  if(val < INT_MAX/2)
      return(val*2);
  ```

```
else
    /* Alternate action */
```

- The probable root cause for overflow, if indicated in the tooltip.

  In the preceding example, the software identifies a stubbed function, `getVal`, as probable cause.

  Possible fix: To avoid the overflow, constrain the return value of `getVal`. For instance, specify that `getVal` returns values in a certain range, for example, `1..10`. For more information, see “Stubbed Functions” on page 6-9.

Step 2: Determine Root Cause of Check

Trace the data flow starting from the operand variable that you want to constrain. Identify a suitable point to specify your constraint.

In the following example, trace the data flow starting from `tempVal`.

```
val = func();
.
.
tempVal = val;
.
.
tempVal++ ;
```

In this example, you might find that:

1. `tempVal` obtains a full-range of values from `val`.

   Possible fix: Assign `val` to `tempVal` only if `val` is less than a certain value.

2. `val` obtains a full-range of values from `func`.

   Possible fix: Constrain the return value of `func`, either in the body of `func` or through the Polyspace Constraint Specification interface, if `func` is stubbed. For more information, see “Stubbed Functions” on page 6-9.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.

- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.
- Otherwise:

  1. Find the previous write operation on the operand variable.

     Example: The previous write operation on `tempVal` is `tempVal=val`.
  2. At the previous write operation, identify a new variable to trace back.

     Place your cursor on the variables involved in the write operation to see their values. The values help you decide which variable to trace.

     Example: At `tempVal=val`, you find that `val` has a full-range of values. Therefore, you trace `val`.
  3. Find the previous write operation on the new variable. Continue tracing back in this way until you identify a point to specify your constraint.

     Example: The previous write operation on `val` is `val=func()`. You can choose to specify your constraint on the return value of `func`.

Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with  and read operations with  .
Function return value `ret=func();`	1 Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. 2 In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 4-104.

Tip To distinguish between integer and float overflows, use the **Detail** column on the **Results List** pane. Click the column header so that integer and float overflows are grouped together. If you do not see the **Detail** column, right-click any other column header and enable this column.

Step 3: Look for Common Causes of Check

If the operation causing the overflow occurs in a loop or in the body of a recursive function:

- See if you can reduce the number of loop runs or recursions.
- See if you can place an exit condition in the loop or recursive function before the operation overflows.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

For instance, you are using a volatile variable in your code. Then:

1. Polyspace assumes that the volatile variable is full-range at every step in the code.
2. An operation using that variable can overflow and is therefore orange.
3. If you know that the variable takes a smaller range of values, add a comment and justification in your code describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Detect Overflows in Buffer Size Computation

If you are computing the size of a buffer from `unsigned` integers, for the **Overflow mode for unsigned integer** option, instead of the default value `allow`, use `forbid`. Using this option helps you detect an overflow at the buffer computation stage. Otherwise, you might see an error later due to insufficient buffer. This option is available on the **Check Behavior** node under **Code Prover Verification** in the **Configuration** pane.

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

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

int get_value(void);

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

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

1. Create a Polyspace project and add `display.c` to the project.
2. On the **Configuration** pane, select the following options:
   - **Target & Compiler**: From the **Target processor type** drop-down list, select a type with 16-bit `int` such as `c167`.
   - **Check Behavior**: From the **Overflow mode for unsigned integer** drop-down list, select `allow`.
3. Run the verification and open the results.

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

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

4 From the **Overflow mode for unsigned integer** drop-down list, select `forbid`.

5 Polyspace detects a red **Overflow** error in the computation `num_items * sizeof(int)`.

See Also

Polyspace Analysis Options

Polyspace Results
`Illegally dereferenced pointer` | `Overflow`

Review and Fix Return Value Not Initialized Checks

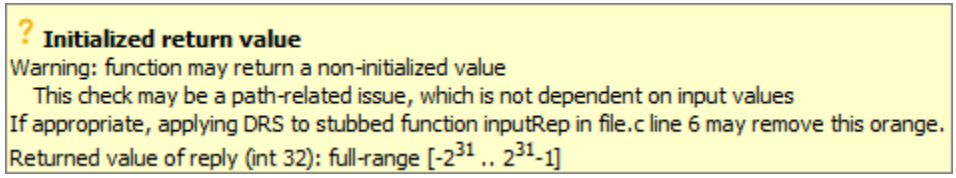
Follow one or more of these steps until you determine a fix for the **Return value not initialized** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Return value not initialized`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. On the **Result Details** pane, view further information about the check.



View the probable cause of check, if mentioned on the **Result Details** pane.

In the preceding example, the software identifies a stubbed function, `inputRep`, as probable cause.

Possible fix: To avoid the check, constrain the argument or return value of `inputRep`. For instance, specify that `inputRep` returns values in a certain range, for example, `1..10`. For more information, see “Stubbed Functions” on page 6-9.

Step 2: Determine Root Cause of Check

Determine the root cause of the check in the function body. You can perform the following steps in the Polyspace user interface only.

1. Navigate to the function definition.

   Right-click the function call containing the check. Select **Go To Definition**, if the option exists.

2. In the function body, check if a `return` statement occurs before the closing brace of the function.

3. If a `return` statement does not exist:

   a. On the **Search** pane, search for the word `return`, or manually scroll through the function body and look for `return` statements.

   b. For each `return` statement, determine if the statement appears in a scope smaller than function scope.

      For instance, a `return` statement occurs only in one branch of an `if-else` statement.

   Possible fix: See if you can place the `return` statement at the end of the function body. For instance, replace the following code

```
int func(int ch) {
    switch(ch) {
        case 1: return 1;
        break;
        case 2: return 2;
        break;
    }
}
```

   with

```
int func(int ch) {
    int temp;
    switch(ch) {
        case 1: temp = 1;
        break;
        case 2: temp = 2;
        break;
    }
    return temp;
}
```

   For information on how to enforce this practice, see `Number of Return Statements`.

Step 3: Look for Common Causes of Check

Look for common causes of the **Return value not initialized** check.

- See if the `return` statements appear in `if-else`, `for` or `while` blocks. Identify conditions when the function does not enter the block.

  For instance, the function might not enter a `while` block for certain function inputs.

  Possible fix:

  - See if you can place the `return` statement at the end of the function body.
  - Otherwise, determine how to avoid the condition when the function does not enter the block.

    For instance, if a function does not enter a block for certain inputs, see if you must pass different inputs.
- See if you have code constructs such as `goto` that interrupt the normal control flow. See if there are conditions when `return` statements in your function cannot be reached because of these code constructs.

  Possible fix:

  - Avoid `goto` statements. For information on how to enforce this practice, see `Number of Goto Statements`.
  - Otherwise, determine how to avoid the condition when `return` statements in your function cannot be reached.

Step 4: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

For instance, you have a `return` statement in branches of an `if-elseif` block but you do not have the final `else` block. The condition you are testing in the `if-elseif` blocks involve variables obtained from an undefined function. Then:

1. Polyspace assumes that for certain values of those variables, none of the `if-elseif` blocks are entered.

2. Therefore, it is possible that the `return` statements are not reached.
3. If you know that those values of the variables do not occur, add a comment and justification describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Disabling This Check

You can disable this check. If you disable this check, Polyspace assumes the following about a function return value if the function is missing `return` statements:

- If the return value is a non-pointer variable, it has full-range of values allowed by its type.
- If the return value is a pointer, it can be `NULL`-valued or point to a memory block at an unknown offset.

For more information, see `Disable checks for non-initialization (-disable-initialization-checks)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Review and Fix Uncaught Exception Checks

Follow one or more of these steps until you determine a fix for the **Uncaught exception** check. For a description of the check and code examples, see `Uncaught exception`.

Step 1: Interpret Check Information

Select the check on the **Results List** pane. On the **Result Details** pane, view further information about the check.

The message for a red or orange **Uncaught exception** check typically states one of these reasons.

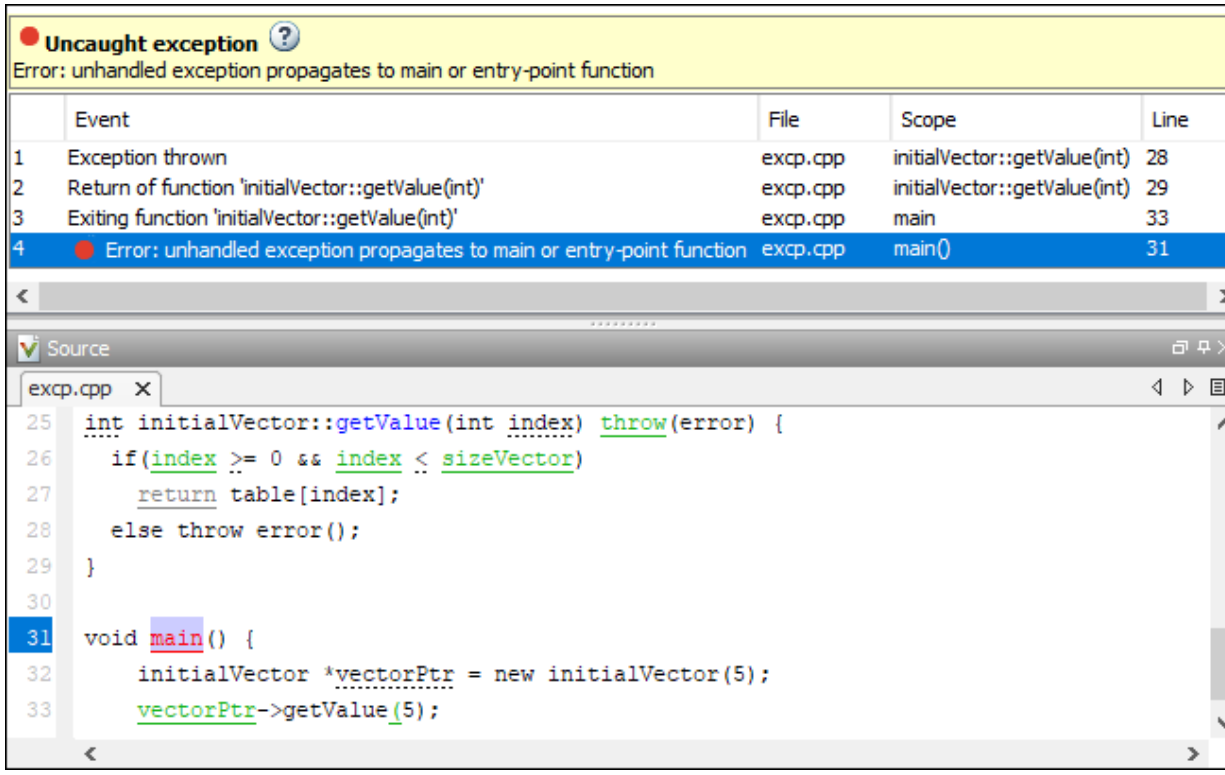
Message	What This Means
Unhandled exception propagates to main or entry-point function.	An exception is thrown and not handled in a `catch` block. The exception escapes to the `main`.
Call to *`typeName`* throws during "catch" parameter construction.	Creating the `catch` parameter invokes a constructor. The constructor throws an exception.
Throw during destructor or delete.	A destructor throws an exception.
Exception raised is not specified in the throw list.	A function signature indicates that only certain exception types can be thrown or an exception cannot be thrown. However, in the function body, an operation violates this contract and throws an exception of a different type.

Step 2: Determine Root Cause of Check

The most common root cause is that an exception propagates up the function call hierarchy from its origin to the `main` function.

In the event traceback associated with the check, you see the origin of the exception and one path up the function call tree to the `main` or another entry-point function. Click each event to navigate to the corresponding point in the source code.

In this example, the exception is thrown in the method `getValue()` which is called from the `main` function through an object of type `initialVector`.



The event list shows these points in the code:

1. The statement that throws an exception.
2. The return from the function where the exception is thrown, in this case, the `getValue` method.
3. The function call site in the `main`.
4. The `main` function.

Using this event list, you can trace how the exception escapes and place a `try-catch` block to handle the exception. For instance, you can place the call:

```
vectorPtr->getValue(5)
```

in a `try-catch` block. In the `catch` block, you can catch an exception of type `error`.

Review and Fix Unreachable Code Checks

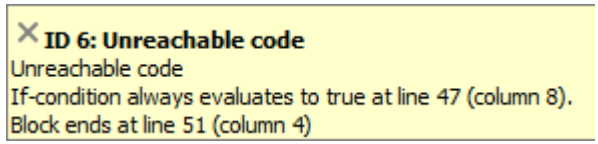
Follow one or more of these steps until you determine a fix for the **Unreachable code** check. There are multiple ways to fix this check. For a description of the check and code examples, see `Unreachable code`.

If you determine that the check represents defensive code, add a comment and justification in your result or code explaining why you did not change your code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

Step 1: Interpret Check Information

1. Select the check on the **Results List** or **Source** pane.
2. View the message on the **Result Details** pane.

   The message explains why the block of code is unreachable.

3. A code block is usually unreachable when the condition that determines entry into the block is not satisfied. See why the condition is not satisfied.

   a. On the **Source** pane, place your cursor on the variables involved in the condition to determine their values.

   b. Using these values, see why the condition is not satisfied.

   Note Sometimes, a condition itself is redundant. For example, it is always true or coupled:

   - Through an `||` operator to another condition that is always true.
   - Through an && operator to another condition that is always false.

   For example, in the following code, the condition `x%2==0` is redundant because the first condition `x>0` is always true.

   ```
   assert(x>0);
   if(x>0 || x%2 == 0)
   ```

If a condition is redundant, instead of a block of code, the condition itself is marked gray.

Step 2: Determine Root Cause of Check

Trace the data flow for each variable involved in the condition.

In the following example, trace the data flow for `arg`.

```
void foo(void) {
    int x=0;
    .
    .
    bar(x);
    .
    .
}

void bar(int arg) {
    if(arg==0) {
        /*Block 1*/
    }
    else {
        /*Block 2*/
    }
}
```

You might find that `bar` is called only from `foo`. Since the only argument of `bar` has value 0, the `else` branch of `if(arg==0)` is unreachable.

Possible fix: If you do not intend to call `bar` elsewhere and know that the values passed to `bar` will not change, you can remove the `if-else` statement in `bar` and retain only the content of `Block 1`.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click on the **Source** pane. Select **Go To Line**.

- Otherwise, for each variable involved in the condition, find previous instances and trace back to the root cause of check. For more information on common root causes, see “Step 3: Look for Common Causes of Check” on page 4-96.

  Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with  and read operations with  .

Variable	How to Find Previous Instances of Variable
Function return value `ret=func();`	1 Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. 2 In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 4-104.

Step 3: Look for Common Causes of Check

Look for common causes of the **Unreachable code** check.

- Look for the following in your `if` tests:
  - You are testing the variables that you intend to test.

    For example, you might have a local variable that shadows a global variable. You might be testing the local variable when you intend to test the global one.
  - You are using parentheses to impose the sequence in which you want operations in the `if` test to execute.

    For example, `if((!a && b) || c)` imposes a different sequence of operations from `if(!(a && b) || c)`. Unless you use parentheses, the operations obey operator precedence rules. The rules can cause the operations to execute in a sequence that you did not intend.
  - You are using = and == operators in the right places.

  Possible fix: Correct errors if any.

  - Use Polyspace Bug Finder to check for common defects such as `Invalid use of = operator` and `Variable shadowing`.
  - To avoid errors due to incorrect operation sequence, check for violations of `MISRA C:2012 Rule 12.1`.

- See if you are performing a test that you have performed previously.

  The redundant test typically occurs on the argument of a function. The same test is performed both in the calling and called function.

  ```
  void foo(void) {
      if(x>0)
          bar(x);
      .
      .
  }

  void bar(int arg) {
      if(arg==0) {
      }
  }
  ```

  Possible fix: If you intend to call `bar` later, for example, in yet unwritten code, or reuse `bar` in other programs, retain the test in `bar`. Otherwise, remove the test.

- See if your code is unreachable because it follows a `break` or `return` statement.

  Possible fix: See if you placed the `break` or `return` statement at an unintended place.

- See if the section of unreachable code exists because you are following a coding standard. If so, retain the section.

  For example, the `default` block of a `switch-case` statement is present to capture abnormal values of the `switch` variable. If such values do not occur, the block is unreachable. However, you might violate a coding standard if you remove the block.

- See if the unreachable code is related to an orange check earlier in the code. Following an orange check, Polyspace normally terminates execution paths that contain an error. Because of this termination, code following an orange check can appear gray.

  For example, Polyspace places an orange check on the dereference of a pointer `ptr` if you have not vetted `ptr` for `NULL`. However, following the dereference, it considers that `ptr` is not `NULL`. If a test `if(ptr==NULL)` follows the dereference of `ptr`, Polyspace marks the corresponding code block unreachable.

  For more examples, see:

  - “Gray Check Following Orange Check” on page 1-51

An exception to this behavior is overflow. If you specify the appropriate **Overflow mode for signed integer** or **Overflow mode for unsigned integer**, Polyspace wraps the result of an overflow and does not terminate the execution paths. See `Overflow mode for signed integer (-signed-integer-overflows)` or `Overflow mode for unsigned integer (-unsigned-integer-overflows)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

- “Left operand of left shift may be negative”

Possible fix: Investigate the orange check. In the above example, investigate why the test `if(ptr==NULL)` occurs after the dereference and not before.

Review and Fix User Assertion Checks

Follow one or more of these steps until you determine a fix for the **User assertion** check. There are multiple ways to fix this check. For a description of the check and code examples, see `User assertion`.

Sometimes, especially for an orange check, you can determine that the check does not represent a real error but a Polyspace assumption that is not true for your code. If you can use an analysis option to relax the assumption, rerun the verification using that option. Otherwise, you can add a comment and justification in your result or code.

For the general workflow that applies to all checks, see “Interpret Polyspace Code Prover Access Results” on page 1-3.

How to use this check: Typically you use `assert` statements during debugging to check if a condition is satisfied at the current point in your code. For instance, if you expect a variable `var` to have values in a range `[1,10]` at a certain point in your code, you place the following statement at that point:

```
assert(var >=1 && var <= 10);
```

Polyspace statically determines whether the condition is satisfied.

Therefore, you can judiciously insert `assert` statements that test for requirements from your code.

- A red result for the **User assertion** check indicates that the requirement is definitely not met.
- An orange result for the **User assertion** check indicates that the requirement is possibly not met.

Step 1: Determine Root Cause of Check

Trace the data flow for each variable involved in the `assert` statement.

In the following example, trace the data flow for `myArray`.

```
int* getArray(int numberOfElements) {
    .
    .
    arrayPtr = (int*) malloc(numberOfElements);
```

```
    .
    .
    return arrayPtr;
}
void func() {
    int* myArray = getArray(10);
    assert(myArray!=NULL);
    .
    .
}
```

In this example, it is possible that in `getArray`, the return value of `malloc` is not checked for `NULL`.

Possible fix: If you expect a certain requirement, see if you have tests that enforce the requirement. In this example, if you expect `getArray` to return a non-`NULL` value, in `getArray`, test the return value of `malloc` for `NULL`.

To trace the data flow, select the check and note the information on the **Result Details** pane.

- If the **Result Details** pane shows the sequence of instructions that lead to the check, select each instruction.
- If the **Result Details** pane shows the line number of probable cause for the check, right-click in the **Source** pane. Select **Go To Line**. Enter the line number.
- Otherwise, for each variable involved in the condition, find previous instances and trace back to the root cause of the check. For more information on common root causes, see “Step 3: Look for Common Causes of Check” on page 4-96.

  Depending on the variable, use the following navigation shortcuts to find previous instances. You can perform the following steps in the Polyspace user interface only.

Variable	How to Find Previous Instances of Variable
Local Variable	Use one of the following methods: • Search for the variable. 1 Right-click the variable. Select **Search For All References**. All instances of the variable appear on the **Search** pane with the current instance highlighted. 2 On the **Search** pane, select the previous instances. • Browse the source code. 1 Double-click the variable on the **Source** pane. All instances of the variable are highlighted. 2 Scroll up to find the previous instances.
Global Variable Right-click the variable. If the option **Show In Variable Access View** appears, the variable is a global variable.	1 Select the option **Show In Variable Access View**. On the **Variable Access** pane, the current instance of the variable is shown. 2 On this pane, select the previous instances of the variable. Write operations on the variable are indicated with  and read operations with  .
Function return value `ret=func();`	1 Find the function definition. Right-click `func` on the **Source** pane. Select **Go To Definition**, if the option exists. If the definition is not available to Polyspace, selecting the option takes you to the function declaration. 2 In the definition of `func`, identify each `return` statement. The variable that the function returns is your new variable to trace back.

You can also determine if variables in any operation are related from some previous operation. See “Find Relations Between Variables in Code” on page 4-104.

Step 2: Look for Common Causes of Check

1. If the check is orange and occurs in a function, see if the function is called multiple times. Determine if the assertion fails only on certain calls. For those calls, navigate to the caller body and see if you have tests that enforce your assertion requirement.

   - To see the callers of a function, select the function name on the **Source** pane. All callers appear on the **Call Hierarchy** pane. Select a caller name to navigate to it in your source.
   - To determine if a subset of calls cause the orange check, use the option `Sensitivity context (-context-sensitivity)`.. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

2. If you have tests that enforce the assertion requirement, see if:

   - The assertion statement is within the scope of the tests.
   - You modify the test variables between the test and the assertion.

   For instance, the test `if(index < SIZE)` enforces that `index` is less than `SIZE`. However, the assertion `assert(index < SIZE)` can fail if:

   - It occurs outside the `if` block.
   - Before the assertion, you modify `index` in the `if` block.

   Possible fix: Consider carefully whether you must meet your assertion requirements. If you must meet those requirements, place tests that enforce your requirement. After the tests, avoid modifying the test variables.

Step 3: Trace Check to Polyspace Assumption

See if you can trace the orange check to a Polyspace assumption that occurs earlier in the code. If the assumption does not hold true in your case, add a comment or justification in your result or code. See “Address Polyspace Results Through Bug Fixes or Comments” on page 2-2.

For instance, after a function call, you assert a relation between two variables. Then:

1. Depending on the depth of the function call and the complexity of your code, Polyspace can sometimes approximate the variable ranges. Because of the approximation, the software cannot establish if the relation holds and produces an orange **User assertion** check.
2. Run dynamic tests on the orange check to determine if the assertion fails.
3. Try to reduce your code complexity and rerun the verification. Otherwise, add a comment and a justification in your result or code describing why you did not change your code.

For more information, see “Code Prover Analysis Assumptions”.

Note Before justifying an orange check, consider carefully whether you can improve your coding design.

Find Relations Between Variables in Code

This tutorial shows how to determine if the variables in an arbitrary operation in your code are previously related.

For instance, consider this operation:

```
return(var1 - var2);
```

- In your IDE, you can place breakpoints to stop execution and determine the values of `var1` and `var2` *for a specific run*.
- In Polyspace, after you analyze your code, the tooltips on `var1` and `var2` show their range of values *for all runs* that the verification considers.

However, the range information is not enough to determine if the variables are related. You must perform additional steps to determine the relation.

Insert Pragma to Determine Variable Relation

In this example, consider the operation highlighted. You cannot tell from a quick glance if `wheel_speed` and `wheel_speed_old` are related. However, this information is crucial to understand a possible bug in subsequent operations.

```
#define MAX_SPEED 120
#define TEST_TIME 10000

int wheel_speed;
int wheel_speed_old;

int out;

int update_speed(int new_speed) {
    if(new_speed < MAX_SPEED)
    return new_speed;
    else
    return MAX_SPEED;
}

void increase_speed(void)
{

  int temp, index=1;
```

```
  while(index<TEST_TIME) {
    temp = wheel_speed - wheel_speed_old;

    if(index > 1) {
       if (temp < 0)
         out = 1;
       else
         out = 0;
    }

    wheel_speed_old = update_speed(wheel_speed);
    index++;
  }

}
```

To understand why you need the relation between `wheel_speed` and `wheel_speed_old` and how to find the relation:

1. Constrain the range of the variable `wheel_speed` to 0..100 for the Polyspace analysis. Use the analysis option `Constraint setup (-data-range-specifications)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.
2. Run analysis on this code and open the results. Select the gray **Unreachable code** check.

   ```
   if (temp < 0)
            out = 1;
   ```

   The check indicates that the variable `temp` is nonnegative. `temp` comes from the previous operation:

   ```
   temp = wheel_speed - wheel_speed_old;
   ```
3. See the range of `wheel_speed` and `wheel_speed_old`. Place your cursor on these variables. You see these ranges:

   - `wheel_speed`: 0..100
   - `wheel_speed_old`: Full range of an `int` variable.

   It is not clear from these ranges why `wheel_speed - wheel_speed_old` is always nonnegative. You have to find out if the variables are somehow related.

4. Insert a pragma before the line where you want the variable relation. Add the following line just before `if(temp < 0)`:

   ```
   #pragma Inspection_Point wheel_speed wheel_speed_old
   ```

5. Rerun the analysis and open the results. Place your cursor on `wheel_speed_old` in the line that you added.

   The tooltip confirms that `wheel_speed` and `wheel_speed_old` are related:

   ```
   wheel_speed_old <= wheel_speed
   ```

6. To find how the two variables got related, search for all instances of `wheel_speed_old`. On the **Source** pane, right-click `wheel_speed_old` and select **Search For All References**.

   Browse through the instances. In this case, you see that the line which relates `wheel_speed` and `wheel_speed_old` is:

   ```
   wheel_speed_old = update_speed(wheel_speed);
   ```

   This line ensures that after the first run of the `while` loop, `wheel_speed_old` is less than or equal to `wheel_speed_old`. The branch `if(index > 1)` is entered from the second run onwards. In this branch, the relation between `wheel_speed` and `wheel_speed_old` is reflected through the gray **Unreachable code** check.

Tip Ignore the details of the relation shown in the tooltip. Use the tooltip to confirm if certain variables are related. Then, search for instances of the variable to find how they are related.

Further Exploration

You can use the pragma `Inspection_Point` to determine the relation between variables at any point in the code. You can enter as many variables as you want in the `#pragma` statement:

```
#pragma Inspection_Point var1 var2 ... varn
```

Try this technique on other examples. For instance, select **Help** > **Examples** > **Code_Prover_Example.psprj**. Group the results by file. In the file `single_file_analysis.c`, you see this code:

```
if (output_v7 >= 0) {

  #pragma Inspection_Point output_v7 s8_ret
  saved_values[output_v7] = s8_ret;
  return s8_ret;
}
```

If you place your cursor on `s8_ret` in the last two statements, you find that the ranges of `s8_ret` are different. Find out what changed between the two statements.

Hint: The tooltip in the `#pragma` statement indicates that the variable `s8_ret` is related to the variable `output_v7`. Note the orange check that reduces the range of `output_v7`.

See Also

Related Examples

- “Interpret Polyspace Code Prover Access Results” on page 1-3

Coding Rule Sets and Concepts

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 1-78
- “Software Quality Objective Subsets (AC AGC)” on page 1-84

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 5-3

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

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

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	The text *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* precedes each of the following messages: • 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 Specification
1.1 (cont.)		The text *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* precedes each of the following messages: • 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.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• Too many enumeration constants: N_1. The limit is N_0.	

Language Extensions

N.	MISRA Definition	Messages in report file	Polyspace Specification
2.1	Assembly language shall be encapsulated and isolated.	Assembly language shall be encapsulated and isolated.	No warnings if code is encapsulated in the following: • `asm` functions or `asm pragma` • Macros
2.2	Source code shall only use /* */ 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 /* shall not be used within a comment	The character sequence /* shall not appear within a comment.	This rule violation is also raised when the character sequence /* inside a C++ comment. **Note**: This rule cannot be annotated in the source code.

Documentation

Rule	MISRA Definition	Messages in report file	Polyspace Specification
3.4	All uses of the *#pragma* directive shall be documented and explained.	All uses of the #pragma 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 `Allowed pragmas (-allowed-pragmas)`. 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 Code Prover or Polyspace Code Prover Server

Character Sets

N.	MISRA Definition	Messages in report file	Polyspace Specification
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.	\<character> 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 Specification
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.
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.	• 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)	Warning when a tag name is reused as another identifier name
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)	Warning when a static name is reused as another identifier name Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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)	Warning when an `idf` in a namespace is reused in another namespace
5.7	No identifier name should be reused.	{identifier}'%s' should not be reused. (already used as {identifier} at %s:%d)	No violation reported when: • 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 Specification
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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.	• 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	*typedefs* that indicate size and signedness should be used in place of the basic types	typedefs 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 *unsigned int* or *signed int*.	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type *signed int* shall be at least 2 bits long.	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 Specification
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	• 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 Specification
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	• 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

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.4	If objects or functions are declared more than once their types shall be compatible.	• 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.	Violations of this rule might be generated during the link phase. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.5	There shall be no definitions of objects or functions in a header file	• 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.	Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that: • Do not have initializers. • Do not have storage class specifiers, or have the `static` 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** `addrescape`.
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 Specification
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.	Restricted to explicit extern declarations (tentative definitions are ignored). Polyspace considers that variables or functions declared `extern` in a non-header file violate this rule. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.9	Definition: An identifier with external linkage shall have exactly one external definition.	• Procedure/Global variable XX multiply defined. • Forbidden multiple tentative definitions for object XX • Global variable has multiple tentative definitions • Undefined global variable XX	The checker flags multiple definitions only if the definitions occur in different files. No warnings appear on predefined symbols. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
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.	Assumes that 8.1 is not violated. No warning if 0 uses. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.11	The *static* 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.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	

Initialization

N.	MISRA Definition	Messages in report file	Polyspace Specification
9.1	All automatic variables shall have been assigned a value before being used.		Checked during code analysis. Violations displayed as Non-initialized variable results. 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 `Verification level (-to)`. See .the documentation for Polyspace Code Prover or Polyspace Code Prover Server for more on analysis options and how to check for coding standard violations.
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
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 Specification
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: • 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	• 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.	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. An expression of bool or enum types has int as underlying type. Plain char may have signed or unsigned underlying type (depending on Polyspace 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 is not token into account and it assumes that only signed | unsigned int are used for bitfield (Rule 6.4). This rule violation is not produced on operations involving pointers. No violation reported when: • The implicit conversion is a type widening, without

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• 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.	change of signedness of integer • The expression is an argument expression or a return expression No violation reported when the following are true: • 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 the constant value or the result of the operation. • The expression is an argument expression or a return expression or an operand expression of a non-bitwise operator. 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if • 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	• 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.	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. No violation reported when: • 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 Specification

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.	• 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 `i` but not the second. In the first assignment, a composite expression `i+1` 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.
			
			`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 */`
			• 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
			• 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 is not token 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 *unsigned char* or *unsigned short*, 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.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.6	The “U” suffix shall be applied to all constants of *unsigned* types	No explicit 'U suffix on constants of an unsigned type.	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`. For example, when the size of the `int` and `long int` data types is 32 bits, the coding rule checker will report a violation of rule 10.6 for the following line: `int a = 2147483648;` There is a difference between decimal and hexadecimal constants when `int` and `long int` are not the same size.

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	Casts and implicit conversions involving a function pointer. Casts or implicit conversions from `NULL` or `(void*)0` do not give any warning.
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.	There is also a warning on qualifier loss This rule maps to **ISO/IEC TS 17961 ID** `alignconv`.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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** `alignconv`.
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 *const* or *volatile* qualification from the type addressed by a pointer	A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer	Extended to all conversions

Expressions

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	• The value of '*sym*' depends on the order of evaluation. • The value of volatile '*sym*' depends on the order of evaluation because of multiple accesses.	Rule 12.2 check assumes that no assignment in expressions that yield a Boolean values (rule 13.1). The expression is a simple expression of symbols. `i = i ++;` is a violation, but `tab[2] = tab[2]++;` is not a violation.
12.3	The `sizeof` operator should not be used on expressions that contain side effects.	The `sizeof` operator should not be used on expressions that contain side effects.	No warning on volatile accesses

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.4	The right hand operand of a logical && or || operator shall not contain side effects.	The right hand operand of a logical && or || operator shall not contain side effects.	No warning on volatile accesses
12.5	The operands of a logical && or || shall be primary-expressions.	• operand of logical && is not a primary expression • operand of logical || is not a primary expression • The operands of a logical && or || shall be primary-expressions.	During preprocessing, violations of this rule are detected on the expressions in #if directives. Allowed exception on associatively (a && b && c), (a || b || c).

N.	MISRA Definition	Messages in report file	Polyspace Specification

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 !).	• 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 '?:'.
			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.
			
			Some operators may return Boolean-like expressions, for example, `(var == 0)`.
			
			Consider the following code:
			
			`unsigned char flag;`
			`if (!flag)`
			
			The rule checker reports a violation of rule 12.6:
			
			`Operand of '!' logical operator should be effectively Boolean.`
			
			The operand `flag` is not a Boolean but an `unsigned char`.
			
			To be compliant with rule 12.6, the code must be rewritten either as
			
			`if (!( flag != 0))`
			
			or
			
			`if (flag == 0)`
			
			The use of the option `-boolean-types` may increase or decrease the

N.	MISRA Definition	Messages in report file	Polyspace Specification
			number of warnings generated.
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed	• [~/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	The underlying type for an integer is signed when: • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
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.	• 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)..	The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63 Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	• Unary - on operand of unsigned underlying type XX. • Minus operator applied to an expression whose underlying type is unsigned	The underlying type for an integer is signed when: • 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.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	Warning when: • 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: `union { float f; int i; } …`
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 Specification
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 Specification
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 `-boolean-types` 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 directs tests only.
13.4	The controlling expression of a *for* statement shall not contain any objects of floating type	The controlling expression of a for statement shall not contain any objects of floating type	If *for* index is a variable symbol, checked that it is not a float.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.5	The three expressions of a *for* statement shall be concerned only with loop control	• 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 for loops are allowed: (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 for 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 *for* loop for iteration counting should not be modified in the body of the loop.	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop.	Detect only direct assignments if the for loop index is known and if it is a variable symbol.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.7	Boolean operations whose results are invariant shall not be permitted	• 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.	During compilation, check comparisons with at least one constant operand. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results. • Bug Finder flags some violations of this rule through the `Dead code` and `Useless if` checkers. • Code Prover does not use gray code to flag violations of this rule. In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. See .the documentation for Polyspace Code Prover or Polyspace Code Prover Server for more on analysis options and how to check for coding standard violations.. The rule violation appears when you check whether an `enum` variable value lies between its lower and upper bound. The violation appears even if you increment or decrement the variable outside its bounds, for

N.	MISRA Definition	Messages in report file	Polyspace Specification

			instance, in this `for` loop condition:
			`enum ec {RED, BLUE, GREEN} col;`
			`for(col=RED; col<=GREEN; col++)`
			`{}`
			An `enum` 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 enum to an integer before the comparison, for instance:
			`enum ec {RED, BLUE, GREEN} col;`
			`for(col=RED; (int)col<=GREEN; col+`
			`{}`

Control Flow

N.	MISRA Definition	Messages in report file	Polyspace Specification
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	All non-null statements shall either: • have at least one side effect however executed, or • cause control flow to change	

N.	MISRA Definition	Messages in report file	Polyspace Specification
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	We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when: • 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 *goto* statement shall not be used.	The goto statement shall not be used.	
14.5	The *continue* statement shall not be used.	The continue statement shall not be used.	
14.6	For any iteration statement there shall be at most one *break* 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	
14.8	The statement forming the body of a *switch, while, do while* or *for* statement shall be a compound statement	• 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	

N.	MISRA Definition	Messages in report file	Polyspace Specification
14.9	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	• 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 *if else if* constructs should contain a final *else* clause.	All if else if constructs should contain a final else clause.	

Switch Statements

N.	MISRA Definition	Messages in report file	Polyspace Specification

15.0	The MISRA C switch syntax shall be used.	switch statements syntax normative restrictions.	Warning on declarations or any statements before the first switch case.
			
			Warning on label or jump statements in the body of switch cases.
			
			On the following example, the rule is displayed in the log file at line 3:
			
			`1 ...`
			`2 switch(index) {`
			`3  var = var + 1;`
			`// RULE 15.0`
			`// violated`
			`4case 1: ...`
			
			The code between switch statement and first case is checked as dead code by Polyspace. It follows ANSI standard behavior.
			
			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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a *switch* 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 *break* 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 *switch* statement shall be the *default* clause	The final clause of a switch statement shall be the default clause	
15.4	A *switch* 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 `-boolean-types` may increase the number of warnings generated.
15.5	Every *switch* statement shall have at least one *case* clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	Done by Polyspace software (Use the call graph in Polyspace Code Prover). Polyspace also partially checks this rule during the compilation phase.
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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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 *void*.	Functions with no parameters shall be declared with parameter type void.	Definitions are also checked.
16.6	The number of arguments passed to a function shall match the number of parameters.	• 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 pointer to `const` if the pointer is not used to modify the addressed object.	Pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.	Warning if a non-`const` 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 `const` 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.
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.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	Warning if a non-`void` function is called and the returned value is ignored. No warning if the result of the call is cast to `void`. No check performed for calls of `memcpy`, `memmove`, `memset`, `strcpy`, `strncpy`, `strcat`, or `strncat`.

Pointers and Arrays

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	Warning on: • Operations on pointers. (`p +I`, `I+p`, and `p-I`, where `p` is a pointer and `I` an integer). • Array indexing on nonarray pointers.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	Warning when assigning address to a global variable, returning a local variable address, or returning a parameter address. This rule maps to **ISO/IEC TS 17961 ID** `accfree`.

Structures and Unions

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	• 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 Specification
19.1	#include statements in a file shall only be preceded by other preprocessors directives or comments	#include statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a #include directive is preceded by other things than preprocessor directives, comments, spaces or “new lines”.

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.2	Nonstandard characters should not occur in header file names in #include directives	• A message is displayed on characters ', " or /* between < and > in #include <filename> • A message is displayed on characters ', or /* between " and " in #include "filename"	
19.3	The *#include* directive shall be followed by either a <filename> or "filename" sequence.	• '#include' expects "FILENAME" or <FILENAME> • '#include_next' expects "FILENAME" or <FILENAME>	

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	We assume that a macro definition does not violate this rule when it expands to: • 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 #undefd within a block.	• 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.

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.8	A function-like macro shall not be invoked without all of its arguments	• 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)
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.	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`. 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,`.

N.	MISRA Definition	Messages in report file	Polyspace Specification
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.	

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,
			`#ifndef <control macro>`
			`#define <control macro>`
			`<contents> #endif`
			or,
			`#ifndef <control macro>`
			`#error ...`
			`#else`
			`#define <control macro>`
			`<contents> #endif`
			it is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
19.17	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	• '#elif' not within a conditional. • '#else' not within a conditional. • '#elif' not within a conditional. • '#endif' not within a conditional. • unbalanced '#endif'. • unterminated '#if' conditional. • unterminated '#ifdef' conditional. • unterminated '#ifndef' conditional.	

Standard Libraries

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	• The macro '<name> shall not be redefined. • The macro '<name> shall not be undefined.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.2	The names of standard library macros, objects and functions shall not be reused.	Identifier XX should not be used.	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**. Tentative definitions are considered as definitions. For objects with file scope, tentative definitions are declarations that: • Do not have initializers. • Do not have storage class specifiers, or have the `static` specifier

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	Warning for argument in library function call if the following are all true: • Argument is a local variable • Local variable is not tested between last assignment and call to the library function • Library function is a common mathematical function • Corresponding parameter of the library function has a restricted input domain. The library function can be one of the following : `sqrt`, `tan`, `pow`, `log`, `log10`, `fmod`, `acos`, `asin`, `acosh`, `atanh`, or `atan2`. Bug Finder and Code Prover check this rule differently. The analysis can produce different results.
20.4	Dynamic heap memory allocation shall not be used.	• The macro '<name> 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 errno shall not be used	The error indicator errno shall not be used	Assumes that rule **20.2** is not violated

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.6	The macro *offsetof*, in library <stddef.h>, shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	Assumes that rule **20.2** is not violated
20.7	The *setjmp* macro and the *longjmp* function shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the longjmp 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 <signal.h> shall not be used.	• The macro '<name> 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 <stdio.h> shall not be used in production code.	• The macro '<name> 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 atof, atoi and atoll from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the atof, atoi and atoll 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 abort, exit, getenv and system from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the abort, exit, getenv and system functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.12	The time handling functions of library <time.h> shall not be used.	• The macro '<name> 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 Specification
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: • static verification tools/ techniques; • dynamic verification tools/ techniques; • explicit coding of checks to handle runtime faults.		Done by Polyspace. 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 `Verification level (-to)`. See .the documentation for Polyspace Code Prover or Polyspace Code Prover Server for more on analysis options and how to check for coding standard violations..

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 “**Polyspace Specification**” column describes the reason each rule is not checked.

Environment

Rule	Description	Polyspace Specification
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/compilers/ 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.

Language Extensions

Rule	Description	Polyspace Specification
2.4 (Advisory)	Sections of code should not be “commented out”	One way a tool can check this rule is to determine if the code compiles when commented out sections are uncommented. However, such checking can be expensive and inaccurate.

Documentation

Rule	Description	Polyspace Specification
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	Polyspace Specification
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 4.7, 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 Code Prover or Polyspace Code Prover Server .

There are additional subsets of MISRA C:2012 guidelines defined by Polyspace called Software Quality Objectives (SQO) 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 1-88.

2. MISRA and MISRA C are registered trademarks of MIRA Ltd., held on behalf of the MISRA Consortium.

See Also

See Also

More About

- “MISRA C:2012 Directives and Rules”

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	`bool` or `_Bool` (defined in `stdbool.h`) If you define a boolean type through a `typedef`, you must specify this type name before coding rules checking. For more information, see `Effective boolean types (-boolean-tyoes)`. For more on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server .
Essentially character	`char`
Essentially enum	named `enum`
Essentially signed	signed `char`, signed `short`, signed `int`, signed `long`, signed `long long`
Essentially unsigned	unsigned `char`, unsigned `short`, unsigned `int`, unsigned `long`, unsigned `long long`
Essentially floating	`float`, `double`, `long double`

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
Directive 4.4	Advisory	Advisory	Sections of code should not be “commented out”
Directive 4.12	Required	Required	Dynamic memory allocation shall not be used

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 1-97.

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...

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	Polyspace Specification
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.	

N.	Category	MISRA Definition	Polyspace Specification
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.
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	Polyspace Specification
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	Polyspace Specification
2-2-1	Document	The character set and the corresponding encoding shall be documented.	To observe this rule, check your compiler documentation.

N.	Category	MISRA Definition	Polyspace Specification
2-7-2	Required	Sections of code shall not be "commented out" using C-style comments.	One way a tool can check this rule is to determine if the code compiles when commented out sections are uncommented. However, such checking can be expensive and inaccurate.
2-7-3	Advisory	Sections of code should not be "commented out" using C++ comments.	One way a tool can check this rule is to determine if the code compiles when commented out sections are uncommented. However, such checking can be expensive and inaccurate.

Expressions

N.	Category	MISRA Definition	Polyspace Specification
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.		MISRA Definition	Polyspace Specification
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	Polyspace Specification
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.		MISRA Definition	Polyspace Specification
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	Polyspace Specification
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.	

N.	Category	MISRA Definition	Polyspace Specification
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	Polyspace Specification
17-0-3	Required	The names of standard library functions shall not be overridden.	
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 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 C++ Coding Rules

Supported JSF C++ Coding Rules

Code Size and Complexity

N.	JSF++ Definition	Polyspace Specification
1	Any one function (or method) **will** contain no more than 200 logical source lines of code (L-SLOCs).	Message in report file: <*function name*> has <*num*> logical source lines of code.
3	All functions **shall** have a cyclomatic complexity number of 20 or less.	Message in report file: <*function name*> has cyclomatic complexity number equal to <*num*>.

Environment

N.	JSF++ Definition	Polyspace Specification
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: <*digraph*>. Reports the digraph. If the rule level is set to warning, the digraph will be allowed even if it is not supported in `-compiler iso`.
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.	

N.	JSF++ Definition	Polyspace Specification
15	Provision **shall** be made for run-time checking (defensive programming).	Done with checks in the software.

Libraries

N.	JSF++ Definition	Polyspace Specification
17	The error indicator `errno` **shall not** be used.	`errno` should not be used as a macro or a global with external "C" linkage.
18	The macro `offsetof`, in library `<stddef.h>`, **shall not** be used.	`offsetof` should not be used as a macro or a global with external "C" linkage.
19	`<locale.h>` and the `setlocale` function **shall not** be used.	`setlocale` and `localeconv` should not be used as a macro or a global with external "C" linkage.
20	The `setjmp` macro and the `longjmp` function **shall not** be used.	`setjmp` and `longjmp` 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.	`signal` and `raise` 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 `atof`, `atoi` and `atol` from library `<stdlib.h>` **shall not** be used.	`atof`, `atoi` and `atol` should not be used as a macro or a global with external "C" linkage.
24	The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` **shall not** be used.	`abort`, `exit`, `getenv` and `system` 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.	`clock`, `difftime`, `mktime`, `asctime`, `ctime`, `gmtime`, `localtime` and `strftime` should not be used as a macro or a global with external "C" linkage.

Pre-Processing Directives

N.	JSF++ Definition	Polyspace Specification
26	Only the following preprocessor directives **shall** be used: `#ifndef`, `#define`, `#endif`, `#include`.	
27	`#ifndef`, `#define` and `#endif` **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 `#if !defined`, `#pragma once`, `#ifdef`, and missing `#define`.
28	The `#ifndef` and `#endif` 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 `#ifndef`.
29	The `#define` 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: • 29.1 : The `#define` preprocessor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros.
30	The `#define` preprocessor directive **shall not** be used to define constant values. Instead, the `const` qualifier **shall** be applied to variable declarations to specify constant values.	Reports `#define` of simple constants.
31	The `#define` preprocessor directive **will** only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of `#define` that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The `#include` preprocessor directive **will** only be used to include header (*.h) files.	

Header Files

N.	JSF++ Definition	Polyspace Specification
33	The `#include` directive **shall** use the `<filename.h>` 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 Specification
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.
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.	

N.	JSF++ Definition	Polyspace Specification
47	Identifiers **will not** begin with the underscore character '_'.	
48	Identifiers **will not** differ by: • 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: • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by the presence/absence of the underscore character. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by a mixture of case. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by letter `O`, 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: • 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.

N.	JSF++ Definition	Polyspace Specification
51	All letters contained in function and variables names **will** be composed entirely of lowercase letters.	Messages in report file: • 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: • 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`.
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.

N.	JSF++ Definition	Polyspace Specification
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.	Messages in report file: • 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.

N.	JSF++ Definition	Polyspace Specification
63	Spaces will not be used around '.' or '->', nor between unary operators and operands.	Reports when the following characters are not directly connected to a white space: • . • ->\n• ! • ~ • - • ++ • — Note that a violation will be reported for "." used in float/double definition.

Classes

N.	JSF++ Definition	Polyspace Specification
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.

N.	JSF++ Definition	Polyspace Specification
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: • `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 Specification
81	The assignment operator shall handle self-assignment correctly	Reports when copy assignment body does not begin with "`if (this != arg)`" A violation is not raised if an empty `else` statement follows the `if`, or the body contains only a return statement. A violation is raised when the `if` statement is followed by a statement other than the return statement.
82	An assignment operator **shall** return a reference to `*this`.	The following operators should return `*this` on method, and `*first_arg` on plain function. `operator=operator+=operator-=operator*=operator >>=operator <<=operator /=operator %=operator |=operator &=operator ^=prefix operator++ prefix operator--` Does not report when no return exists. No special message if type does not match. Messages in report file: • An assignment operator shall return a reference to `*this`. • 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 Specification
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: • Multiple inheritance on public implementation shall not be allowed: *<public_base_class>* is not an interface. • Multiple inheritance on protected implementation shall not be allowed : *<protected_base_class_1>*. • *<protected_base_class_2>* 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 Specification
98	Every nonlocal name, except `main()`, **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 Specification
104	A template specialization **shall** be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Polyspace Specification
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.	

N.	JSF++ Definition	Polyspace Specification
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 `sizeof <= 2 * sizeof(int)`. Does not report for copy-constructor.
119	Functions **shall** not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	Direct recursion is reported statically. Indirect recursion reported through the software. Message in report file: Function <*F*> shall not call directly itself.
121	Only functions with 1 or 2 statements **should** be considered candidates for inline functions.	Reports inline functions with more than 2 statements.

Comments

N.	JSF++ Definition	Polyspace Specification
126	Only valid C++ style comments (`//`) **shall** be used.	
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).	Reports when a file does not begin with two comment lines. **Note**: This rule cannot be annotated in the source code.

Declarations and Definitions

N.	JSF++ Definition	Polyspace Specification
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 Specification
136	Declarations should be at the smallest feasible scope.	Reports when: • 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. **Note** • 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.	
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.	Reports all duplicate declarations inside a translation unit. Reports when the declaration localization is not the same in all translation units.
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 Specification
142	All variables **shall** be initialized before use.	Done with Non-initialized variable checks in the software.

N.	JSF++ Definition	Polyspace Specification
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 Specification
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 Specification
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.

N.	JSF++ Definition	Polyspace Specification
151.1	A string literal shall not be modified.	Report when a `char*`, `char[]`, or `string` type is used not as `const`. A violation is raised if a string literal (for example, “ “) is cast as a non `const`.

Variables

N.	JSF++ Definition	Polyspace Specification
152	Multiple variable declarations **shall not** be allowed on the same line.	

Unions and Bit Fields

N.	JSF++ Definition	Polyspace Specification
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 Specification
157	The right hand operand of a && or `||` operator shall not contain side effects.	Assumes rule 159 is not violated. Messages in report file: • The right hand operand of a && operator shall not contain side effects. • The right hand operand of a `||` operator shall not contain side effects.

N.	JSF++ Definition	Polyspace Specification
158	The operands of a logical && or || **shall** be parenthesized if the operands contain binary operators.	Messages in report file: • 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: • 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.	
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 Specification
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: • 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 Specification
177	User-defined conversion functions **should** be avoided.	Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor. Additional message for constructor case: This constructor should be flagged as "explicit".
178	Down casting (casting from base to derived class) **shall** only be allowed through one of the following mechanism: • 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, dynamic_cast 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 dynamic_cast.

N.	JSF++ Definition	Polyspace Specification

180	Implicit conversions that may result in a loss of information **shall not** be used.	Reports the following implicit casts :
		
		`integer => smaller integer`
		`unsigned => smaller or eq signed`
		`signed => smaller or eq un-signed`
		`integer => float float => integer`
		
		Does not report for cast to `bool` reports for implicit cast on constant done with the option `-scalar-overflows-checks signed-and-unsigned`
		
		Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

181	Redundant explicit casts **will not** be used.	Reports useless cast: `cast T to T`. Casts to equivalent `typedefs` 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 `float->int` conversions. Does not report implicit ones.
185	C++ style casts (`const_cast`, `reinterpret_cast`, and `static_cast`) **shall** be used instead of the traditional C-style casts.	

Flow Control Standards

N.	JSF++ Definition	Polyspace Specification
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.

N.	JSF++ Definition	Polyspace Specification
187	All non-null statements **shall** potentially have a side-effect.	
188	Labels **will not** be used, except in switch statements.	
189	The `goto` statement **shall** not be used.	
190	The `continue` statement **shall not** be used.	
191	The `break` statement **shall not** be used (except to terminate the cases of a switch statement).	
192	All `if`, `else if` constructs will contain either a final `else` clause or a comment indicating why a final `else` clause is not necessary.	`else if` should contain an `else` clause.
193	Every non-empty `case` clause in a switch statement **shall** be terminated with a `break` statement.	
194	All `switch` statements that do not intend to test for every enumeration value **shall** contain a final `default` clause.	Reports only for missing `default`.
195	A `switch` expression **will** not represent a Boolean value.	
196	Every `switch` statement **will** have at least two cases and a potential `default`.	
197	Floating point variables **shall not** be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a `for` loop **will** perform no actions other than to initialize the value of a single `for` loop parameter.	Reports if `loop` parameter cannot be determined. Assumes Rule 200 is not violated. The `loop variable` parameter is assumed to be a variable.
199	The increment expression in a `for` 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.

N.	JSF++ Definition	Polyspace Specification
200	Null initialize or increment expressions in `for` loops **will not** be used; a `while` loop will be used instead.	
201	Numeric variables being used within a *for* 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 Specification
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: • 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: • 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.

N.	JSF++ Definition	Polyspace Specification
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	Reports when: • Variable is written more than once in an expression • Variable is read and write in sub-expressions • Volatile variable is accessed more than once **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 Specification
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 Specification
208	C++ exceptions **shall not** be used.	Reports `try`, `catch`, `throw spec`, and `throw`.

Portable Code

N.	JSF++ Definition	Polyspace Specification
209	The basic types of `int`, `short`, `long`, `float` and `double` **shall not** be used, but specific-length equivalents should be `typedef`'d accordingly for each compiler, and these type names used in the code.	Only allows use of basic types through direct `typedefs`.

N.	JSF++ Definition	Polyspace Specification
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:`p + Ip - Ip++p--p+=p-=` Allows `p[i]`.

Unsupported JSF++ Rules

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: • 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: • 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	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.) 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.

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.

N.	JSF++ Definition
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	The invariant for a class should be: • 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.
91	Public inheritance will be used to implement “is-a” relationships.

N.	JSF++ Definition
92	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: • 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. 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.
93	“has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	Elements from a namespace should be selected as follows: • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	Templates shall be reviewed as follows: **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.
117	Arguments should be passed by reference if NULL values are not possible: • **117.1** – An object should be passed as `const T&` if the function should not change the value of the object. • **117.2** – An object should be passed as `T&` if the function may change the value of the object.
118	Arguments should be passed via pointers if NULL values are possible: • **118.1** – An object should be passed as `const T*` if its value should not be modified. • **118.2** – An object should be passed as `T*` if its value may be modified.
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.
122	Trivial accessor and mutator functions should be inlined.
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
127	Code that is not used (commented out) shall be deleted. **Note**: This rule cannot be annotated in the source code.
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.

N.	JSF++ Definition
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	A single operation with side-effects shall only be used in the following contexts: **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.
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.

Approximations Used During Verification

Why Polyspace Verification Uses Approximations

Polyspace Code Prover uses *static verification* to prove the absence of run-time errors. Static verification derives the dynamic properties of a program without actually executing it. Static verification differs significantly from other techniques such as run-time debugging because the verification does not rely on a specific test case or set of test cases. The properties obtained from static verification are true for *all* executions of your program[5].

Static verification uses representative approximations of software operations and data. For instance, consider the following code:

```
for (i=0 ; i<1000 ; ++i)  {
    tab[i] = foo(i);
}
```

To check that the variable `i` never overflows the range of `tab`, one approach can be to consider each possible value of `i`. This approach requires a thousand checks.

In static verification, the software models a variable by its domain. In this case, the software models that `i` belongs to the static interval, [0..999]. Depending on the complexity of the data, the software uses more elaborate models such as convex polyhedrons or integer lattices for this purpose.

An approximation, by definition, leads to information loss. For instance, the verification loses the information that `i` is incremented by one every cycle in the loop. However, even without this information, it is possible to ensure that the range of `i` is smaller than the range of `tab`. Only one check is required to establish this property. Therefore, static verification is more efficient compared to traditional approaches.

When performing approximations, the verification does not compromise with exhaustiveness. The reason is that the approximations performed are upper

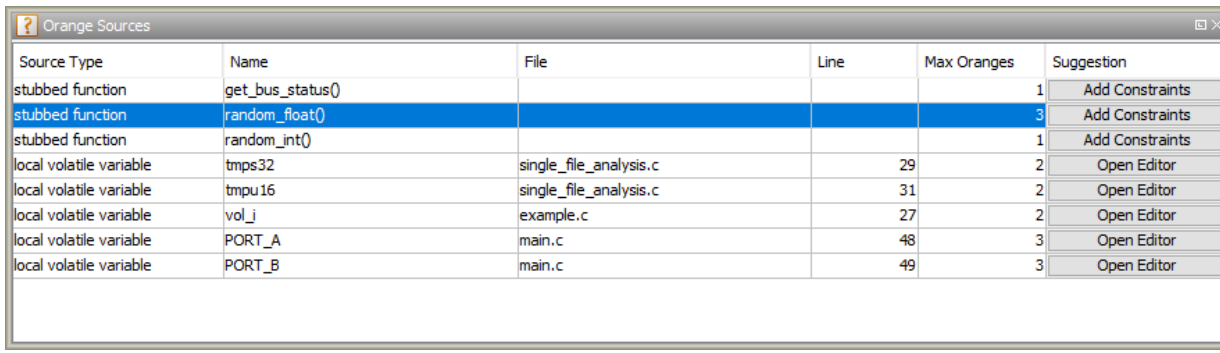
5. The properties obtained from static verification hold true only if you execute your program under the same conditions that you specified through the analysis options. For instance, the default verification assumes that pointers obtained from external sources are non-null. Unless you specify the option `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`, the verification results are obtained under this assumption. They might not hold true during program execution if the assumption is invalidated and a null pointer is obtained from an external source. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

approximations or over-approximations. In other words, the computed domain of a variable is a superset of its actual domain.

Orange Sources

The **Orange Sources** pane shows unconstrained sources such as volatile variables and stubbed functions that can lead to multiple orange checks (unproven results) in a Code Prover analysis. If you constrain an orange source, you can address several orange checks together. To see the **Orange Sources** pane, click the  button on the **Result Details** pane.

The sources essentially indicate variables whose values cannot be determined from your code. The variables can be inputs to functions whose call context is unknown or return values of undefined functions. Code Prover assumes that these variables take the full range of values allowed by their data type. This broad assumption can lead to one or more orange checks in the subsequent code.



Source Type	Name	File	Line	Max Oranges	Suggestion
stubbed function	get_bus_status()			1	Add Constraints
stubbed function	random_float()			3	Add Constraints
stubbed function	random_int()			1	Add Constraints
local volatile variable	tmps32	single_file_analysis.c	29	2	Open Editor
local volatile variable	tmpu16	single_file_analysis.c	31	2	Open Editor
local volatile variable	vol_i	example.c	27	2	Open Editor
local volatile variable	PORT_A	main.c	48	3	Open Editor
local volatile variable	PORT_B	main.c	49	3	Open Editor

For instance, in this example, if the function `random_float` is not defined, you see three orange **Overflow** checks.

```
static void Close_To_Zero(void)
{
    float xmin = random_float();
    float xmax = random_float();
    float y;

    if ((xmax - xmin) < 1.0E-37f) { /* Overflow 1 */
        y = 1.0f;
    } else {
        /* division by zero is impossible here */
        y = (xmax + xmin) / (xmax - xmin); /* Overflows 2 and 3 */
    }
}
```

The function `random_float` is therefore an orange source that causes at most three orange checks.

Using the **Orange Sources** pane, you can:

- Review all orange checks originating from the same source.

  In the preceding example, if you select the function `random_float`, the results list shows only the three orange checks caused by this source. See “Filter Using Orange Sources” on page 3-6.
- Constrain variable ranges by specifying external constraints or through additional code. Constraining the range of an orange source can remove several orange checks that come from overapproximation. The remaining orange checks indicate real issues in your code.

  In the preceding example, you can constrain the return value of `random_float`.

For efficient review, click the **Max Oranges** column header to sort the orange sources by number of orange checks that result from the source. Constrain the sources with more orange checks before tackling the others.

Constrain Orange Sources

How you constrain variable ranges and work around the default Polyspace assumptions depends on the type of orange source:

Stubbed function

If the definition of a function is not available to the Polyspace analysis, the function is stubbed. The analysis makes several assumptions about stubbed functions. For instance, the return value of a stubbed function can take any value allowed by its data type.

See “Stubbed Functions” on page 6-9 for assumptions about stubbed functions and how to work around them.

Volatile variable

If a variable is declared with the `volatile` specifier, the analysis assumes that the variable can take any value allowed by its data type at any point in the code.

See “Volatile Variables” on page 6-18 to work around around this assumption.

Extern variable

If a variable is declared with the `extern` specifier but not defined elsewhere in the code, the analysis assumes that the variable can take any value within its data type range before it is first assigned.

Determine where the variable is defined and why the definition is not available to the analysis. For instance, you might have omitted an include folder from the analysis.

Function called by the main generator

If your code does not contain a `main` function, a `main` function is generated for the analysis. By default, the generated `main` function calls uncalled functions with inputs that can take any value allowed by their data type.

Variable initialized by the main generator

If your code does not contain a `main` function, a `main` function is generated for the analysis. By default, in each function called by the generated `main`, a global variable can take any value within its data type range before it is first assigned.

Variable set in a permanent range by the main generator

If you explicitly constrain a global variable to a specific range in the permanent mode, the analysis assumes that the variable can take any value within this range at any point in the code.

Absolute address

If a pointer is assigned an absolute address, the analysis assumes that the pointer dereference leads to a range of potential values determined by the pointer data type.

See `Absolute address usage` for examples of absolute address usage and corresponding Code Prover assumptions. To remove this assumption and flag all uses of absolute address, use the option `-no-assumption-on-absolute-addresses`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Sometimes, more than one orange source can be responsible for an orange check. If you plug an orange source but do not see the expected disappearance of an orange check, consider if another source is also responsible for the check.

See Also

More About

- “Orange Checks in Code Prover” on page 1-57
- “Filter Using Orange Sources” on page 3-6

Variable Ranges

If Polyspace cannot determine a variable value from the code, it assumes that the variable has a full range of values allowed by its type.

For instance, for a variable of integer type, to determine the minimum and maximum value allowed, Polyspace uses the following criteria:

- The C standard specifies that the range of a signed *n*-bit integer-type variable must be at least $[-(2^{n-1}-1), 2^{n-1}-1]$.

  The **Target processor type** that you specify determines the number of bits allocated for a certain type. For more information, see `Target processor type (-target)` For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server..
- Polyspace assumes that your target uses the two's complement representation for signed integers. The software uses this representation to determine the exact range of a variable. In this representation, the range of a signed *n*-bit integer-type variable is $[-2^{n-1}, 2^{n-1}-1]$.

For example, for an `i386` processor:

- A `char` variable has 8 bits. The C standard specifies that the range of the `char` variable must be at least [-127,127].
- Using the two's complement representation, Polyspace assumes that the exact range of the `char` variable is [-128,127].

To determine the range that Polyspace assumes for a certain type:

1. Run verification on this code. Replace *`type`* with the type name such as `int`.

   ```
   type getVal(void);
   void main() {
           type val = getVal();
   }
   ```

2. Open your verification results. On the **Source** pane, place your cursor on `val`.

   The tooltip provides the range that Polyspace assumes for *`type`*. Since `getVal` is not defined, Polyspace assumes that the return value of `getVal` has full range of values allowed by *`type`*.

Stubbed Functions

The verification stubs functions that are not defined in your source code or that you choose to stub. For a stubbed function:

- The verification makes certain assumptions about the function return value and other side effects of the function.

  You can fine-tune the assumptions by specifying constraints.
- The verification ignores the function body if it exists. Operations in the function body are not checked for run-time errors.

If the verification of a function body is imprecise and causes many orange checks when you call the function, you can choose to stub the function. To reduce the number of orange checks, you stub the function, and then constrain the return value of the function and specify other side effects.

To stub functions, you can use these options:

- `Functions to stub (-functions-to-stub)`: Specify functions that you want stubbed.
- `Generate stubs for Embedded Coder lookup tables (-stub-embedded-coder-lookup-table-functions)`: Stub functions that contain lookup tables in code generated from models using Embedded Coder®.
- `-function-behavior-specifications`: Stub functions that correspond to a standard function that Polyspace recognizes.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

If you use the first option to stub a function, you constrain the function return value and model other side effects by specifying constraints. If you want to specify constraints more fine-grained than the ones available through the Polyspace constraint specification interface, define your own stubs. If you use the other options to stub functions, the software itself constrains the function return value and models its side effects appropriately.

The verification makes the following assumptions about the arguments of stubbed functions.

Function Return Value

Assumptions

The verification assumes that:

- The variable returned by the function takes the full range of values allowed by its data type.

  If the function returns an `enum` variable, the variable value is in the range of the enum. For instance, if an `enum` type takes values {0,5,-1,32} and a stubbed function has that return type, the verification assumes that the function returns values in the range -1..32.
- If the function returns a pointer, the pointer is not NULL and safe to dereference. The pointer does not point to dynamically allocated memory or another variable in your code.
- C++ specific assumptions: The operator `new` returns allocated memory. Operators such as `operator=`, `operator+=`, `operator--`(prefixed version) or `operator<<` returns:

  - A reference to `*this`, if the operator is part of a class definition.

    For instance, if an operator is defined as:

    ```
    class X {
      X& operator=(const X& arg) ;
    };
    ```

    It returns a reference to `*this` (the object that calls the operator). The object that calls the operator or its data members have the full range of values allowed by their type.
  - The first argument, if the operator is not part of a class definition.

    For instance, if an operator is defined as:

    ```
    X& operator+=(X& arg1, const X& arg2) ;
    ```

    It returns `arg1`. The object that `arg1` refers to or its data members have the full range of values allowed by their type.

  Functions declared with `__declspec(no_return)` (Visual Studio®) or `__attribute__ ((noreturn))` (GCC) do not return.

How to Change Assumptions

You can change the default assumptions about the function return value.

- If the function returns a non-pointer variable, you can constrain its range. Use the option `Constraint setup (-data-range-specifications)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

  Through the constraint specification interface, you can specify an absolute range `[`*`min`*`..`*`max`*`]`. To specify more complicated constraints, write a function stub.

  For instance, an undefined function has the prototype:

  ```
  int func(int ll, int ul);
  ```

  Suppose you know that the function return value lies between the first and the second arguments. However, the software assumes full range for the return value because the function is not defined. To model the behavior that you want and reduce orange checks from the imprecision, write a function stub as follows:

  ```
  int func(int ll, int ul) {
       int ret;
       assert(ret>=ll && ret <=ul);
       return ret;
  }
  ```

  Provide the function stub in a separate file for verification. The verification uses your stub as the function definition.

  If the definition of `func` exists in your code and you want to override the definition because the verification of the function body is imprecise, embed the actual definition and the stub in a `#ifdef` statement:

  ```
  #ifdef POLYSPACE
  int func(int ll, int ul) {
       int ret;
       assert(ret>=ll && ret <=ul);
       return ret;
  }
  #else
  int func(int ll, int ul) {
      /*Your function body */
  ```

```
}
#endif
```

Define the macro `POLYSPACE` by using the option `Preprocessor definitions (-D)`. The verification uses your stub instead of the actual function definition.

- If the function returns a pointer variable, you can specify that the pointer might be NULL.
  - To specify this assumption for all stubbed functions, use the option `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`.
  - To specify this assumption for specific stubbed functions, use the option `Constraint setup (-data-range-specifications)`.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Function Arguments That are Pointers

Assumptions

The verification assumes that:

- If the argument is a pointer, the function can write any value to the object that the pointer points to. The range of values is constrained by the argument data type alone.

  If the pointed object is previously uninitialized, the function can leave the object uninitialized.

  For instance, in this example, the verification assumes that the stubbed function `stubbedFunc` writes any possible value to `val`. Therefore, the assertion is orange.

  ```
  void stubbedFunc(int*);

  void main() {
      int val=0, *ptr=&val;
      stubbedFunc(ptr);
      assert(val==0);
  }
  ```
- If the argument is a pointer to a structure, the function can write any value to the structure fields. The range of values is constrained only by the data type of the fields.

If the pointed structure is previously uninitialized or partially initialized, the function can leave the object uninitialized or partially initialized.

- If the argument is a pointer to another pointer, the function can write any value to the object that the second pointer points to (C code only). This assumption continues to arbitrary depths of a pointer hierarchy.

  For instance, suppose that a pointer `**pp` points to another pointer `*p`, which points to an `int` variable `var`. If a stubbed function takes `**p` as argument, the verification assumes that following the function call, `var` has any `int` value. `*p` can point to anywhere in allocated memory or can point to `var` but does not point to another variable in the code.
- If the argument is a function pointer, the function that it points to gets called (C code only).

  For instance, in this example, the stubbed function `stubbedFunc` takes a function pointer `funcPtr` as argument. `funcPtr` points to `func`, which gets called when you call `stubbedFunc`.

  ```
  typedef int (*typeFuncPtr) (int);

  int func(int x){
        return x;
  }

  int stubbedFunc(typeFuncPtr);

  void main() {
      typeFuncPtr funcPtr = (typeFuncPtr)(&func);
      int result = stubbedFunc(funcPtr);
  }
  ```

  If the function pointer takes another function pointer as argument, the function that the second function pointer points to gets stubbed.

How to Change Assumptions

You can constrain the range of the argument that is passed by reference. Use the option `Constraint setup (-data-range-specifications)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Through the constraint specification interface, you can specify an absolute range `[`*`min`*`..`*`max`*`]`. To specify more complicated constraints, write a function stub.

For instance, an undefined function has the prototype:

```
void func(int *x, int ll, int ul);
```

Suppose you know that the value written to `x` lies between the second and the third arguments. However, the software assumes full range for the value of `*x` because the function is not defined. To model the behavior that you want and reduce orange checks from the imprecision, write a function stub as follows:

```
void func(int *x, int ll, int ul) {
     assert(*x>=ll && *x <=ul);
}
```

Provide the function stub in a separate file for verification. The verification uses your stub as the function definition.

If the definition of `func` exists in your code and you want to override the definition because the verification of the function body is imprecise, embed the actual definition and the stub in a `#ifdef` statement:

```
#ifdef POLYSPACE
void func(int *x, int ll, int ul) {
     assert(*x>=ll && *x <=ul);
}
#else
void func(int *x, int ll, int ul) {
     /* Your function body */
}
#endif
```

Define the macro `POLYSPACE` by using the option `Preprocessor definitions (-D)`. The verification uses your stub instead of the actual function definition. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Global Variables

Assumptions

The verification assumes that the function stub does not modify global variables.

How to Change Assumptions

To model write operations on a global variable, write a function stub.

For instance, an undefined function has the prototype:

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

Suppose you know that the function writes the value 0 or 1 to a global variable `glob`. To model the behavior that you want, write a function stub as follows:

```
void func(void) {
     volatile int randomVal;
     if(randomVal)
          glob = 0;
     else
          glob = 1;
}
```

Provide the function stub in a separate file for verification. The verification uses your stub as the function definition.

If the definition of `func` exists in your code and you want to override the definition because the verification of the function body is imprecise, embed the actual definition and the stub in a `#ifdef` statement as follows:

```
#ifdef POLYSPACE
void func(void) {
     volatile int randomVal;
     if(randomVal)
          glob = 0;
     else
          glob = 1;
}
#else
void func(void) {
     /* Your function body */
}
#endif
```

Define the macro `POLYSPACE` using the option `Preprocessor definitions (-D)`. The verification uses your stub instead of the actual function definition. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Initialization of Global Variables

If your code contains a `main` function, the verification considers that global variables are initialized according to ANSI C standards. The default values are:

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

To disable this assumption and raise an error for global variables not explicitly initialized, use the option `Ignore default initialization of global variables (-no-def-init-glob)` For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server..

If your code does not have a `main` function, the software begins verifying each uncalled function with the assumption that global variables have full range value, constrained only by their data type.

The software uses the dummy function `_init_globals()` to initialize global variables. The `_init_globals()` function is the first function implicitly called in the `main` function.

Consider the following code in the application `gv_example.c`.

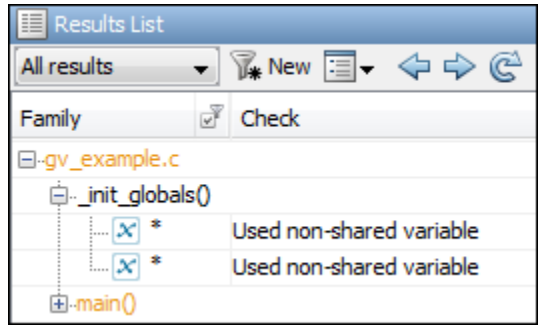
```
extern int func(int);  /* External function */

/* Global variables initialized in _init_globals() */
/* before the execution of main() procedure     */
int garray[3] = {1, 2, 3};
/* Initialized: written in __init_globals() */
int gvar = 12;
/* Initialized: written in __init_globals() */

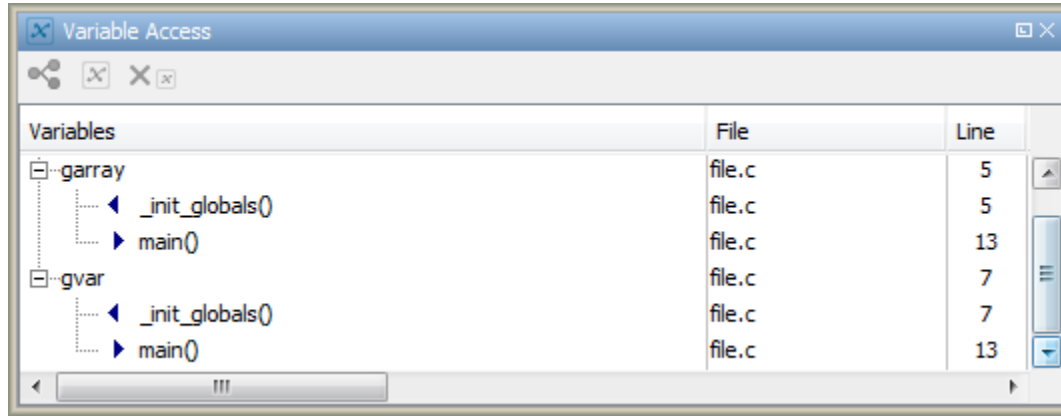
int main(void) {
    int i, lvar = 0;
    for (i = 0; i < 3; i++)
        lvar += func(garray[i] + gvar);
    return lvar;
}
```

After verification:

- On the **Results List** pane, if you select **File** from the  list, under the node `gv_example.c`, you see `_init_globals`.



- On the **Variable Access** pane, `gv_example._init_globals` represents the first write operation on a global variable, for example, `garray`. In the **Values** column, the corresponding value represents the value of the global variable immediately after initialization.



Volatile Variables

The values of volatile variables can change without explicit write operations.

For local volatile variables:

- Polyspace assumes that the variable has a full range of values allowed by its type.
- Unless you explicitly initialize the variable, when you read the variable, Polyspace produces an orange **Non-initialized local variable** check.

In this example, Polyspace assumes that `val1` is potentially noninitialized but `val2` is initialized. Polyspace considers that the + operation can cause an overflow because it assumes both variables to have all possible values allowed by their data types.

```
int func (void)
{
    volatile int val1, val2=0;
    return( val1 + val2);
}
```

For global volatile variables:

- Polyspace assumes that the variable has a full range of values allowed by its type.

  You can constrain the range externally. See *Constrain Global Variable Range* in the documentation for Polyspace Code Prover or Polyspace Code Prover Server.
- Even if you do not explicitly initialize the variable, when you read the variable, Polyspace produces a green **Non-initialized variable** check.

If the root cause of an orange check is a local volatile variable, you cannot override the default assumptions and constrain the values of the volatile variables. Try one of the following:

- If the volatile variable represents hardware-supplied data, see if you can use a function call to model this data retrieval. For example, replace `volatile int port_A` with `int port_A = read_location()`. You do not have to define the function. Polyspace stubs the undefined functions. You can then specify constraints on the function return values using the option `Constraint setup (-data-range-specifications)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.
- See if you can copy the contents of the volatile variable to a global nonvolatile variable. You can then constrain the global variable values throughout your code. See

Constrain Global Variable Range in the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

- Replace the volatile variable with a stubbed function, but only for verification. Before verification, specify constraints on the stubbed functions.

  1. Write a Perl script that replaces each volatile variable declaration with a nonvolatile declaration where you obtain the variable value from a function call.

     For example, if your code contains the line `volatile s8 PORT_A`, your Perl script can contain this substitution:

     ```
     $line=~ s/^\s*volatile\s*s8\s*PORT_A;/s8 PORT_A = random_s8();/g;
     ```

  2. Specify the location of this Perl script for the analysis option `Command/script to apply to preprocessed files (-post-preprocessing-command)`.

  3. In an include file, provide the function declaration. For example, for a function `random_s8`, the include file can contain the following declaration:

     ```
     #ifndef POLYSPACE_H
     #define POLYSPACE_H
     signed char random_s8(void);
     #endif
     ```

  4. Insert a `#include` directive for your include file in the relevant source files

     Instead of a manual insertion, specify the location of your include file for the analysis option `Include (-include)`.

  For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Definitions and Declarations

The definition and declaration of a variable are two different but related operations.

Definition

- If you define a function it means that the body of the function is written: `int f(void) { return 0; }`
- If you define a variable, it means that a part of memory is reserved for the variable: `int x; or extern int x=0;`

When a variable is not defined, the software considers the variable to be initialized, and to have potentially any value in its full range.

When a function is not defined, the software stubs the function.

Declaration

- Function declaration: `int f(void);`
- Variable declaration: `extern int x;`

A declaration provides information about the type of the function or variable. If you use the function or variable in a file where it has not been declared, a compilation error results.

Implicit Data Type Conversions

If an operation involves two operands, the verification assumes that before the operation takes place, the operands can undergo implicit data type conversion. Whether this conversion happens depends on the original data types of the operands.

Following are the conversion rules that apply if the operands in a binary operation have the same data type. Both operands can be converted to `int` or `unsigned int` type before the operation is performed. This conversion is called integer promotion. The conversion rules are based on the ANSI C99 Standard.

- `char` and `signed short` variables are converted to `int` variables.
- `unsigned short` variables are converted to `int` variables only if an `int` variable can represent all possible values of an `unsigned short` variable.

  For targets where the size of `int` is the same as size of `short`, `unsigned short` variables are converted to `unsigned int` variables. For more information on data type sizes, see

  `Target processor type (-target)` For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server..

  .
- Types such as `int`, `long` and `long long` remain unchanged.

Following are some of the conversion rules that apply when the operands have different data types. The rules are based on the ANSI C99 Standard.

- If both operands are `signed` or `unsigned`, the operand with a lower-ranked data type is converted to the data type of the operand with the higher-ranked type. The rank increases in the order `char`, `short`, `int`, `long`, and `long long`.
- If one operand is `unsigned` and the other `signed`, and the `unsigned` operand data type has a rank higher or the same as the `signed` operand data type, the `signed` operand is converted to the `unsigned` operand type.

  For instance, if one operand has data type `int` and the other has type `unsigned int`, the `int` operand is converted to `unsigned int`.

Implicit Conversion When Operands Have Same Data Type

This example shows implicit conversions when the operands in a binary operation have the same data type. If you run verification on the examples, you can use tooltips on the **Source** pane to see the conversions.

In the first addition, `i1` and `i2` are not converted before the addition. Their sum can have values outside the range of an `int` type because `i1` and `i2` have full-range values. Therefore, the `Overflow` check on the first addition is orange.

In the second addition, `c1` and `c2` are promoted to `int` before the addition. The addition does not overflow because an `int` variable can represent all values that result from the sum of two `char` variables. The **Overflow** check on the second addition is green. However, when the sum is assigned to a `char` variable, an overflow occurs during the conversion from `int` back to `char`. An orange **Overflow** check appears on the `=` operation.

```
extern char input_char(void);
extern int input_int(void);

void main(void) {
    char c1 = input_char();
    char c2 = input_char();
    int i1 = input_int();
    int i2 = input_int();

    i1 = i1 + i2;
    c1 = c1 + c2;
}
```

Implicit Conversion When Operands Have Different Data Types

The following examples show implicit conversions that happen when the operands in a binary operation have different data types. If you run verification on the examples, you can use tooltips on the **Source** pane to see the conversions.

In this example, before the `<=` operation, `x` is implicitly converted to `unsigned int`. Therefore, the `User assertion` check is red.

```
#include <assert.h>
int func(void) {
    int x = -2;
```

```
    unsigned int y = 5;
    assert(x <= y);
}
```

In this example, in the first `assert` statement, `x` is implicitly converted to `unsigned int` before the operation `x <= y`. Because of this conversion, in the second `assert` statement, `x` is greater than or equal to zero. The `User assertion` check on the second `assert` statement is green.

```
int input(void);

void func(void) {
    unsigned int y = 7;
    int x = input();
    assert ( x >= -7 && x <= y );
    assert ( x >=0 && x <= 7);
}
```

Using memset and memcpy

In this section...

Polyspace Specifications for memcpy

Syntax:

```
#include <string.h>
void * memcpy ( void * destinationPtr, const void * sourcePtr, size_t num );
```

If your code uses the `memcpy` function, see the information in this table.

Specification	Example

Polyspace runs a **Invalid use of standard library routine** check on the function. The check determines if the memory block that `sourcePtr` or `destinationPtr` points to is greater than or equal in size to the memory assigned to them through `num`.	`#include <string.h>`
	`typedef struct {`
	`  char a;`
	`  int b;`
	`} S;`
	
	`void func(int);`
	
	`void main() {`
	`  S s;`
	`  int d;`
	`  memcpy(&d, &s, sizeof(S));`
	`}`
	
	In this code, Polyspace produces a red **Invalid use of standard library routine** error because:
	• `d` is an `int` variable.
	• `sizeof(S)` is greater than `sizeof(int)`.
	• A memory block of size `sizeof(S)` is assigned to `&d`.

Specification	Example

Polyspace does not check if the memory that `sourcePtr` points to is itself initialized.
 	
Following the use of `memcpy`, Polyspace considers that the variables that `destinationPtr` points to can have any value allowed by their type.
	`#include <string.h>`
	`typedef struct {`
	`  char a;`
	`  int b;`
	`} S;`
	
	`void func(int);`
	
	`void main() {`
	`  S s, d={'a',1};`
	`  int val;`
	`  val = d.b; // val=1`
	
	`  memcpy(&d, &s, sizeof(S));`
	`  val = d.b;`
	`  // val can have any int value`
	`}`
	
	In this code, when the `memcpy` function copies `s` to `d`, Polyspace does not produce a red **Non-initialized local variable** error. Following the copy, the verification considers that the fields of `d` can have any value allowed by their type. For instance, `d.b` can have any value in the range allowed for an `int` variable.

Polyspace raises a red **Invalid use of standard library routine** check if the source and destination arguments overlap. Overlapping assignments are forbidden by the C Standard.	A red check is produced for this memory assignment:
	
	`#include <string.h>`
	
	`int main() {`
	`  char arr[4];`
	`  memcpy (arr, arr + 3, sizeof(int));`
	`}`

Polyspace Specifications for memset

Syntax:

```
#include <string.h>
void * memset ( void * ptr, int value, size_t num );
```

If your code uses the `memset` function, see the information in this table.

Specification	Example

Polyspace runs a **Invalid use of standard library routine** check on the function. The check determines if the memory block that `ptr` points to is greater than or equal in size to the memory assigned to them through `num`.	`#include <string.h>`
	`typedef struct {`
	`  char a;`
	`  int b;`
	`} S;`
	
	`void main() {`
	`  int val;`
	`  memset(&val,0,sizeof(S));`
	`}`
	
	In this code, Polyspace produces a red **Invalid use of standard library routine** error because:
	
	• `val` is an `int` variable.
	• `sizeof(S)` is greater than `sizeof(int)`.
	• A memory block of size `sizeof(S)` is assigned to `&val`.

Specification	Example

If `value` is 0, following the use of `memset`, Polyspace considers that the variables that `ptr` points to have the value 0.	`#include <string.h>`
	`typedef struct {`
	`  char a;`
	`  int b;`
	`} S;`
	
	`void main() {`
	`  S s;`
	`  int val;`
	`  memset(&s,0,sizeof(S));`
	`  val=s.b; //val=0`
	`}`
	
	In this code, Polyspace considers that following the use of `memset`, each field of `s` has value 0.

Specification	Example

Following the use of `memset`, if `value` is anything other than 0, Polyspace considers that:
• The variables that `ptr` points to can be noninitialized.
• If initialized, the variables can have any value that their type allows.
	`#include <string.h>`
	`typedef struct {`
	`  char a;`
	`  int b;`
	`} S;`
	
	`void main() {`
	`  S s;`
	`  int val;`
	`  memset(&s,1,sizeof(S));`
	`  val=s.b;`
	`  // val can have any int value`
	`}`
	
	In this code, Polyspace considers that following the use of `memset`, each field of `s` has any value that its type allows. For instance, `s.b` can have any value in the range allowed for an `int` variable.
	
	Following the `memset`, the structure fields can have different values depending on the structure packing and padding bits. Therefore, structure field assignments with `memset` are implementation-dependent. Code Prover performs this part of the analysis in an implementation-independent way. The analysis allows all possible paddings and therefore full range of values for the structure fields.

#pragma Directives

The verification ignores most `#pragma` directives, because they do not carry information relevant to the verification.

However, the verification takes into account the behavior of these pragmas.

Pragma	Effect on Verification
`#pragma asm` and `#pragma endasm`, or `#asm` and `#endasm`	The verification ignores the content between the pragmas.
`#pragma hdrstop`	For Visual C++® compilers, the verification stops processing precompiled headers at the point where it encounters the pragma.
`#pragma once`	The verification allows the current source file to be included only once in a compilation.
`#pragma pack(n)`, `#pragma pack(push[,n])`, `#pragma pack(pop)`	The verification takes into account the boundary alignment specified in the pragmas. `#pragma pack` without an argument is treated as `#pragma pack(1)`. For more information, see the following example.
`#error` *`message`*	The verification stops if it encounters the directive.

For more information on the pragmas, see your compiler documentation. If the verification does not take into account a certain pragma from the preceding list, see if you specified the right compiler for your verification. For more information, see `Compiler (-compiler)`.

For instance, in this code, the directives `#pragma pack(`*`n`*`)` force a new alignment boundary in the structure. The `User assertion` checks in the `main` function are green because the verification takes into account the behavior of the directives. The verification uses these options:

- `Target processor type (-target)`: `i386` (`char`: 1 byte, `int`: 4 bytes)
- `Compiler (-compiler)`: `gnu4.9`

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

```
#include <assert.h>

#pragma pack(2)

struct _s6 {
    char c;
    int i;
} s6;

#pragma pack() /* Restores default packing: pack(4) */

struct _sb {
    char c;
    int i;
} sb;

#pragma pack(1)

struct _s5 {
    char c;
    int i;
} s5;


int main(void) {
    assert(sizeof(s6) == 6);
    assert(sizeof(sb) == 8);
    assert(sizeof(s5) == 5);
    return 0;
}
```

Standard Library Float Routines

For some two-argument standard library float routines, the verification can ignore the function arguments and assume that the function returns all possible values in its range.

In this code, the first `assert` statement is true and the second `assert` statement is false. However, because the verification assumes that `fmodf` and `nextafterf` return full-range values, it considers that the `assert` statements are false but only on a fraction of possible execution paths. Therefore, the `User assertion` checks on the `assert` statements are orange.

```
#include <math.h>
int main() {
 float val1=10.0, val2=3.0,res;
 res = fmodf(val1/val2);
 assert(res==1.0);

 res = nextafterf(val2,val1);
 assert(res<3.0);
}
```

Unions

In some situations, unions can help you construct efficient code. However, if you write a union member and read back a different union member, the behavior depends on the member sizes and can be implementation-dependent. You have to determine the following for your implementation:

- **Padding** – Padding can be inserted at the end of a union.
- **Alignment** – Members of structures within a union can have different alignments.
- **Endianness** – Whether the most significant byte of a word is stored at the lowest or highest memory address.
- **Bit-order** – Bits within bytes can have both different numbering and allocation to bit fields.

When you use unions in your code, because of these issues, Polyspace verification can lose precision.

If you write a union member and read back another union member, Polyspace considers that the latter member can have any value that its type allows. In this code, the member `b` of `X` is written, but `a` is read. Polyspace considers that `a` can have any `int` value and both branches of the `if-else` statement are reachable.

```
typedef union _u {
    int a;
    char b[4];
} my_union;

void main() {
    my_union X;

    X.b[0] = 1;
    X.b[1] = 1;
    X.b[2] = 1;
    X.b[1] = 1;
    if (X.a == 0x1111) {
    }
    else {
    }
}
```

To avoid using unions in your code, check for violations of `MISRA C:2012 Rule 19.2`.

Note If you initialize a union using a static initializer, following ANSI C standard, Polyspace considers that the union member appearing first in the declaration list gets initialized.

Variable Cast as Void Pointer

The C language allows the use of statements that cast a variable as a void pointer. However, Polyspace verification of these statements entails a loss of precision.

Consider:

```
1   typedef struct {
2   int x1;
3   } s1;
4
5   s1 object;
6
7   void g(void *t) {
8   int x;
9   s1 *p;
10
11  p = (s1 *)t;
12  x = p->x1;   // x should be assigned value 5 but p->x1 is full-range
13  }
14
15  void main(void) {
16  s1 * p;
17
18  object.x1 = 5;
19  p = &object;
20  g((void *)p); // p cast as void pointer
21  }
```

On line 12, the variable `x` must be assigned the value 5. However, the software assumes that `p->x1` has full range of values allowed by its type.

Assembly Code

Polyspace recognizes most inline assemblers as introduction of assembly code. The verification ignores the assembly code but accounts for the fact that the assembly code can modify variables in the C code.

If introduction of assembly code causes compilation errors:

1. Embed the assembly code between a `#pragma` *`my_asm_begin`* and a `#pragma` *`my_asm_end`* statement.
2. Specify the analysis option `-asm-begin` *`my_asm_begin`* `-asm-end` *`my_asm_end`*.

For more information, see `-asm-begin -asm-end`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Recognized Inline Assemblers

Polyspace recognizes these inline assemblers as introduction of assembly code.

- `asm`

  Examples:

  - ```
    int f(void)
    {
        asm ("% reg val; mtmsr val;");
        asm("\tmove.w #$2700,sr");
        asm("\ttrap #7");
        asm(" stw r11,0(r3) ");
        assert (1); // is green
        return 1;
    }
    ```
  - ```
    int other_ignored2(void)
    {
        asm "% reg val; mtmsr val;";
        asm mtmsr val;
        assert (1); // is green
        asm ("px = pm(0,%2); \
            %0 = px1; \
            %1 = px2;"
            : "=d" (data_16), "=d" (data_32)
    ```

```
        : "y" ((UI_32 pm *)ram_address):
"px");
    assert (1); // is green
}
```

- ```
  int other_ignored4(void)
  {
      asm {
          port_in: /* byte = port_in(port); */
          mov EAX, 0
          mov EDX, 4[ESP]
              in AL, DX
              ret
              port_out: /* port_out(byte,port); */
          mov EDX, 8[ESP]
          mov EAX, 4[ESP]
          out DX, AL
          ret }
  assert (1); // is green
  }
  ```

- `__asm__`

  Examples:

  - ```
    int other_ignored6(void)
    {
    #define A_MACRO(bus_controller_mode) \
        __asm__ volatile("nop"); \
        __asm__ volatile("nop"); \
        __asm__ volatile("nop"); \
        __asm__ volatile("nop"); \
        __asm__ volatile("nop"); \
        __asm__ volatile("nop")
            assert (1); // is green
            A_MACRO(x);
            assert (1); // is green
            return 1;
    }
    ```

  - ```
    int other_ignored1(void)
    {
        __asm
            {MOV R8,R8
            MOV R8,R8
            MOV R8,R8
            MOV R8,R8
    ```
    ```

```
            MOV R8,R8}
        assert (1); // is green
    }
```

- ```
  int GNUC_include (void)
  {
      extern int __P (char *__pattern, int __flags,
      int (*__errfunc) (char *, int),
      unsigned *__pglob) __asm__ ("glob64");
      __asm__ ("rorw $8, %w0" \
          : "=r" (__v) \
          : "0" ((guint16) (val)));
      __asm__ ("st g14,%0" : "=m" (*(AP)));
      __asm("" \
          : "=r" (__t.c) \
          : "0" ((((union { int i, j; } *) (AP))++)->i));
      assert (1); // is green
      return (int) 3 __asm__("% reg val");
  }
  ```
- ```
  int other_ignored3(void)
  {
      __asm {ldab 0xffff,0;trapdis;};
  __asm {ldab 0xffff,1;trapdis;};
      assert (1); // is green
      __asm__ ("% reg val");
      __asm__ ("mtmsr val");
      assert (1); // is green
      return 2;
  }
  ```

- #pragma asm #pragma endasm

  Examples:

  - ```
    int pragma_ignored(void)
    {
        #pragma asm
            SRST
        #pragma endasm
            assert (1); // is green
    }
    ```
  - ```
    void test(void)
    {
      #asm
        mov _as:pe, reg
    ```

```
            jre _nop
        #endasm
           int r;
           r=0;
           r++;
     }
```

Single Function Containing Assembly Code

The software stubs a function that is preceded by `asm`, even if a body is defined.

```
asm int h(int tt)              // function h is stubbed even if body is defined
{
  % reg val;                   // ignored
  mtmsr val;                  // ignored
  return 3;                     // ignored
};

void f(void) {
  int x;
  x = h(3);                     // x is full-range
}
```

Multiple Functions Containing Assembly Code

The functions that you specify through the following pragma are stubbed automatically, even if function bodies are defined.

```
#pragma inline_asm(list of functions)
```

Code examples:

```
#pragma inline_asm(ex1, ex2)
   // The functions ex1 and ex2 are
   // stubbed, even if their bodies are defined

int ex1(void)
{
  % reg val;
  mtmsr val;
  return 3;                         // ignored
};

int ex2(void)
{
  % reg val;
```

```
  mtmsr val;
  assert (1);                    // ignored
  return 3;
};


#pragma inline_asm(ex3)  // the definition of ex3 is ignored

int ex3(void)
{
  % reg val;
  mtmsr val;                 // ignored
  return 3;
};

void f(void) {
  int x;

  x = ex1();                   // ex1 is stubbed : x is full-range
  x = ex2();                   // ex2 is stubbed : x is full-range
  x = ex3();                   // ex3 is stubbed : x is full-range
}
```

Local Variables in Functions with Assembly Code

The verification ignores the content of assembly language instructions, but following the instructions, it makes some assumptions about:

- *Uninitialized local variables*: The assembly instructions can initialize these variables.
- *Initialized local variables*: The assembly instructions can write any possible value to the variables allowed by the variable data types.

For instance, the function `f` has assembly code introduced through the `asm` statement.

```
int f(void) {
    int val1, val2 = 0;
    asm("mov 4%0,%%eax"::"m"(val1));
    return (val1 + val2);
}
```

On the `return` statement, the **Non-initialized local variable** check has the following results:

- `val1`: The check is orange because the assembly instruction can initialize `val1`.

- `val2`: The check is green. However, `val2` can have any `int` value.

If the variable is static, the assumptions are true anywhere in the function body, even before the assembly instructions.

Determination of Program Stack Usage

The Polyspace Code Prover analysis can estimate stack usage of each function in your program and compute the entire program stack usage. The analysis uses the function call hierarchy of your program to estimate stack usage. The stack usage of a function is the sum of local variable sizes in the function plus the maximum stack usage from function callees. The stack usage of the function at the top of the call hierarchy is the program stack usage.

For instance, for this call hierarchy, the stack usage of `func` is the size of local variables in `func` plus the maximum stack usage from `func1` and `func2` (unless they are called in mutually exclusive branches of a conditional statement).

For details, see:

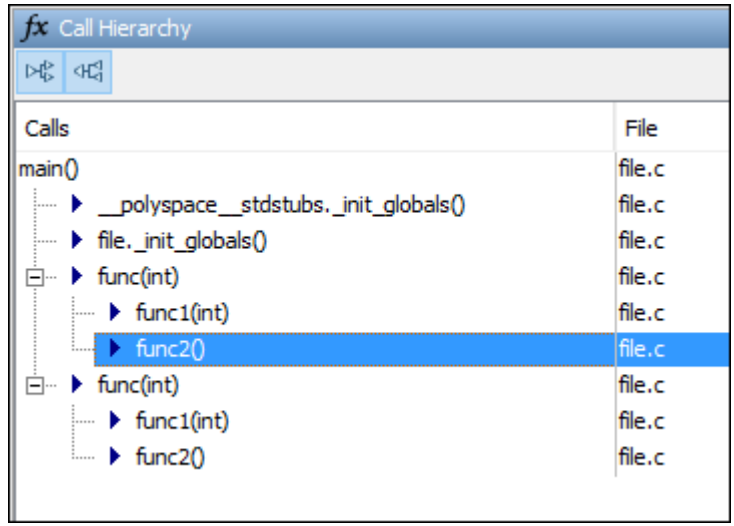
- Function metrics: `Maximum Stack Usage` and `Minimum Stack Usage`
- Project metrics: `Program Maximum Stack Usage` and `Program Minimum Stack Usage`

Investigate Possible Stack Overflow

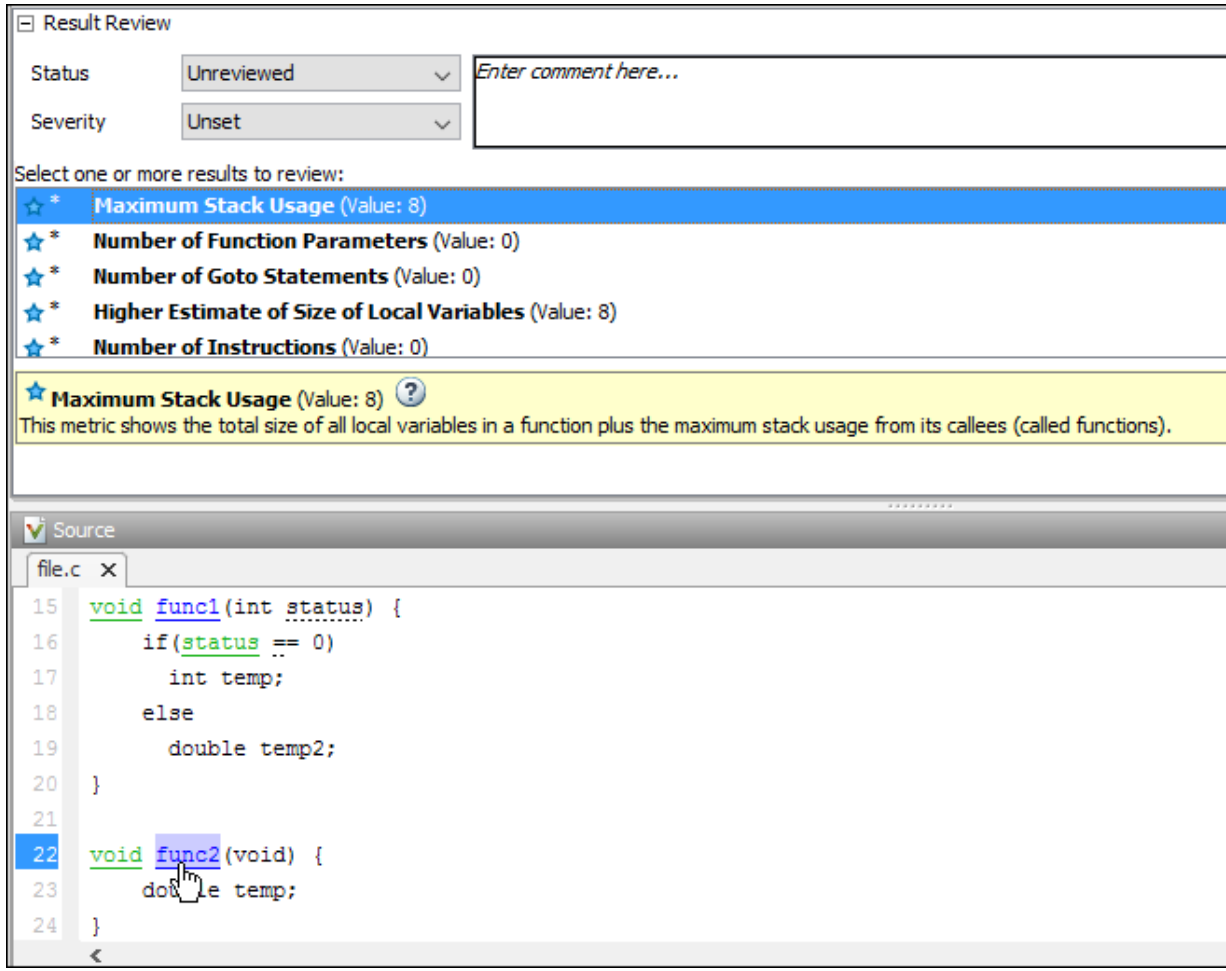
If your stack usage exceeds available stack space, you can identify which function is responsible. Begin at the `main` function and navigate your program call tree. During navigation, look for the function that has an unreasonable size of local variables. If you cannot identify such a function, look for a call sequence that is unreasonably long. The detailed steps for navigation are:

1. On the **Source** pane, select the `main` function. On the **Call Hierarchy** pane, you see the functions called from `main` (callees). To see the full hierarchy, right-click a function and expand all nodes.

If the **Call Hierarchy** pane is not open by default, select **Window > Show/Hide View > Call Hierarchy**.



2 To navigate to the callee definition in your source, on the **Call Hierarchy** pane, double-click each callee name. Then, click the callee name on the **Source** pane. The **Result Details** pane shows the higher estimate of local variable size and stack usage by the callee.



Stack Usage Not Computed

The analysis does not compute stack usage if your code has:

- Red checks. If a definite run-time error occurs in a function or one of its callees, the analysis does not compute its stack usage or the program stack usage. The reason is that code following a red check is not analyzed. If the unanalyzed code contains function calls, any stack usage estimate for the caller function is inaccurate.

In this example, the stack usage of `func` is not computed because following the red overflow, the remainder of the function is not analyzed. If the stack usage was computed, function calls in the unanalyzed code, such as the call to `func2`, would not be part of the computation.

```
#include <limits.h>
void func(void) {
    int val=INT_MAX;
    val++;
    func2();
}
```

- Recursive functions.

If the program stack usage appears as not computed, make sure that the stack usage of all functions are computed. In the **Information** column on the **Results List** pane, check if a function stack usage result shows the value `Not computed`.

Stack Usage Assumptions

If a function is called but not defined in the code that you provide to Polyspace, the stack usage determination does not take the function call into account.

This assumption applies to:

- Implicit C++ constructors.

  For instance, in this example, `func` calls the constructor of class `myClass` when `myObj` is defined. Stack usage determination does not consider the constructor as a callee of `func`.

  ```
  class myClass {std::string str;};

  void func() {
     myClass myObj;
  }
  ```

- Standard library functions or other functions whose definitions are missing from the code in your Polyspace project.

  For instance, in this example, `func` calls the standard library function `cos`. Unless you provide the definition of `cos`, stack usage determination does not consider it as a callee of `func`.

```
#include <math.h>

double func(double arg) {
   return cos(arg);
}
```

Limitations of Polyspace Verification

Code verification has certain limitations. The *Polyspace Code Prover Limitations* document describes known limitations of the code verification process.

This document is stored as `codeprover_limitations.pdf` in the following folder:

`polyspaceroot``\polyspace\verifier\code_prover_desktop`

Here, *`polyspaceroot`* is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2019a`.