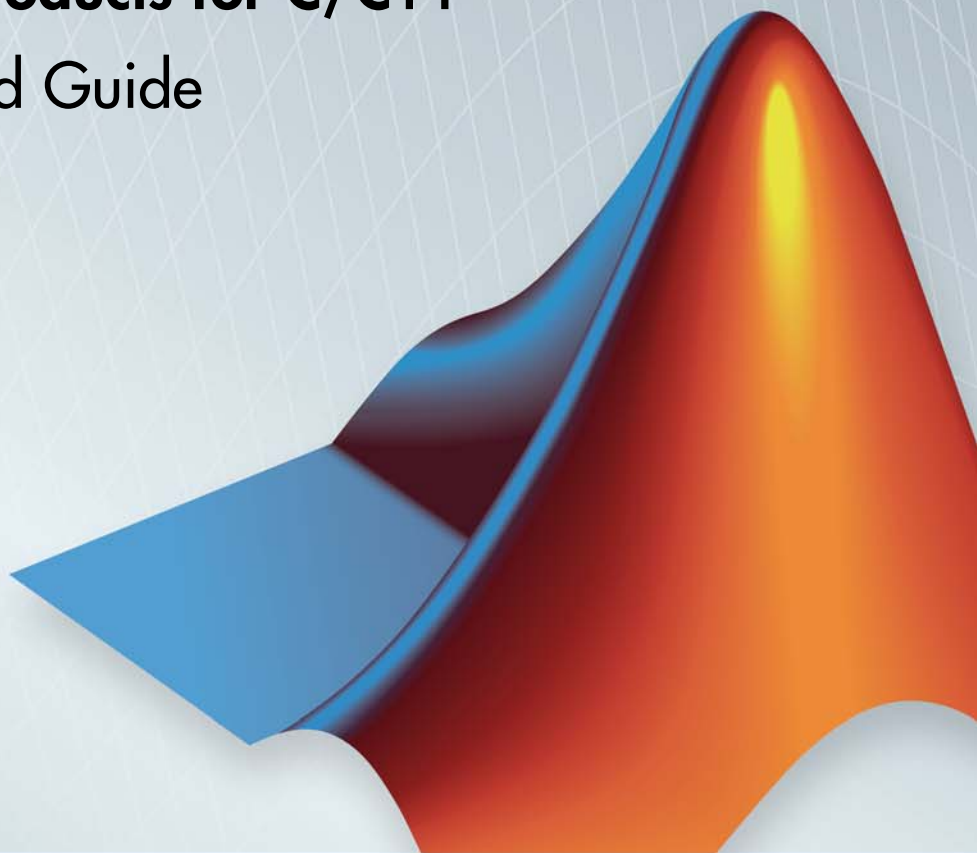


Polyspace® Products for C/C++ Getting Started Guide

R2013a



How to Contact MathWorks



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



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Polyspace® Products for C/C++ Getting Started Guide

© COPYRIGHT 1997–2013 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2008	First printing	Revised for Version 5.1 (Release 2008a)
October 2008	Second printing	Revised for Version 6.0 (Release 2008b)
March 2009	Third printing	Revised for Version 7.0 (Release 2009a)
September 2009	Online only	Revised for Version 7.1 (Release 2009b)
March 2010	Online only	Revised for Version 7.2 (Release 2010a)
September 2010	Fourth Printing	Revised for Version 8.0 (Release 2010b)
April 2011	Fifth Printing	Revised for Version 8.1 (Release 2011a)
September 2011	Online only	Revised for Version 8.2 (Release 2011b)
March 2012	Online only	Revised for Version 8.3 (Release 2012a)
September 2012	Online only	Revised for Version 8.4 (Release 2012b)
March 2013	Online only	Revised for Version 8.5 (Release 2013a)

Introduction to Polyspace Products for Verifying C/C++ Code

1

Product Description	1-2
Polyspace Client for C/C++	1-2
Polyspace Server for C/C++	1-2
 Polyspace Verification	 1-4
Overview of Polyspace Verification	1-4
The Value of Polyspace Verification	1-4
 Product Components	 1-7
Polyspace Verification Environment	1-7
Other Polyspace Components	1-10
 Install Polyspace Products	 1-12
Find the Installation Instructions	1-12
Obtain Licenses for Polyspace Software	1-12
 Polyspace Software Workflow and Tutorials	 1-13
Basic Workflow	1-13
Tutorials	1-14
 Additional Information and Support	 1-16
Product Help	1-16
MathWorks Online	1-16
 Related Products	 1-17
Polyspace Products for Verifying Ada Code	1-17
Polyspace Products for Linking to Models	1-17

Set Up a Polyspace Project

2

Set Up Polyspace Project	2-2
Tutorial Overview	2-2
What Is a Project?	2-2
Prepare Project Folders	2-3
Open Polyspace Verification Environment	2-4
Create a New Project to Verify the Example C File	2-6

Run a Verification

3

Run Verification	3-2
Tutorial Overview	3-2
Before You Start the Tutorial	3-3
Prepare for Verification	3-3
Start Server Verification from Project Manager	3-9
Start Client Verification from Project Manager	3-20

Review Verification Results

4

Review Verification Results	4-2
Tutorial Overview	4-2
Before You Start	4-2
Open Verification Results	4-3
Explore Results Manager perspective	4-3
Review Results	4-7
Review Results Systematically	4-23
Automatically Test Unproven Code	4-28
Generate Reports of Verification Results	4-28

Check Compliance with Coding Rules

5

Check Compliance with Coding Rules	5-2
Tutorial Overview	5-2
Before You Start	5-3
Create New Module for Coding Rules Checking	5-3
Set MISRA C Checking Option	5-10
Select Coding Rules to Check	5-11
Exclude Files from MISRA C Checking	5-14
Run a Verification with Coding Rules Checking	5-15
Examine MISRA C Violations	5-16
Open MISRA-C Report	5-19

Index

Introduction to Polyspace Products for Verifying C/C++ Code

- “Product Description” on page 1-2
- “Polyspace Verification” on page 1-4
- “Product Components” on page 1-7
- “Install Polyspace Products” on page 1-12
- “Polyspace Software Workflow and Tutorials” on page 1-13
- “Additional Information and Support” on page 1-16
- “Related Products” on page 1-17

Product Description

In this section...
“Polyspace Client for C/C++” on page 1-2
“Polyspace Server for C/C++” on page 1-2

Polyspace Client for C/C++

Prove the absence of run-time errors in source code

Polyspace® Client™ for C/C++ provides code verification that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in source code using static code analysis that does not require program execution, code instrumentation, or test cases. Polyspace Client for C/C++ uses formal methods-based abstract interpretation techniques to verify code. You can use it on handwritten code, generated code, or a combination of the two, before compilation and test.

Support for industry standards is available through IEC Certification Kit (for ISO 26262 and IEC 61508) and DO Qualification Kit (for DO-178).

Key Features

- File- and class-level software component verification
- Formal method based abstract interpretation
- Display of run-time errors directly in code
- MISRA-C:2004, MISRA-C++:2008, and JSF++ coding standard enforcement, with direct source file links
- Cyclomatic complexity and other code metrics
- Eclipse™ and Microsoft® Visual Studio® IDE integration

Polyspace Server for C/C++

Perform code verification on computer clusters and publish metrics

Polyspace Server™ for C/C++ provides code verification that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain

other run-time errors in source code. For faster performance, Polyspace Server for C/C++ lets you schedule verification tasks to run on a computer cluster. Jobs are submitted to the server using Polyspace Client for C/C++. You can integrate jobs into automated build processes and set up e-mail notifications. You can view defects, regressions, and code metrics via a Web browser. You then use the client to download and visualize verification results.

Support for industry standards is available through IEC Certification Kit (for ISO 26262 and IEC 61508) and DO Qualification Kit (for DO-178).

Key Features

- Web-based dashboard providing code metrics and quality status
- Automated job scheduling and e-mail notification
- Multi-server job queue manager
- Accelerated performance on multicore servers
- Verification report generation
- Mixed operating system environment support

Polyspace Verification

In this section...
“Overview of Polyspace Verification” on page 1-4
“The Value of Polyspace Verification” on page 1-4

Overview of Polyspace Verification

Polyspace products verify C, C++, and Ada code by detecting run-time errors before code is compiled and executed.

To verify the source code, you set up verification parameters in a project, run the verification, and review the results. A graphical user interface helps you to efficiently review verification results. Results are color-coded:

- **Green** – Indicates code that never has an error.
- **Red** – Indicates code that always has an error.
- **Gray** – Indicates unreachable code.
- **Orange** – Indicates unproven code (code that might have an error).

The color-coding helps you to quickly identify errors and find the exact location of an error in the source code. After you fix errors, you can easily run the verification again.

The Value of Polyspace Verification

Polyspace verification can help you to:

- “Enhance Software Reliability” on page 1-4
- “Decrease Development Time” on page 1-5
- “Improve the Development Process” on page 1-6

Enhance Software Reliability

Polyspace software ensures the reliability of your C and C++ applications by proving code correctness and identifying run-time errors. Using advanced

verification techniques, Polyspace software performs an exhaustive verification of your source code.

Because Polyspace software verifies all possible executions of your code, it can identify code that:

- Never has an error
- Always has an error
- Is unreachable
- Might have an error

With this information, you know how much of your code is free of run-time errors, and you can improve the reliability of your code by fixing errors.

You can also improve the quality of your code by using Polyspace verification software to check that your code complies with established coding standards, such as the MISRA C®, MISRA® C++ or JSF++ standards.¹

Decrease Development Time

Polyspace software reduces development time by automating the verification process and helping you to efficiently review verification results. You can use it at any point in the development process. However, using it during early coding phases allows you to find errors when it is less costly to fix them.

You use Polyspace software to verify source code before compile time. To verify the source code, you set up verification parameters in a project, run the verification, and review the results. This process takes significantly less time than using manual methods or using tools that require you to modify code or run test cases.

Color coding of results helps you to quickly identify errors. You spend less time debugging because you can see the exact location of an error in the source code. After you fix errors, you can easily run the verification again.

1. MISRA and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

Using Polyspace verification software helps you to use your time effectively. Because you know the parts of your code that do not have errors, you can focus on the code with proven or potential errors.

Reviewing code that might have errors (orange code) can be time consuming, but Polyspace software helps you with the review process. You can use filters to focus on certain types of errors or you can allow the software to identify the code that you must review.

Improve the Development Process

Polyspace software makes it easy to share verification parameters and results, allowing the development team to work together to improve product reliability. Once verification parameters have been set up, developers can reuse them for other files in the same application.

Polyspace verification software supports code verification throughout the development process:

- An individual developer can find and fix run-time errors during the initial coding phase.
- Quality assurance engineers can check overall reliability of an application.
- Managers can monitor application reliability by generating reports from the verification results.

Product Components

In this section...
“Polyspace Verification Environment” on page 1-7
“Other Polyspace Components” on page 1-10

Polyspace Verification Environment

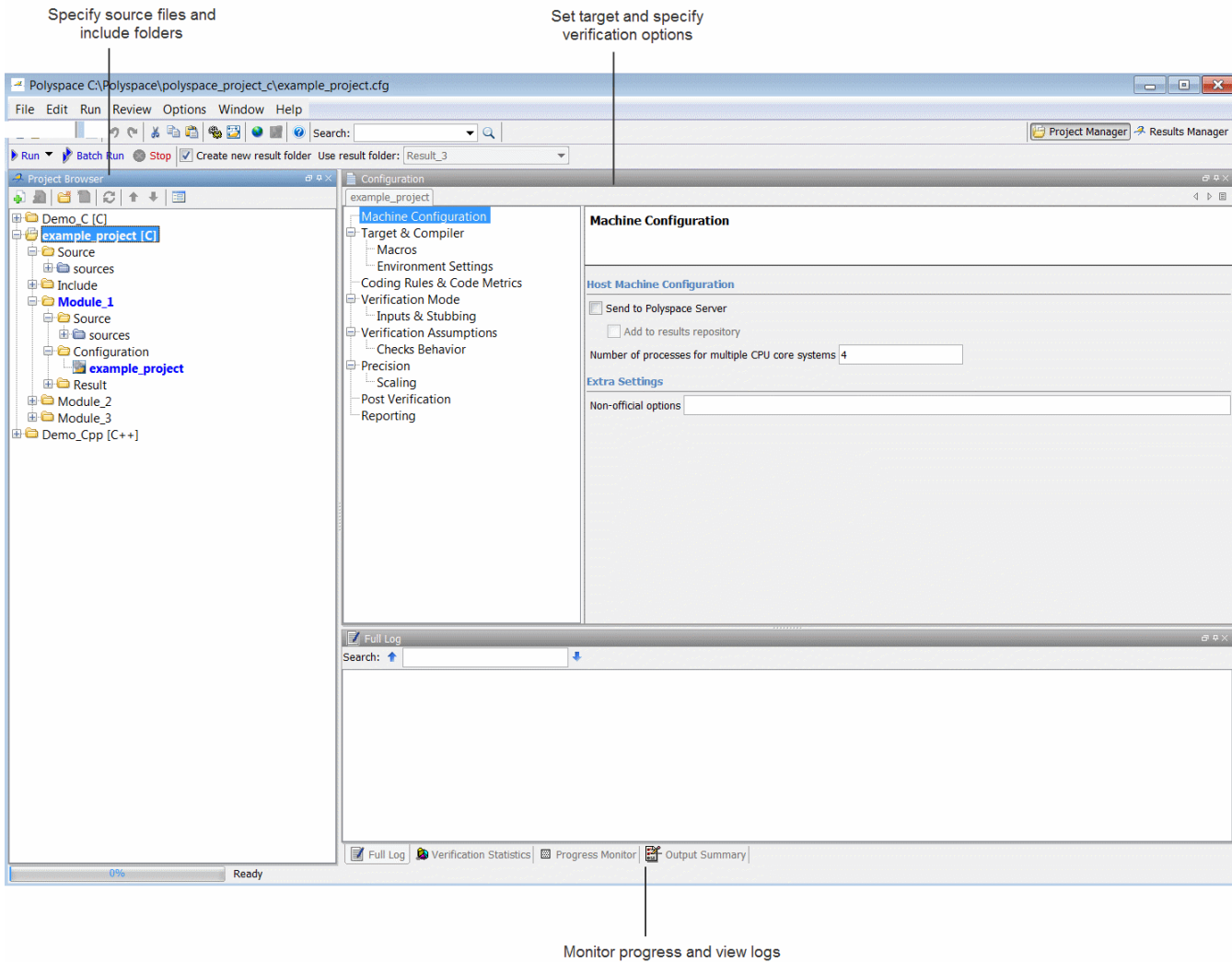
The Polyspace verification environment (PVE) is the graphical user interface of the Polyspace Client for C/C++ software. You use the Polyspace verification environment to create Polyspace projects, start verifications, and review verification results.

The Polyspace verification environment consists of two perspectives:

- “Project Manager Perspective” on page 1-7
- “Results Manager Perspective” on page 1-9

Project Manager Perspective

The Project Manager perspective allows you to create projects, set verification parameters, and start verifications.



You use the Project Manager perspective in the tutorial “Set Up Polyspace Project” on page 2-2.

Results Manager Perspective

The Results Manager perspective allows you to review:

- Results from the Polyspace coding rules checker to verify compliance with established coding standards.
- Verification results, comment individual checks, and track review progress.

The screenshot displays the Polyspace Results Manager perspective with the following components:

- Results Explorer:** Shows a list of procedural entities and checks. A table of checks is visible:

Check	Count	Progress
Red IDP justified / to justify	0/1	0
Red justified / to justify	0/5	0
Gray justified / to justify	0/6	0
Orange justified / to justify	0/18	0
Software reliability indicator	259/294	88
- Check Details:** Displays the details for a selected check (IDP.8 Error: pointer is outside its bounds). The error message states: "dereference of local variable 'p' (pointer to int 32, size: 32 bits): pointer is not null points to 4 bytes at offset 400 in buffer of 400 bytes, so is outside bounds may point to variable or field of variable in: (Pointer_Arithmetic:array)".
- Source:** Shows the source code snippet where the error occurred:


```

104 p = 5; /* Out of bounds */
105
106 else
107 {
108     i++;
109 }
110 }
111
```
- Call Hierarchy:** Shows the call stack for the current function, including 'example.Pointer_Arithmetic', 'pst_stubs_0_get_bus_status', and 'example_get_oil_pressure'.
- Variable Access:** Shows the variable access for the current function, including 'N-SHR' and 'Demo_C'.
- Results Summary:** Shows the overall progress of the review, currently at 0%.

Run-time checks and coding rule violations

Source code

Variable access

Call hierarchy

You use the Results Manager perspective in the tutorials in “Check Compliance with Coding Rules” on page 5-2 and “Review Verification Results” on page 4-2.

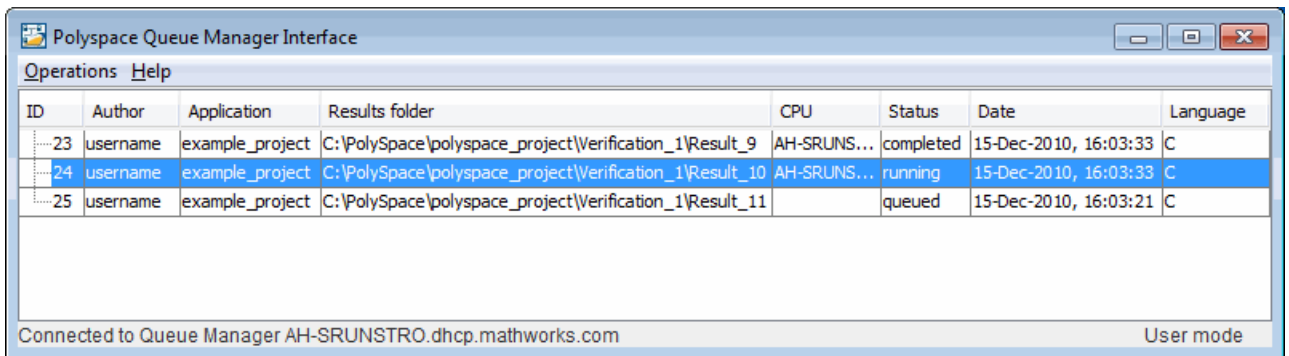
Other Polyspace Components

In addition to the Polyspace verification environment, Polyspace products provide several other components to manage verifications, improve productivity, and track software quality. These components include:

- Polyspace Queue Manager Interface (Spooler)
- Polyspace Metrics Web Interface

Polyspace Queue Manager Interface (Polyspace Spooler)

The Polyspace Queue Manager (also called the Polyspace Spooler) is the graphical user interface of the Polyspace Server for C/C++ software. You use the Polyspace Queue Manager Interface to move jobs within the queue, remove jobs, monitor the progress of individual verifications, and download results.



You use the Polyspace Queue Manager in the tutorial “Start Server Verification from Project Manager” on page 3-9.

Polyspace Metrics Web Interface

Polyspace Metrics is a web-based tool for software development managers, quality assurance engineers, and software developers. Polyspace Metrics

allows you to evaluate software quality metrics, and monitor changes in code metrics, coding rule violations, and run-time checks through the lifecycle of a project.

For information on using Polyspace Metrics, see “Quality Metrics”.

Install Polyspace Products

In this section...
“Find the Installation Instructions” on page 1-12
“Obtain Licenses for Polyspace Software” on page 1-12

Find the Installation Instructions

The tutorials require Polyspace Client for C/C++ and Polyspace Server for C/C++. Instructions for installing Polyspace products are in “Software Installation”. Before installing Polyspace products, you must obtain the required licenses.

Obtain Licenses for Polyspace Software

For information about obtaining licenses for Polyspace products, see “License Administration”.

Polyspace Software Workflow and Tutorials

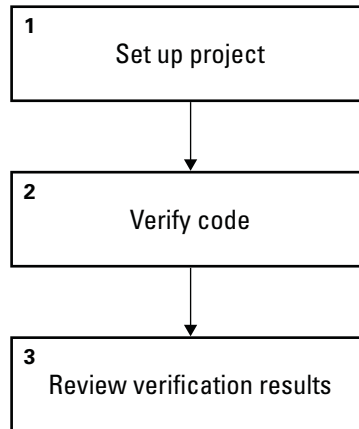
In this section...

“Basic Workflow” on page 1-13

“Tutorials” on page 1-14

Basic Workflow

The following graphic shows the basic workflow for using Polyspace software to verify source code.



In this workflow, you:

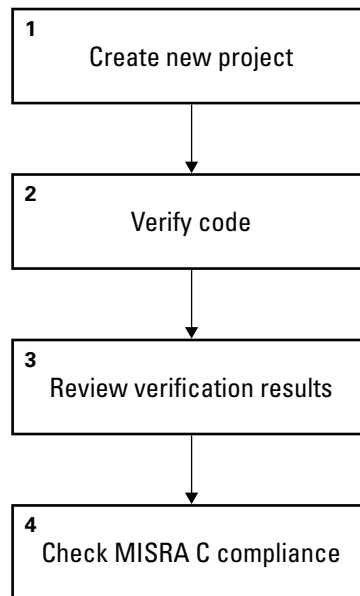
- 1** Use the Project Manager perspective to set up a project file.
- 2** Verify code on a server or client.

You can use the Project Manager perspective to start the verification or you can select files from a Microsoft Windows® folder and send them to Polyspace software for verification. For verifications that run on a server, you use the Polyspace Queue Manager Interface (Polyspace Spooler) to manage the verification and download the results to a client. You can set an option to check coding rules compliance in the first stage of the verification.

3 Use the Results Manager perspective to review verification results.

Tutorials

The tutorials guide you through the basic workflow, including the different options for running verifications. The following graphic shows the workflow that you follow in these tutorials.



In this workflow, you:

1 Create a new project that you use for the workflow.

See “Set Up Polyspace Project” on page 2-2.

2 Verify a single C file.

See “Run Verification” on page 3-2. In this tutorial, you verify the same file using three different methods of running a verification:

- Start a verification that runs on a server using the Project Manager perspective.

- Start a verification that runs on a client using the Project Manager perspective.

3 Review the verification results.

See “Review Verification Results” on page 4-2.

4 Modify the project to include MISRA C checking and review the MISRA C violations in the example file.

See “Check Compliance with Coding Rules” on page 5-2.

Additional Information and Support

In this section...
“Product Help” on page 1-16
“MathWorks Online” on page 1-16

Product Help

To access Polyspace online Help, select **Help > Help** .

To access the online documentation for Polyspace products, go to:

www.mathworks.com/help/polyspace_c

MathWorks Online

For additional information and support, go to:

www.mathworks.com/products/polyspace

Related Products

In this section...
“Polyspace Products for Verifying Ada Code” on page 1-17
“Polyspace Products for Linking to Models” on page 1-17

Polyspace Products for Verifying Ada Code

For information about Polyspace products that verify Ada code, go to:

<http://www.mathworks.com/products/polyspaceclientada/>

<http://www.mathworks.com/products/polyspaceserverada/>

Polyspace Products for Linking to Models

For information about Polyspace products that link to models, go to:

<http://www.mathworks.com/products/polyspacemodels1/>

<http://www.mathworks.com/products/polyspaceumlrh/>

Set Up a Polyspace Project

Set Up Polyspace Project

In this section...
“Tutorial Overview” on page 2-2
“What Is a Project?” on page 2-2
“Prepare Project Folders” on page 2-3
“Open Polyspace Verification Environment” on page 2-4
“Create a New Project to Verify the Example C File” on page 2-6

Tutorial Overview

Before you can run a verification of your source code, you must have a project. In this tutorial, you create the project that you use to run verifications in later tutorials.

What Is a Project?

In Polyspace software, a project is a named set of parameters for verification of your software project source files. A project includes:

- Source files
- Include folders
- One or more Configurations, specifying a set of analysis options
- One or more Modules, each of which include:
 - Source (specific versions of source files used in the verification)
 - Configuration (specific set of analysis options used for the verification)
 - Verification results

You can create your own project or use an existing project. You create and modify a project using the Project Manager perspective.

In this tutorial, you create a new project and save it as a configuration file (.cfg).

Prepare Project Folders

Before you start verifying a C file with Polyspace software, you must know the locations of the C source file and the include files. You must also know where you want to store the verification results.

For each project, you decide where to store source files and results. For example, you can create a project folder, and then in that folder, create separate folders for the source files, include files, and results.

For this tutorial, prepare a project folder as follows:

- 1** Create a project folder named `polyspace_project`.
- 2** Open `polyspace_project`, and create the following folders:
 - `sources`
 - `includes`

- 3** Copy the file `example.c` and `single_file_analysis.c` from

```
Polyspace_Install\polyspace\examples\cxx\Demo_C_Single-File\sources
```

to

```
polyspace_project\sources
```

Polyspace_Install is the installation folder.

- 4** Copy the files `include.h`, `math.h`, `single_file_analysis.h` and `single_file_private.h` from

```
Polyspace_Install\polyspace\examples\cxx\Demo_C_Single-File\sources
```

to

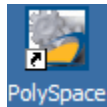
```
polyspace_project\includes.
```

Open Polyspace Verification Environment

You use the Polyspace verification environment to create projects, start verifications, and review verification results.

To open the Polyspace verification environment:

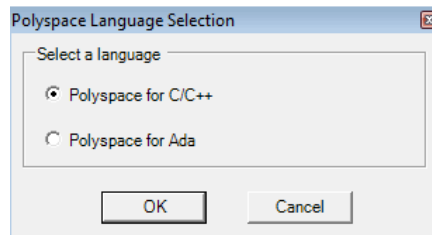
- 1 Double-click the **Polyspace** icon (Windows systems).



Note On a Linux® or UNIX® system, use the following command:

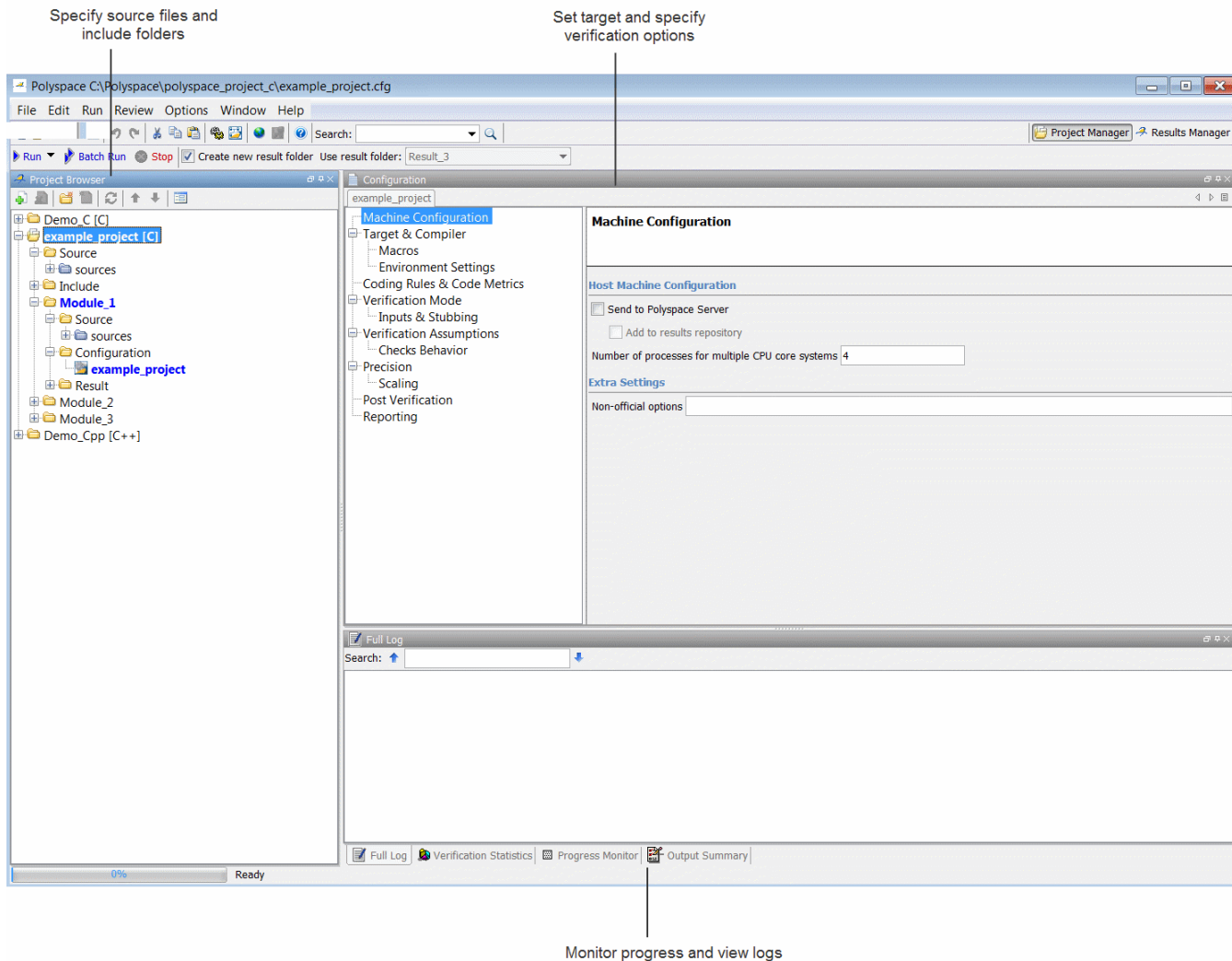
```
/usr/local/Polyspace/PVE/bin/polyspace
```

- 2 If you have only Polyspace Client for C/C++ software installed on your computer, skip this step. If you have both Polyspace Client for C/C++ and Polyspace Client for Ada products on your system, the Polyspace Language Selection dialog box opens.



- 3 Select **Polyspace for C/C++** and click **OK**.

The Polyspace verification environment opens.



By default, the Polyspace verification environment displays the Project Manager perspective. The Project Manager perspective has three main panes.

Use this section...	For...
Project Browser (upper-left)	Specifying: <ul style="list-style-type: none">• Source files• Include folders• Results folder
Configuration (upper-right)	Specifying verification options
Output (lower-right)	Monitoring the progress of a verification, and viewing status, log messages, and general verification statistics.

You can resize or hide any of these panes. You learn more about the Project Manager perspective later in this tutorial.

Create a New Project to Verify the Example C File

You must have a project, saved with file type `cfg`, to run a verification. In this part of the tutorial, you create a new project for verifying `example.c` and `single_file_analysis.c`.

To create a new project, you:

- “Open a New Project” on page 2-6
- “Specify Source Files and Include Folders” on page 2-9
- “Specify Target Environment” on page 2-11
- “Specify Analysis Options” on page 2-12
- “Save the Project” on page 2-12

Open a New Project

To open a new project for verifying `example.c` and `single_file_analysis.c`:

- 1 Select **File > New Project**. The Polyspace Project – Properties dialog box opens.

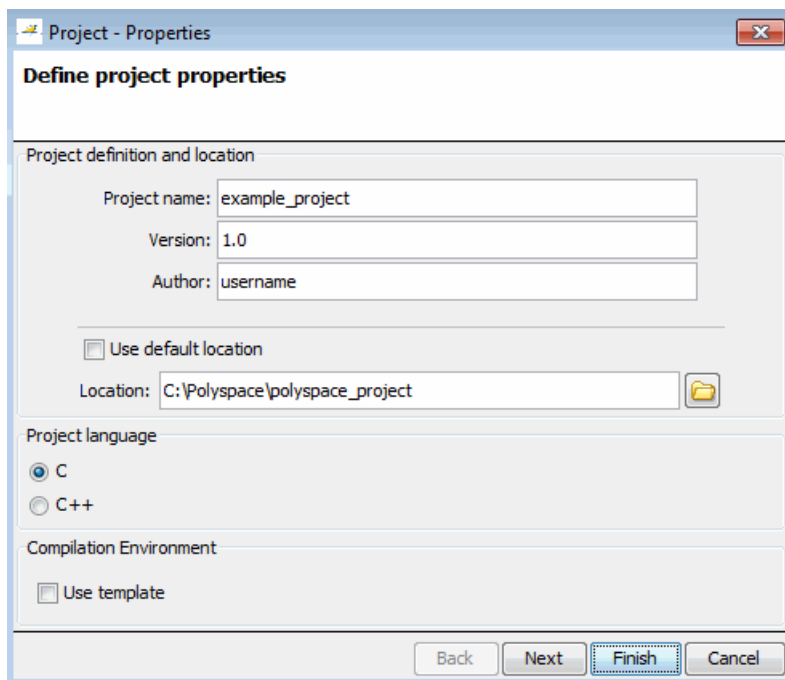
- 2** In the **Project name** field, enter `example_project`.
- 3** Clear the **Default location** check box. In this tutorial, you change the location to the project folder that you created in “Prepare Project Folders” on page 2-3.

Note You can update the default project location. Select **Options > Preferences**, which opens the Polyspace Preferences dialog box. On the **Project and result folder** tab, in the **Define default project location** field, specify the new default location.

- 4** In the **Location** field, enter or navigate to the project folder that you created earlier.

In this tutorial, the project folder is `C:\Polyspace\polyspace_project`.

- 5** In the Project language section, select **C**.




6 Click **Finish**.

The `example_project` opens in the Polyspace verification environment.

Specify Source Files and Include Folders

To specify the source files and include folders for the verification of `example.c` and `single_file_analysis.c`:

1 In the Project Browser, select the Source folder.

2 On the Project Browser toolbar, click the **Add source** icon . The Polyspace Project – Add Source Files and Include Folders dialog box opens.

3 The project folder `polyspace_project` must appear in the field **Look in**. If it does not, navigate to that folder.

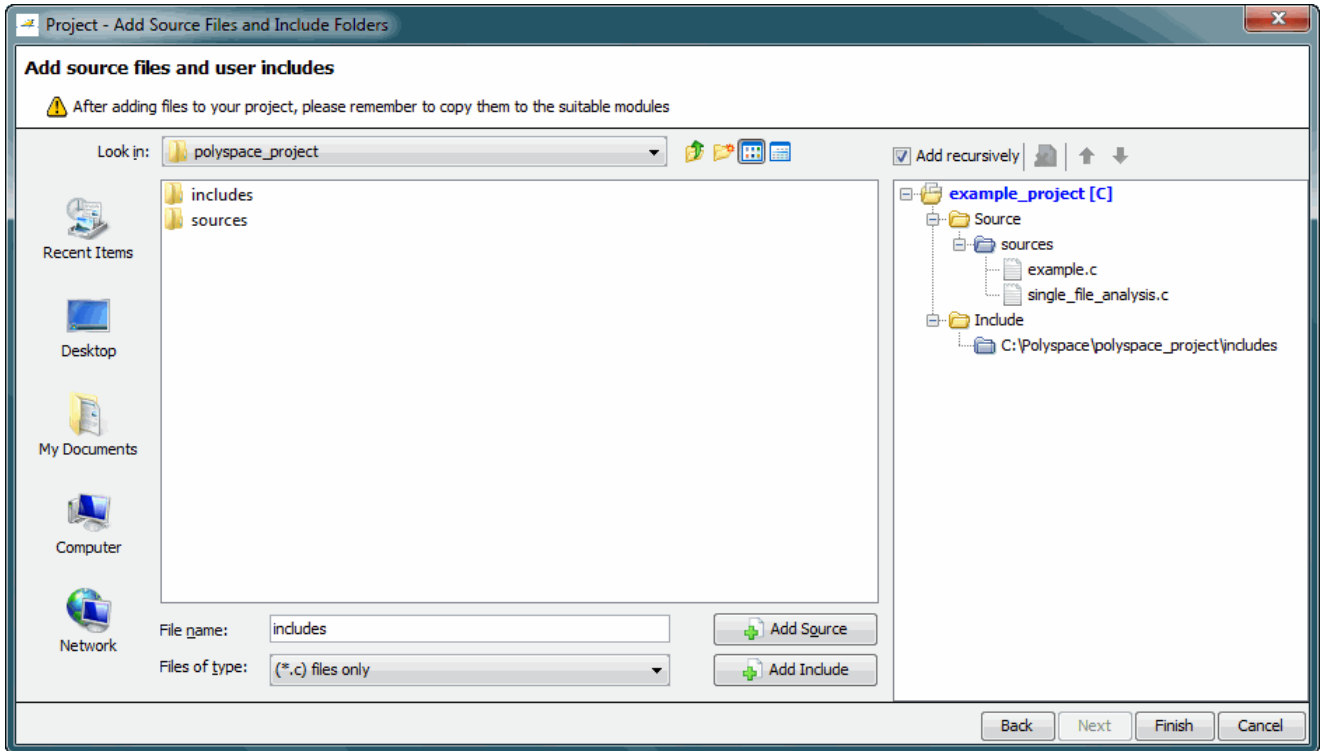
4 Select the sources folder. Then click **Add Source**.

The `example.c` file and the `single_file_analysis.c` file appear in the Source tree for `example_project`.

5 Select the `includes` folder. Then click **Add Include**.

The `includes` folder appears in the Include tree for `example_project`.

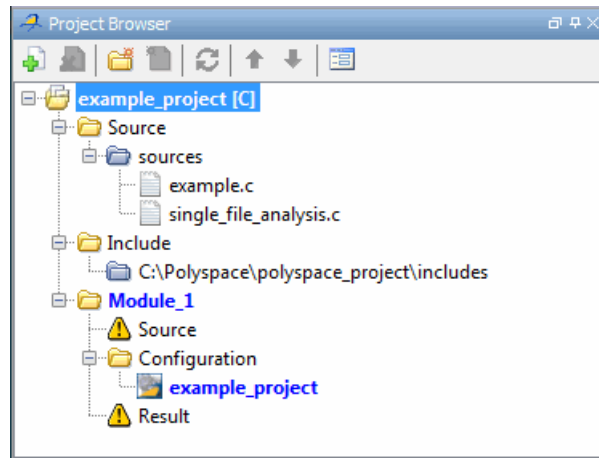
2 Set Up a Polyspace® Project



Note In addition to the include folders that you specify, Polyspace software automatically adds the standard includes to your project.

6 Click **Finish** to apply the changes and close the dialog box.

The Project Browser now looks like the following graphic.



Specify Target Environment

Many applications are designed to run on specific target CPUs and operating systems. Since some run-time errors are dependent on the target, before running you must specify the type of CPU and operating system used in the target environment.

In the Project Manager perspective, the **Configuration > Target & Compiler** pane allows you to specify the target operating system and processor type for your application.

To specify the target environment for this tutorial:

- 1** In the **Configuration** pane, select **Target & Compiler**.
- 2** Set the **Target operating system** to `no_predefined_OS`.
- 3** Set the **Target processor type** to `i386`.

For more information about emulating your target environment, see “Set Up a Target”.

Specify Analysis Options

In the Project Manager perspective, the **Configuration** pane allows you to set analysis options that Polyspace software uses during the verification process. For this tutorial, use the default values for all options.

For more information, see “Analysis Options for C Code” or “Analysis Options for C++ Code”.

Save the Project

To save the project, select **File > Save**.

The software saves your project using the **Project name** and **Location** that you specified when creating the project.

Run a Verification

Run Verification

In this section...
“Tutorial Overview” on page 3-2
“Before You Start the Tutorial” on page 3-3
“Prepare for Verification” on page 3-3
“Start Server Verification from Project Manager” on page 3-9
“Start Client Verification from Project Manager” on page 3-20

Tutorial Overview

Once you have created the project `example_project.cfg`, as described in “Set Up Polyspace Project” on page 2-2, you can run the verification.

You can run a verification on a server or a client.

Use...	For...
Server	<ul style="list-style-type: none"> • Best performance • Large files (more than 800 lines of code, including comments)
Client	<ul style="list-style-type: none"> • When the server is busy • Small files <hr/> <p>Note Verification on a client takes more time. When a verification is running, you might not be able to use your client computer.</p> <hr/>

In this tutorial, you learn how to start a server and client verification using the Project Manager. You also verify the file `example.c` and `single_file_analysis.c`.

The server and client verifications store the same results in your project. You review these results in the tutorial “Review Verification Results” on page 4-2.

Before You Start the Tutorial

Before you start this tutorial, you must complete “Set Up Polyspace Project” on page 2-2. For this tutorial, you use the folders and project file, `example_project.cfg`, from that tutorial.

Prepare for Verification

Open the Project

To run a verification, you must have an open project file. For this tutorial, you use the project file `example_project.cfg` that you created in “Set Up Polyspace Project” on page 2-2. If `example_project.cfg` is not already running, start it.

To open `example_project.cfg`:

- 1** If the Polyspace software is not already open, open it.
- 2** Select **File > Open Project**.
- 3** In the Polyspace Project dialog box, from the **Look in** drop-down list, navigate to `polyspace_project`.
- 4** Select `example_project.cfg`.
- 5** Click **Open** to open the file and close the dialog box.

Specify Source Files to Verify

Each Polyspace project can contain multiple modules. With each module, you can verify a specific set of source files using a specific set of analysis options.

Before you start a verification, you must specify which files you want to verify by copying them into a verification module. In the `example_project[C]` in this tutorial, there are two files to verify. You verify both files separately, so they must be copied into individual modules.

To copy source files to a module:

1 In the Project Browser Source tree, right-click `example.c`.

2 Select **Copy Source File to > Module_1**.

The `example.c` file appears in the Source tree of `Module_1`.

3 In the Project Browser Source tree, right-click `example_project[C]`.

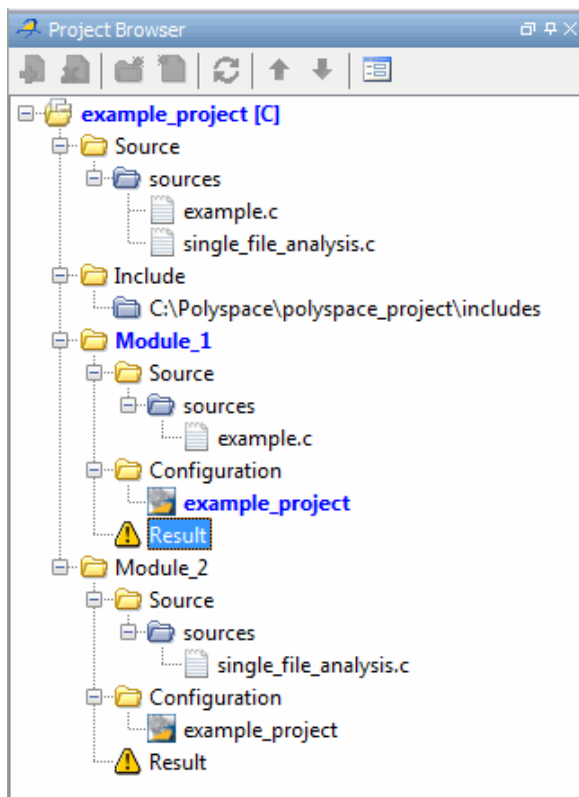
4 Select **Create New Module**.

A new `Module_2` appears in your Project Browser tree.

5 In the Project Browser Source tree, right-click `single_file_analysis.c`.

6 Select **Copy Source File to > Module_2**.

The `single_file_analysis.c` file is now displayed in the Source tree of `Module_2`.



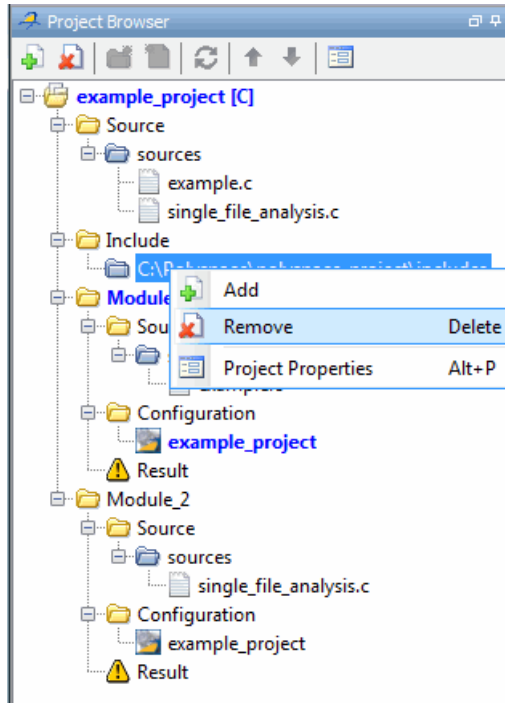
Check for Compilation Problems

During a verification, if the Compilation Assistant detects compilation errors, the verification stops. The software displays errors and possible solutions on the **Output Summary** tab.

Note The Compilation Assistant does not support the verification option `-unit-by-unit`. For more information, see, “Check for Compilation Problems”

To check your project for compilation problems:

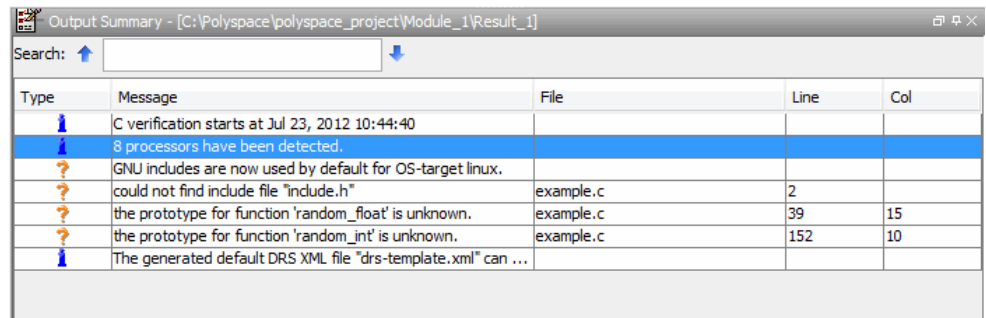
- 1 In the **Configuration > Machine Configuration** pane, make sure the **Send to Polyspace Server** check box is not selected.
- 2 In the **Project Browser** tree, right-click the **Include** folder (C:\Polyspace\polyspace_project\includes), and then select **Remove**. Missing include files will cause compilation problems.



- 3 Select `Module_1` in the Project Browser.

- 4 On the Project Manager toolbar, click .


The software compiles your code and checks for errors. It reports the results on the **Output Summary** tab.



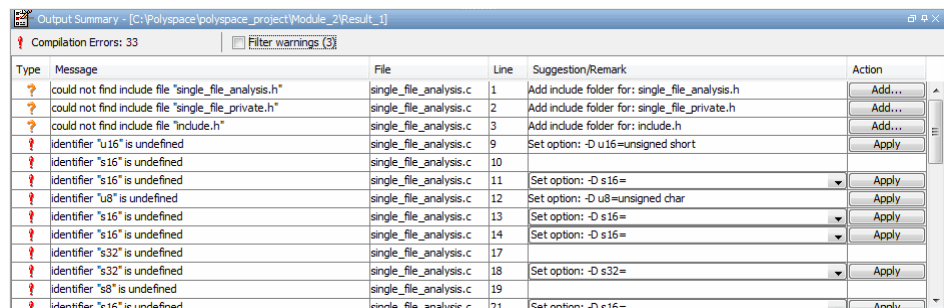
Type	Message	File	Line	Col
Info	C verification starts at Jul 23, 2012 10:44:40			
Info	8 processors have been detected.			
Warning	GNU includes are now used by default for OS-target linux.			
Warning	could not find include file "include.h"	example.c	2	
Warning	the prototype for function 'random_float' is unknown.	example.c	39	15
Warning	the prototype for function 'random_int' is unknown.	example.c	152	10
Info	The generated default DRS XML file "drs-template.xml" can ...			

In this case, the software generates only warnings (in orange), not errors, for the missing include files. Because there are no errors, the verification is still completed. It is a good practice to resolve warnings for missing include files first, because they might be the root cause of other warnings or errors.

5 Select `Module_2` in the Project Browser tree.

6 Click the  button again. The verification software runs with `single_file_analysis.c` as the source file.

Again, the software compiles your code and checks for errors. It reports the results on the **Output Summary** tab.



Type	Message	File	Line	Suggestion/Remark	Action
Warning	could not find include file "single_file_analysis.h"	single_file_analysis.c	1	Add include folder for: single_file_analysis.h	Add...
Warning	could not find include file "single_file_private.h"	single_file_analysis.c	2	Add include folder for: single_file_private.h	Add...
Warning	could not find include file "include.h"	single_file_analysis.c	3	Add include folder for: include.h	Add...
Error	identifier "u16" is undefined	single_file_analysis.c	9	Set option: -D u16=unsigned short	Apply
Error	identifier "s16" is undefined	single_file_analysis.c	10		
Error	identifier "s16" is undefined	single_file_analysis.c	11	Set option: -D s16=	Apply
Error	identifier "u8" is undefined	single_file_analysis.c	12	Set option: -D u8=unsigned char	Apply
Error	identifier "s16" is undefined	single_file_analysis.c	13	Set option: -D s16=	Apply
Error	identifier "s16" is undefined	single_file_analysis.c	14	Set option: -D s16=	Apply
Error	identifier "s32" is undefined	single_file_analysis.c	17		
Error	identifier "s32" is undefined	single_file_analysis.c	18	Set option: -D s32=	Apply
Error	identifier "s8" is undefined	single_file_analysis.c	19		
Error	identifier "s16" is undefined	single_file_analysis.c	21	Set option: -D s16=	Apply

In this case, the software generates:

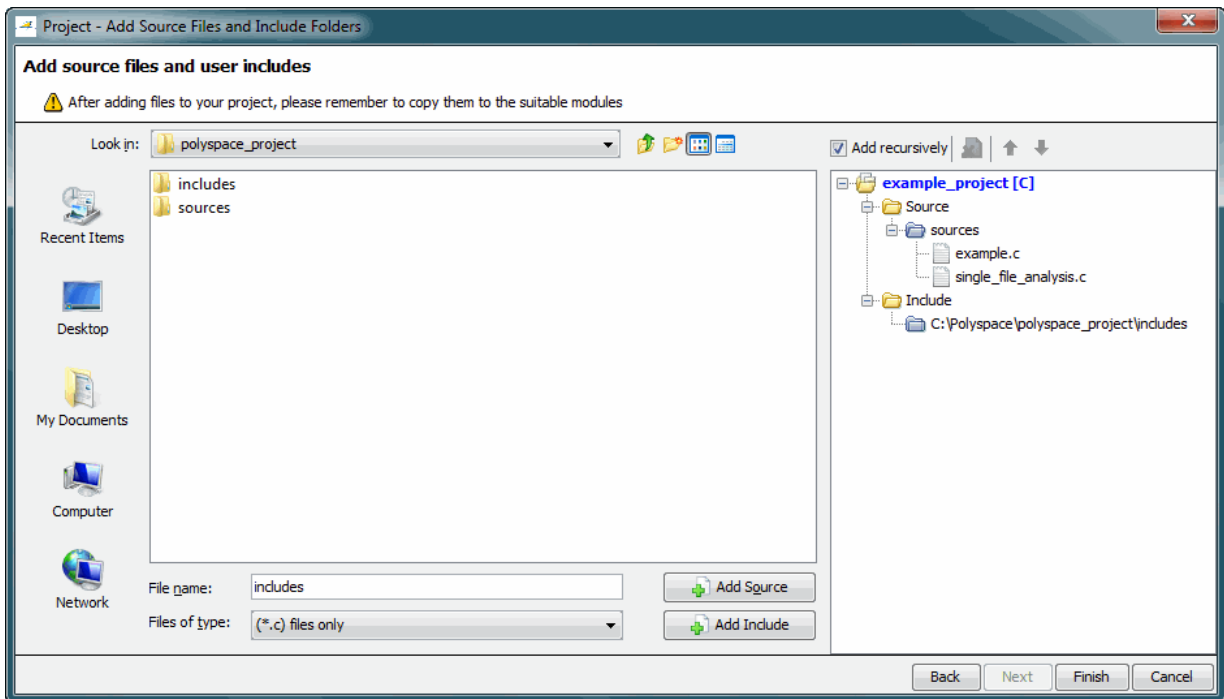
- Warnings for the missing include files.
- Errors for undefined identifiers, which stop the verification. The **Suggestion/Remark** column indicates that definitions for data types

3 Run a Verification

are required. This information is present in the include files that were removed.

- 7 In the **Project Browser** tree, right-click the **Include** folder. From the context menu, select **Add**.

The Add Source Files and Include Folders dialog box opens.




- 8 If you are not in the polyspace_project folder, navigate to this folder.


- 9 Select the includes folder. Then click **Add Include**.

The includes folder appears in the **Include** tree for example_project.

- 10 Click **Finish**.

- 11 In the Project Browser, select Module_1. On the Project Manager toolbar, click .

The verification should start and run to completion without any include file warnings.

- 12 In the Project Browser, select `Module_2`. On the Project Manager toolbar, click .

The verification should start and run to completion without any warnings or errors.


Start Server Verification from Project Manager

- “Start the Verification” on page 3-9
- “Monitor Verification Progress” on page 3-11
- “Remove Verification Results from the Server” on page 3-16
- “Troubleshoot a Failed Verification” on page 3-17

Start the Verification

In this part of the tutorial, you run the verification on a server.

To start a verification that runs on a server:

- 1 In the Project Manager perspective, on the **Configuration > Machine Configuration** pane, select the **Send to Polyspace Server** check box.
- 2 On the Project Manager toolbar, click .

Note If you see the message `Verification process failed`, click **OK** and go to “Troubleshoot a Failed Verification” on page 3-17.

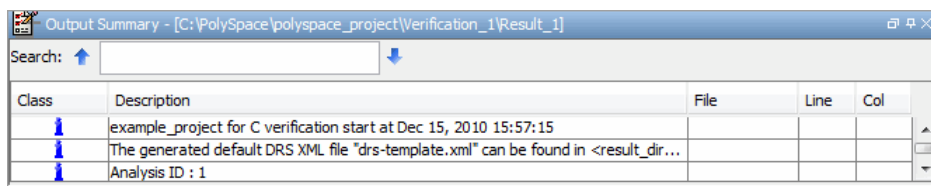
The verification has three main phases:

- a Checking syntax and semantics (the compile phase). Because Polyspace software is independent of any particular C compiler, it ensures that your code is portable, maintainable, and complies with ANSI® standards.

- b** Generating a main if the Polyspace software does not find a main and you have selected the `-main-generator` option. For more information, see “Main Generator Behavior for Polyspace Software”.
- c** Analyzing the code for run-time errors and generating color-coded results.

The compile phase of the verification runs on the client. When the compile phase is complete:

- You see the message `queued on server` at the bottom of the Project Manager perspective. This message indicates that the part of the verification that takes place on the client is complete. The rest of the verification runs on the server.
- A message in **Output Summary** gives you the identification number (Analysis ID) for the verification. For this verification, the identification number is 1.



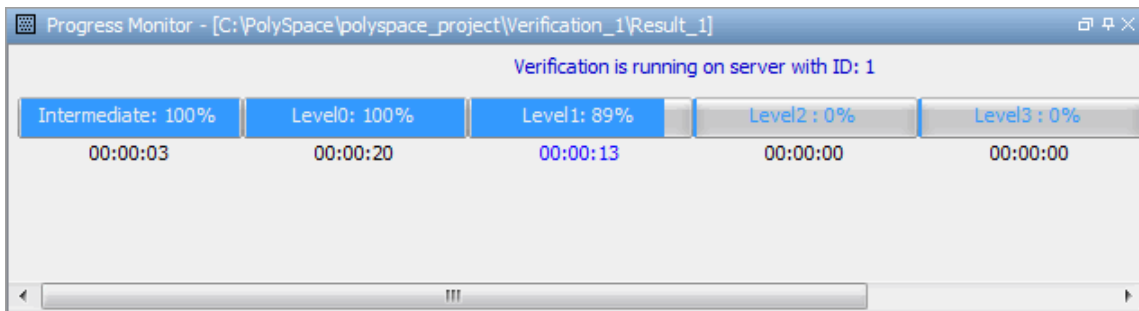
- 3** For information on any message in the log, click the message.

Monitor Verification Progress

There are two ways to monitor the progress of a verification:

- **Using the Project Manager** — Follow the progress of the verifications you submitted to the server, as well as client verifications.
- **Using the Queue Manager (Spooler)** — Follow the progress of any verification job in the server queue.

Monitor Progress Using Project Manager. You can monitor the progress of your verification by viewing the progress monitor and logs at the bottom of the Project Manager perspective.



The progress monitor highlights the current phase in blue. It displays the amount of time and completion percentage for that phase.

The logs report additional information about the progress of the verification. To view a log, click the button for that log. The information appears in the log display area at the bottom of the Project Manager window. To view the logs:

- 1 Click the **Output Summary** tab to display compile phase messages and errors. You can search the log by entering search terms in the **Search in the log** box. Click the left arrows to search back or the right arrows to search ahead.
- 2 Click the **Verification Statistics** tab to display statistics, such as analysis options, stubbed functions, and the verification checks performed.

- 3 Click the **Refresh** button  to update the display as the verification progresses.
- 4 Click the **Full Log** tab to display messages, errors, and statistics for all phases of the verification.

Note You can search the logs. In the **Search in the log** box, enter a search term and click the left arrows to search backward or the right arrows to search forward.

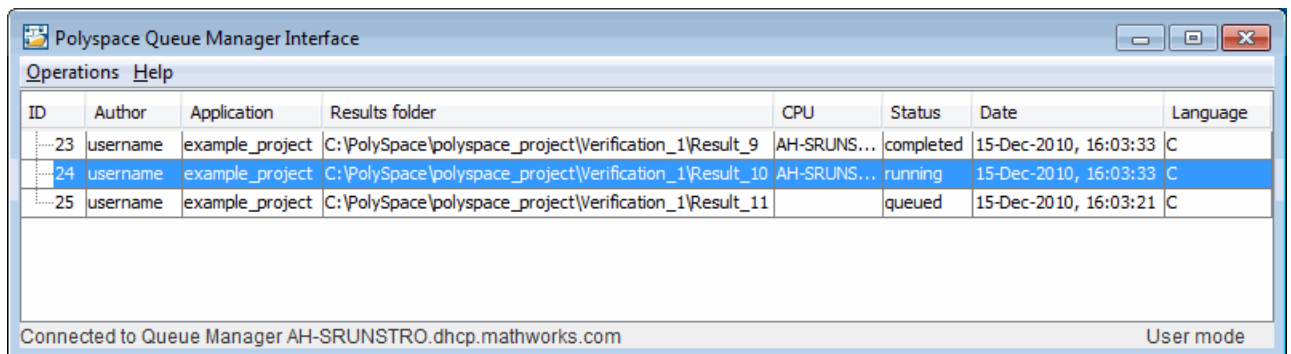
Monitor Progress Using Queue Manager. You monitor the progress of the verification using the Polyspace Queue Manager (also called the Spooler).


To monitor the verification of Example_Project:

- 1 On your desktop, double-click the **Polyspace Spooler** icon.

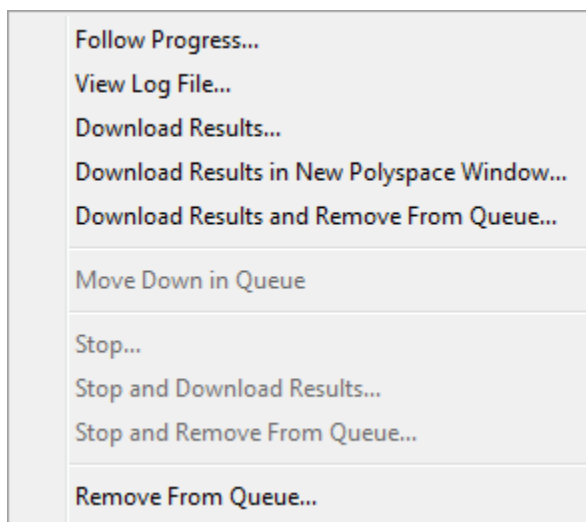


The Polyspace Queue Manager Interface opens.



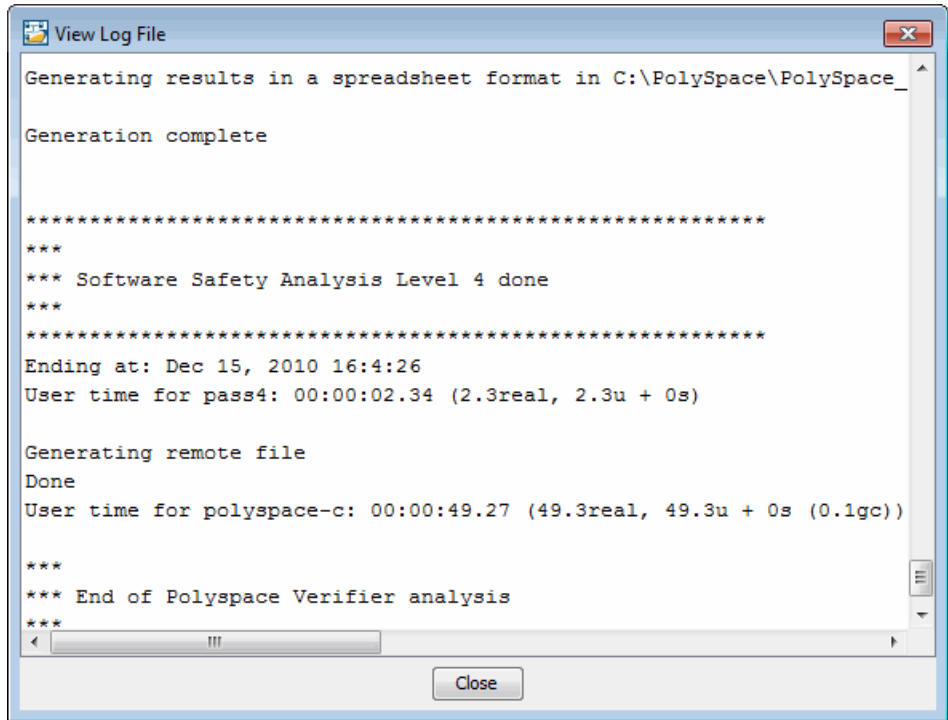
Tip You can also open the Polyspace Queue Manager Interface by clicking the Polyspace Queue Manager icon  on the Results Manager toolbar.

- 2 Point anywhere in the row for ID 1.
- 3 Right-click to open the context menu for this verification.



- 4 Select **View log file**.

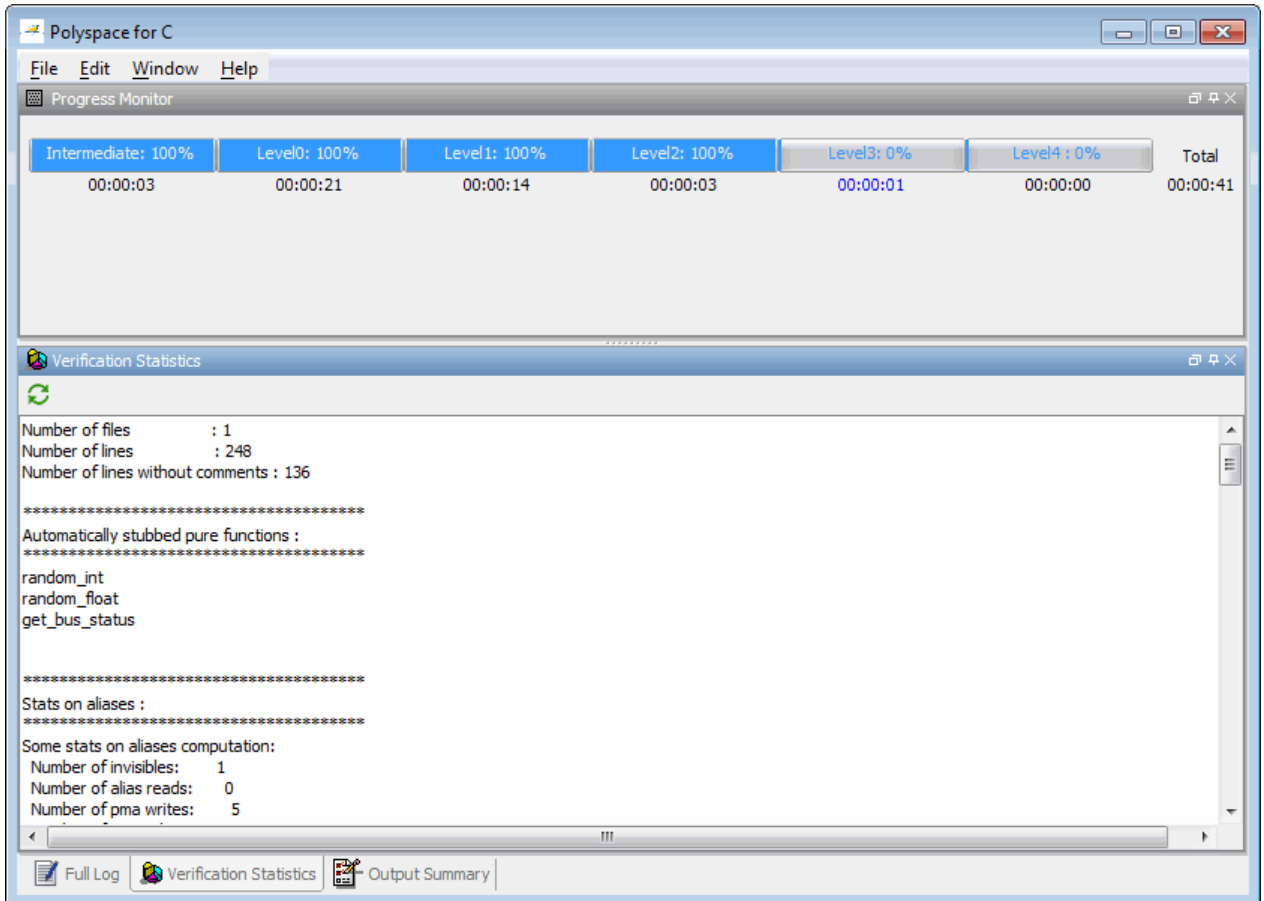
A window opens displaying the last 100 lines of the verification.



5 Click **Close** to close the window.


6 From the context menu, select **Follow Progress**.

The Progress Monitor opens.

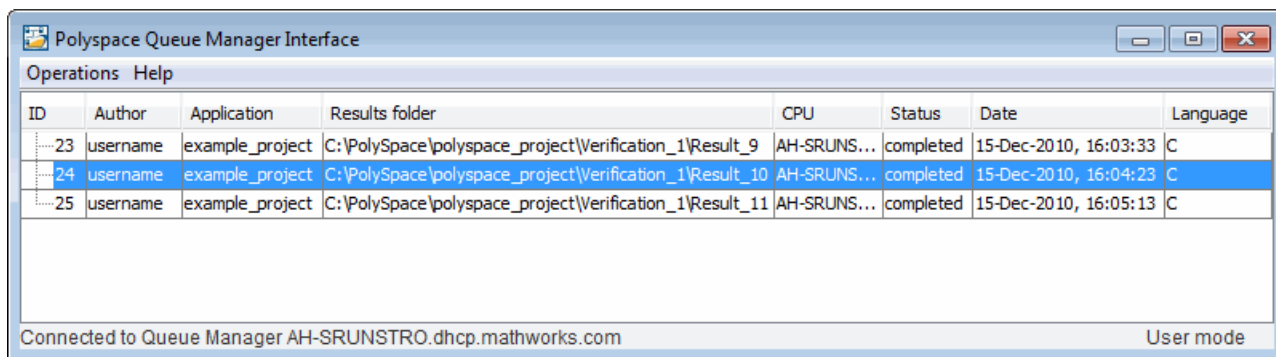


You can monitor the progress of the verification by watching the progress bar and viewing the logs at the bottom of the window. The progress monitor highlights the current phase in blue. It displays the amount of time and completion percentage for that phase.

The logs report additional information about the progress of the verification. To view a log, click the button for that log. The information appears in the log display area at the bottom of the Project Manager window. To view the logs:

- a Click the **Output Summary** tab to display compile phase messages and errors. You can search the log by entering search terms in the **Search in the log** box. Click the left arrows to search back or the right arrows to search ahead.
 - b Click the **Verification Statistics** tab to display statistics, such as analysis options, stubbed functions, and the verification checks performed.
 - c Click the **Refresh** button  to update the display as the verification progresses.
 - d Click the **Full Log** tab to display messages, errors, and statistics for all phases of the verification.
- 7 Select **File > Quit** to close the progress window.
- 8 Wait for the verification to finish.

When the verification is complete, the status in the Polyspace Queue Manager Interface changes from running to completed.



The screenshot shows the Polyspace Queue Manager Interface window. It contains a table with the following data:

ID	Author	Application	Results folder	CPU	Status	Date	Language
23	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_9	AH-SRUNS...	completed	15-Dec-2010, 16:03:33	C
24	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_10	AH-SRUNS...	completed	15-Dec-2010, 16:04:23	C
25	username	example_project	C:\PolySpace\polyspace_project\Verification_1\Result_11	AH-SRUNS...	completed	15-Dec-2010, 16:05:13	C

At the bottom of the window, it says "Connected to Queue Manager AH-SRUNSTRO.dhcp.mathworks.com" and "User mode".

Remove Verification Results from the Server

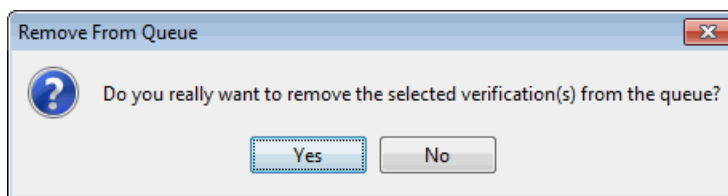
At the end of a server verification, the server automatically downloads verification results to the results folder specified in the project. You do not need to manually download your results. You can, however, manually download verification results to another location on your client system, or to other client systems.

Verification results remain on the server until you remove them. Once your results have been downloaded to the client, you can remove them from the server queue.

To remove your results from the server:

- 1 In the Polyspace Queue Manager Interface, right-click the verification, and select **Remove From Queue**.

A dialog box opens to confirm that you want to remove the verification from the queue.



- 2 Click **Yes**.

Note To download the results and remove the verification from the queue, right-click the verification and select **Download Results And Remove From Queue**. If you download results before the verification is complete, you get partial results and the verification continues.

- 3 Select **Operations > Exit** to close the Polyspace Queue Manager Interface.

Once the results are on your client, you can review them using the Results Manager perspective. You review results from the verification in “Review Verification Results” on page 4-2.

Troubleshoot a Failed Verification

When you see a message that the verification failed, it indicates that Polyspace software could not perform the verification. The following sections present some possible reasons for a failed verification.

Hardware Does Not Meet Requirements. If your computer does not have the minimum hardware requirements, the verification fails. For information about the hardware requirements, go to:

www.mathworks.com/products/polyspaceclientc/requirements.html.

To determine if hardware is the cause of the failed verification, search the log for the message:

```
Errors found when verifying host configuration.
```

You can:

- Upgrade your computer to meet the minimal requirements.
- In the General section of the Analysis options, select the **Continue with current configuration option** and run the verification again.

You Did Not Specify the Location of Include Files. If you see a message in the log such as the following, either the files are missing or you did not specify the location of include files.

```
include.h: No such file or folder
```

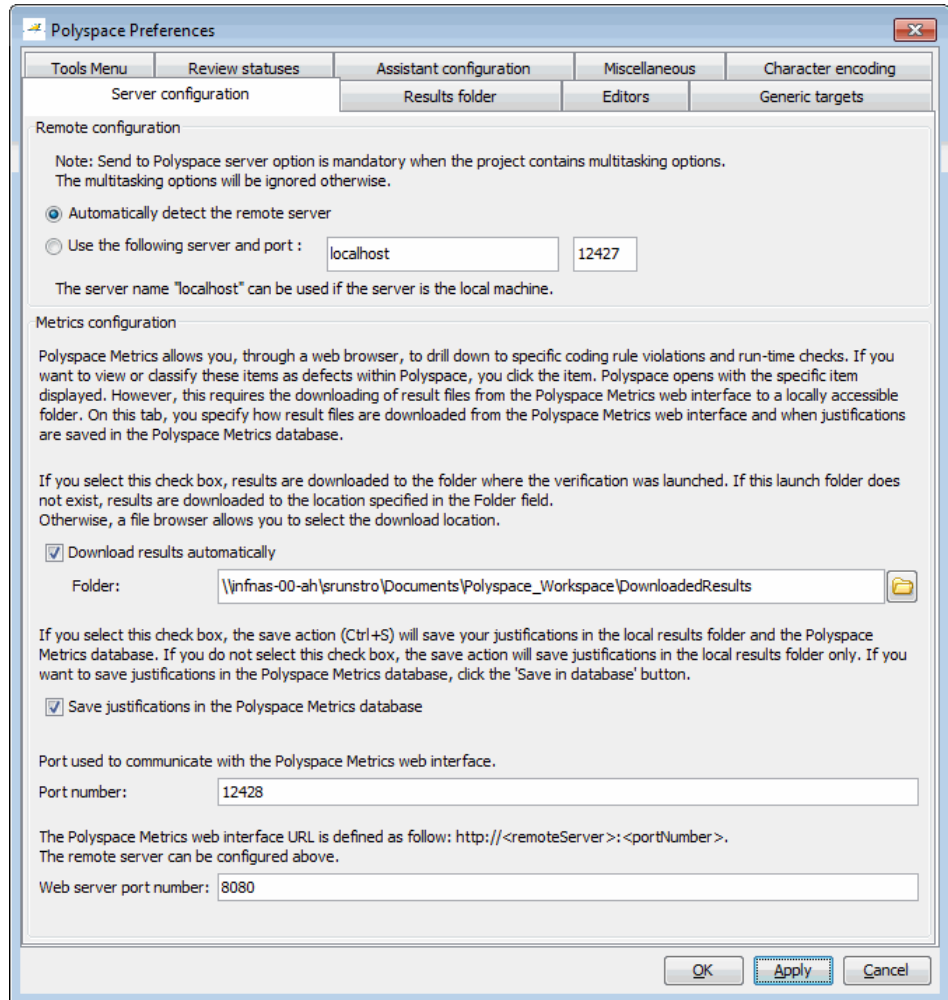
For information on how to specify the location of include files, see “Create a New Project to Verify the Example C File” on page 2-6.

Polyspace Software Cannot Find the Server. If you see the following message in the log, Polyspace software cannot find the server.

```
Error: Unknown host :
```

Polyspace software uses information in the preferences to locate the server. To find the server information in the preferences:

- 1** Select **Options > Preferences**.
- 2** Select the **Server configuration** tab.



By default, Polyspace software automatically finds the server. You can specify the server by selecting **Use the following server and port** and providing the server name and port. For information about setting up a server, see the “Software Configuration”.

Start Client Verification from Project Manager

- “Start the Verification” on page 3-20
- “Monitor the Progress of the Verification” on page 3-22
- “Complete Verification” on page 3-23
- “Stop the Verification Before Completion” on page 3-24

Start the Verification

For the best performance, run verifications on a server. If the server is busy or you want to verify a small file, you can run a verification on a client.


Note Because a verification on a client can process only a limited number of variable assignments and function calls, the source code should have no more than 800 lines of code.

To start a verification that runs on a client:

- 1** If the project `example_project.cfg` is not already open, open the project.

For information about opening a project, see “Prepare for Verification” on page 3-3.

- 2** In the Project Manager perspective, on the **Configuration > Machine Configuration** pane, clear the **Send to Polyspace Server** check box.

- 3** On the Project Manager toolbar, click .

- 4** If you see a caution that Polyspace software will remove existing results from the results folder, click **Yes** to continue and close the message dialog box.

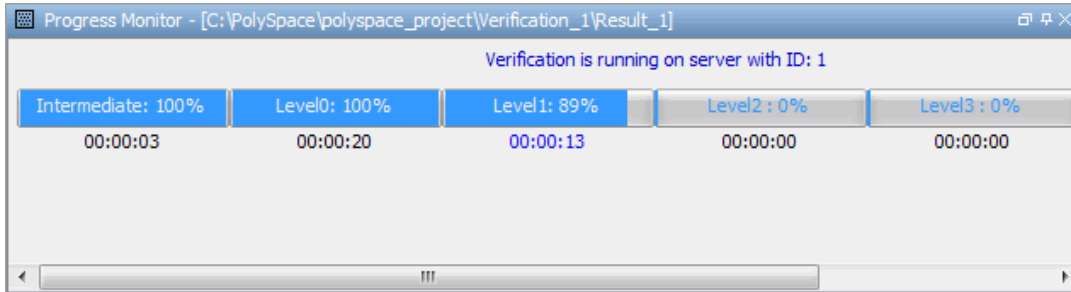
.

The Output Summary and Progress Monitor windows become active, allowing you to monitor the progress of the verification. If you see the

message Verification process failed, click **OK** and go to “Troubleshoot a Failed Verification” on page 3-17.


Monitor the Progress of the Verification

You can monitor the progress of the verification by viewing the progress monitor and logs at the bottom of the Project Manager perspective.



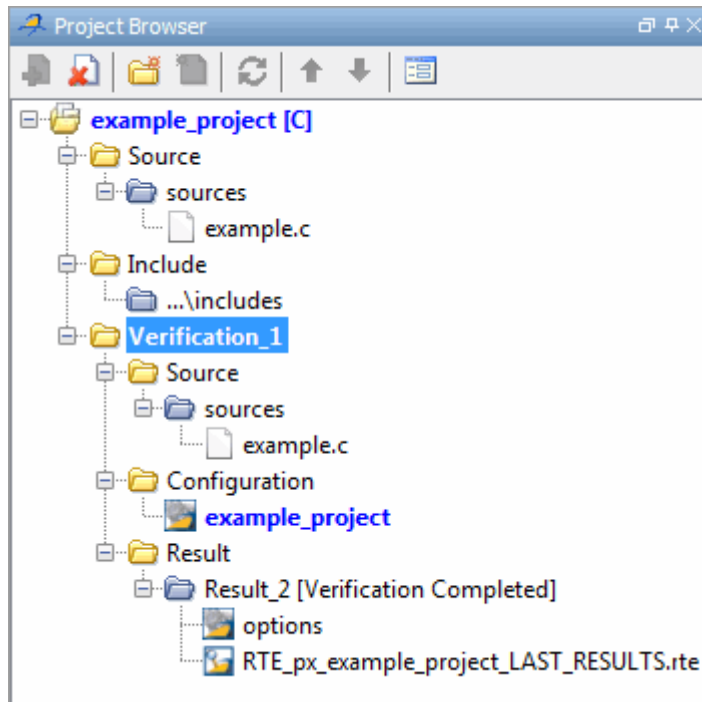
The progress monitor highlights the current phase in blue. It displays the amount of time and completion percentage for that phase.

The logs report additional information about the progress of the verification. To view a log, click the button for that log. The information appears in the log display area at the bottom of the Project Manager window. To view the logs:

- 1 Click the **Output Summary** tab to display compile phase messages and errors. You can search the log by entering search terms in the **Search in the log** box. Click the left arrows to search back or the right arrows to search ahead.
- 2 Click the **Verification Statistics** tab to display statistics, such as analysis options, stubbed functions, and the verification checks performed.
- 3 Click the **Refresh** button  to update the display as the verification progresses.
- 4 Click the **Full Log** tab to display messages, errors, and statistics for all phases of the verification.

Complete Verification

When the verification is complete, the message End of Polyspace Verifier analysis appears in **Full Log**. The results file appears in the **Project Browser** pane.




In the tutorial “Review Verification Results” on page 4-2 , you open the Results Manager perspective and review the verification results.

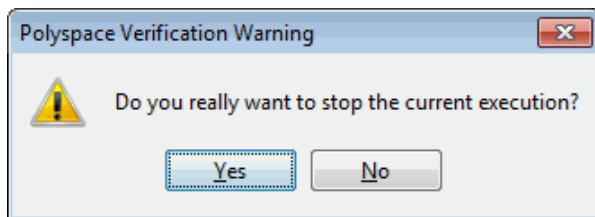
Stop the Verification Before Completion

You can stop the verification before it is complete. If you stop the verification, results are incomplete. If you start another verification, the verification starts from the beginning.

To stop a verification:

- 1 On the Project Manager toolbar, click the **Stop** button 

A warning dialog box opens.



- 2 Click **Yes**. The verification stops.
- 3 Click **OK** to close the Message dialog box.

Note Closing the Polyspace verification environment window does *not* stop the verification. To resume display of the verification progress, start the Polyspace software and open the project.

Review Verification Results

Review Verification Results

In this section...
“Tutorial Overview” on page 4-2
“Before You Start” on page 4-2
“Open Verification Results” on page 4-3
“Explore Results Manager perspective” on page 4-3
“Review Results” on page 4-7
“Review Results Systematically” on page 4-23
“Automatically Test Unproven Code” on page 4-28
“Generate Reports of Verification Results” on page 4-28

Tutorial Overview

In the previous tutorial, “Run Verification” on page 3-2, you completed a verification of `example.c`. In this tutorial, you explore the verification results.

The Polyspace verification environment contains a Results Manager perspective, which you use to review results. In this tutorial, you learn:

- 1 How to use the Results Manager perspective, including how to:
 - Open the Results Manager perspective and view verification results.
 - Review results.
 - Generate reports.
- 2 How to interpret the color coding that Polyspace software uses to identify the severity of an error.
- 3 How to find the location of an error in the source code.

Before You Start

Before starting this tutorial, be sure to complete the tutorial “Run Verification” on page 3-2.

In this tutorial, you use the verification results in this file:

```
polyspace_project\Verification_(1)\Result_(1)\  
RTE_px_example_project_LAST_RESULTS.rte.
```

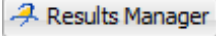
Open Verification Results

- “Open Results Manager perspective” on page 4-3
- “Open Verification Results” on page 4-3

Open Results Manager perspective

Use the Results Manager perspective to review verification results.

To open the Results Manager perspective:

- In the Polyspace verification environment toolbar, select the **Results Manager** button .

Open Verification Results

To open the verification results:

- 1** In the Polyspace verification environment, select **File > Open Result**.
- 2** In the Open results dialog box, navigate to the results folder:

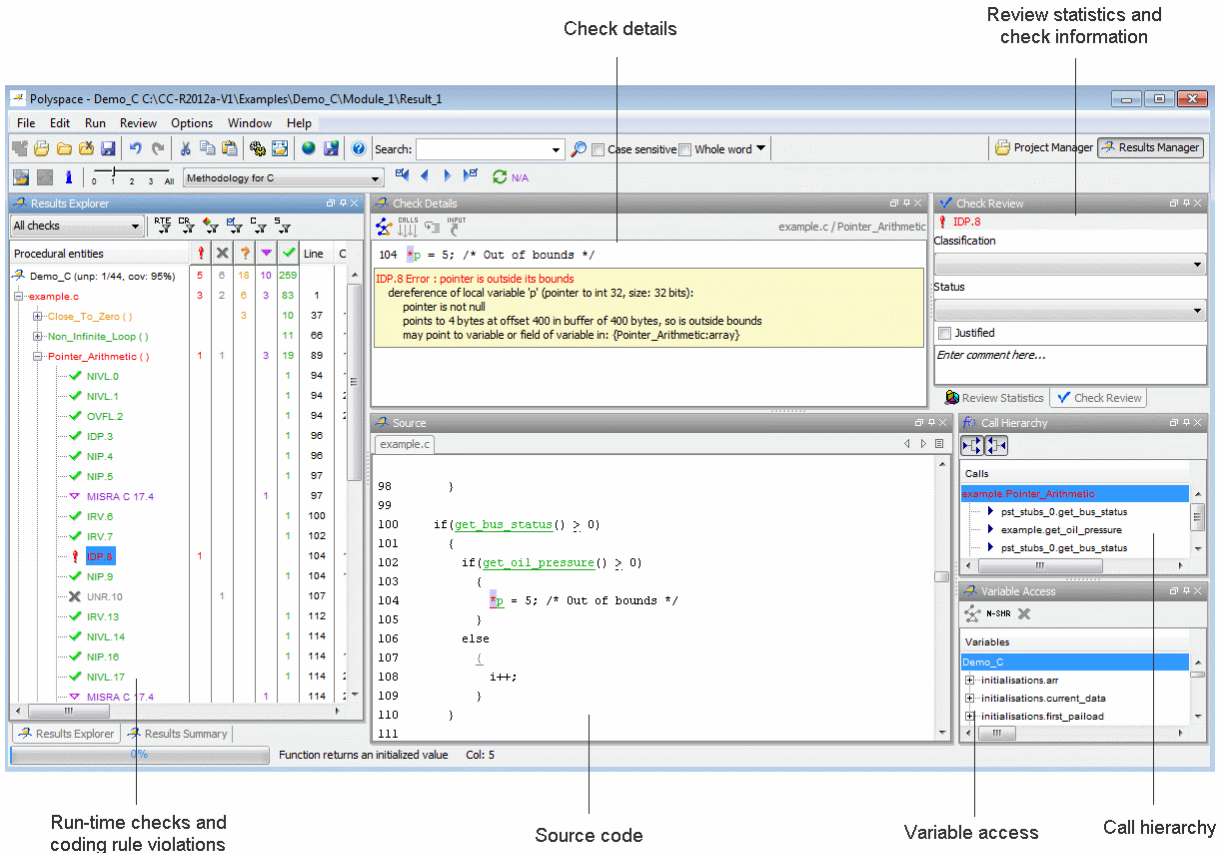
```
polyspace_project\Module_1\Result_2.
```
- 3** Select the file RTE_px_example_project_LAST_RESULTS.rte.
- 4** Click **Open**. The results appear in the Results Manager perspective. You can also open results from the Project Manager perspective by double-clicking the results file in the Project Browser.

Explore Results Manager perspective

- “Overview” on page 4-4
- “Review the Results Explorer Tab” on page 4-5

Overview

The Results Manager perspective looks like the following figure.



The Results Manager perspective has six sections below the toolbar. Each section provides a different view of the results. The following table describes these views.

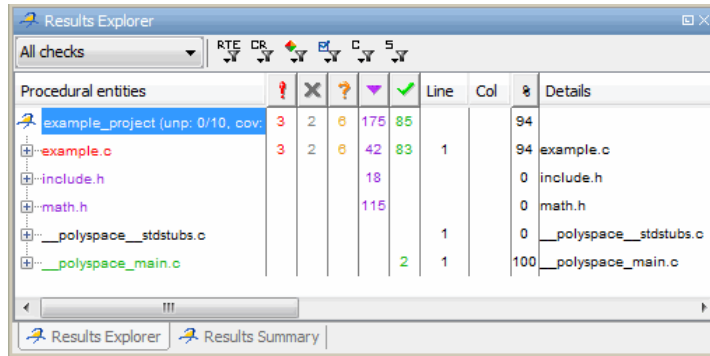
This Pane or View...	Displays...
Results Explorer\Results Summary (Procedural entities view)	List of the checks (diagnostics) for each file and function in the project
Source (Source code view)	Source code for a selected check in the procedural entities view
Check Details (Selected check view)	Details about the selected check
Check Review\Review Statistics (Coding review progress view)	Review information about the selected check. Statistics about the review progress for checks with the same type and category as the selected check
Variable Access (Variables view)	Information about global variables declared in the source code
Call Hierarchy (Call tree view)	Tree structure of function calls

You can resize or hide any of these sections. You learn more about the Results Manager perspective later in this tutorial.





Review the Results Explorer Tab



The **Results Explorer** view displays a table with information about the diagnostics for each file in the project. The **Results Explorer** view is also called the Procedural entities view.

When you first open the results file from the verification of `example.c`, you see the following procedural entities.



The file `example.c` is red because it has a run-time error. Polyspace software assigns each file the color of the most severe error found in that file. The first column of the table in the Procedural Entities view is the procedural entity (the file or function). The following table describes some of the other columns in the procedural entities view.

Column Heading	Indicates
	Number of red checks (operations where an error always occurs)
	Number of gray checks (unreachable code)
	Number of orange checks (warnings for operations where an error might occur)
	Number of purple checks (coding rule violations)

Column Heading	Indicates
	Number of green checks (operations where an error never occurs)
	Selectivity of the verification (percentage of checks that are not orange) This percentage is an indication of the level of proof.

You can select which columns appear in the procedural entities view by editing the preferences.

What you select in the procedural entities view determines what you see in the other views. In the following examples, you learn how to use the views and how they interact.

Review Results

- “What Are Review Levels?” on page 4-7
- “Display All Checks” on page 4-9
- “Review All Checks” on page 4-9
- “Review Additional Examples of Checks” on page 4-15
- “Filter Checks” on page 4-20

What Are Review Levels?

To facilitate your review of verification results, Polyspace allows you to specify the type of results displayed in the **Procedural entities** and **Source** views of the Results Manager perspective. You can specify:

- Type of coding rule violations, that is, purple checks
- Type and number of orange run-time checks

There are five levels at which you can review your results:

- 0 — The software displays violations of coding rules with state *Error* and red and gray run-time checks. You can configure the software to display orange checks that are potential run-time errors. Through the **Polyspace Preferences > Review Configuration** tab, specify the categories of potential run-time errors that you want the software to display. By default, the software does not display any orange checks at this level. See “Review Checks at Level 0” on page 4-23.

This level is suitable for the review of fresh code.

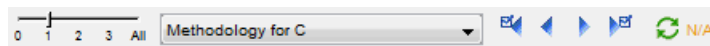
- 1, 2, and 3 — The software displays all purple checks *and* red, gray, and green run-time checks. The software displays orange checks according to values specified on the **Polyspace Preferences > Review Configuration**. You can use either a predefined or a custom methodology to specify the number of orange checks per check category. See “Review Checks at Levels 1, 2, and 3” on page 4-24.

For a predefined methodology, these levels are suitable for reviews at the following stages of the development process.

Level	Development Stage
1	Fresh code
2	Unit tested code
3	Code Review

- All — The software displays all purple checks *and* red, gray, green, and orange checks. When you want to carry out an exhaustive review of your verification results, use this level. See “Review All Checks” on page 4-9.

In the Results Manager perspective, the toolbar provides controls specific to review levels.



The controls include:

- A slider for selecting the review level — 0, 1, 2, 3, or All. By default, the Results Manager perspective opens at level 1.
- A menu for selecting the review methodology for levels 1, 2, and 3.

- Arrows for navigating through checks.

Display All Checks

By default, the Results Manager perspective opens at level 1. To display all checks in the Procedural entities view, move the Review Level slider to All.

Review All Checks

In this part of the tutorial, you learn how to use the Results Manager perspective to examine verification checks. This part of the tutorial covers:

- “Select a Check to Review” on page 4-9
- “Display the Calling Sequence” on page 4-12
- “Track Review Progress” on page 4-12

Select a Check to Review. In the procedural entities view, `example.c` is red, indicating that this file has at least one red check. To review a red check in `example.c`:

- 1 In the procedural entities section of the **Results Explorer** view, expand `example.c`.
- 2 Expand the red procedure `Pointer_Arithmetic()`.

A color-coded list of the checks performed on `Pointer_Arithmetic()` opens.

Pointer_Arithmetic ()	1	1	1	16	89	12	95	example.c
✓ NIVL.5				1	94	13		Local variable is initialized (type: int 32)
✓ NIVL.3				1	94	22		Local variable is initialized (type: int 32)
✓ OVFL.4				1	94	23		Operation [+] on scalar does not overflow in INT32 range
✓ IDP.1				1	96	6		Pointer is within its bounds
✓ NIP.0				1	96	7		Pointer is initialized
✓ NIP.2				1	97	6		Pointer is initialized
! IDP.9	1				104	10		Error : pointer is outside its bounds
✓ NIP.8				1	104	11		Pointer is initialized
✗ UNR.23		1			107	8		Unreachable code
✓ NIVL.16				1	114	6		Local variable is initialized (type: int 32)
? IDP.15			1		114	16		Warning : pointer may be outside its bounds
✓ NIP.14				1	114	18		Pointer is initialized
✓ NIVL.13				1	114	20		Local variable is initialized (type: int 32)
✓ NIVL.22				1	116	11		Local variable is initialized (type: int 32)
✓ NIVL.21				1	118	18		Local variable is initialized (type: int 32)
✓ NIP.18				1	118	10		Pointer is initialized
✓ NIVL.17				1	118	14		Local variable is initialized (type: int 32)
✓ IDP.20				1	119	6		Pointer is within its bounds
✓ NIP.19				1	119	7		Pointer is initialized

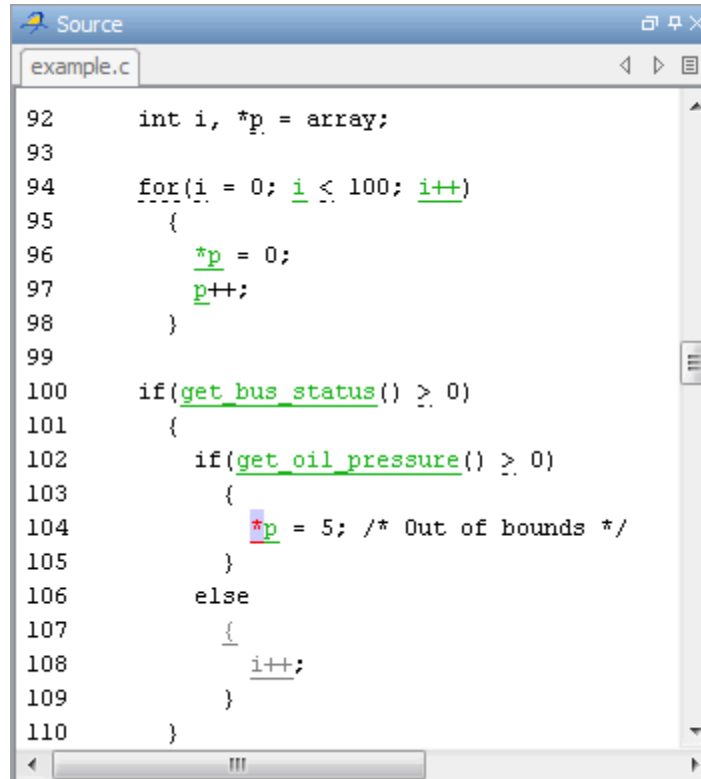
In the list of checks, each item has an acronym and a number. The acronym indicates the check type and the number identifies the check within the function. For example, with IDP.9:

- IDP stands for Illegal Dereferenced Pointer
- 9 states that the check is the ninth check in the function

You can identify a check uniquely by stating the file name, function name, check acronym and check number. For information about different types of checks, see “Run-Time Checks for C Code”.

3 Click the red IDP.9.

The **Source** pane displays the section of source code where this error occurs.



```
92     int i, *p = array;
93
94     for(i = 0; i < 100; i++)
95     {
96         *p = 0;
97         p++;
98     }
99
100    if(get_bus_status() >= 0)
101    {
102        if(get_oil_pressure() >= 0)
103        {
104            *p = 5; /* Out of bounds */
105        }
106        else
107        {
108            i++;
109        }
110    }
```


- 4** At line 104 of the code, click the red code.

An error message box opens indicating that when the pointer `p` is dereferenced, it is outside of its bounds. At line 92, `p` points to the start of array which has 100 elements. The for loop starting at line 94 initializes the elements of array to 0. This for loop leaves `p` pointing to the location after the last element of array.

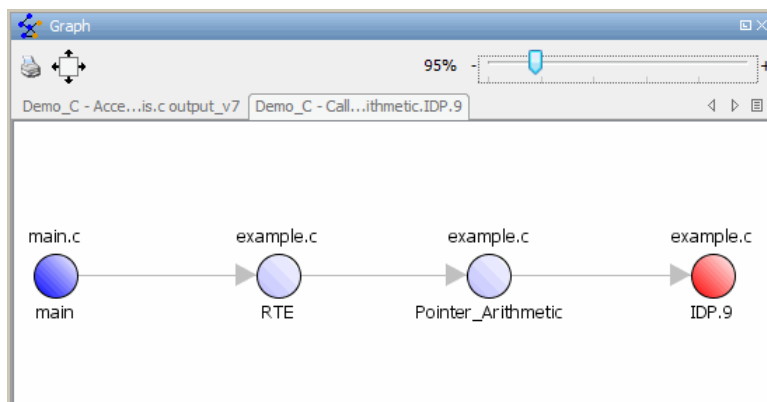
Display the Calling Sequence. You can display the calling sequence that leads to the code associated with a check. To see the calling sequence for the red IDP.9 check in `Pointer_Arithmetic()`:

1 Expand `Pointer_Arithmetic()`.

2 Click the red IDP.9.

3 In the Check Review toolbar, click the **call graph** button .

A window displays the call graph.



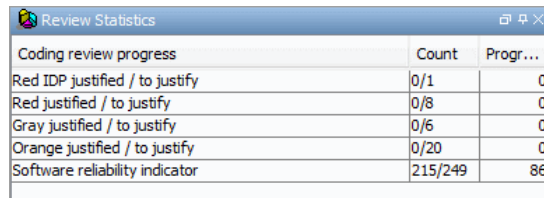
The code associated with IDP.9 is in `Pointer_Arithmetic`. The generated main function calls `RTE`, which calls `Pointer_Arithmetic`.

Track Review Progress. You can keep track of the checks that you have reviewed by marking them. To mark that you have reviewed the red IDP.9 check in `Pointer_Arithmetic()`:

1 Expand `Pointer_Arithmetic()`.

2 Click the red IDP.9.

The **Review Statistics** view displays a table with statistics about the review progress.



Review Statistics	Count	Progr...
Coding review progress		
Red IDP justified / to justify	0/1	0
Red justified / to justify	0/8	0
Gray justified / to justify	0/6	0
Orange justified / to justify	0/20	0
Software reliability indicator	215/249	86

The **Count** column displays a ratio and the **Progress** column displays the equivalent percentage.

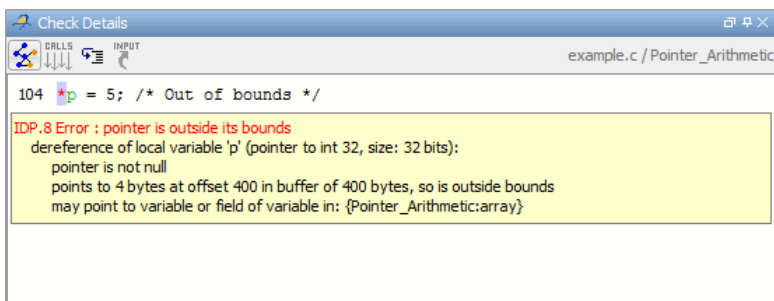
The first row displays the ratio of justified checks to total checks that have the same color and category as the current check. In this example, the first row displays the ratio of reviewed red IDP checks to total red IDP errors in the project.

The second, third, and fourth rows display the ratio of justified checks to total checks for red, gray, and orange checks respectively.

If you specified coding rules checking, the next row displays the ratio of justified coding rule violations to total coding rule violations.

The last row displays the ratio of the number of green checks to the total number of run-time checks, providing an indicator of the reliability of the software.

Information about the current check (the red IDP.9) appears in the **Check Details** pane (selected check view).



- 3 After you review the check, from the **Check Review** tab, select a **Classification** to describe the severity of the issue:

- High
- Medium
- Low
- Not a defect

4 From the **Check Review** tab, select a **Status** to describe how you intend to address the issue:

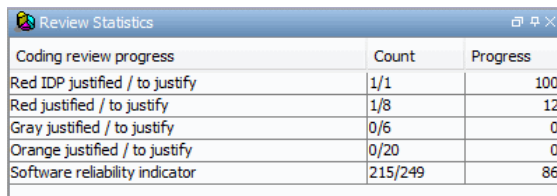
- Fix
- Improve
- Investigate
- Justify with annotations
- No Action Planned
- Other
- Restart with different options
- Undecided

Note You can also define your own statuses. See “Define Custom Status”.

5 On the **Check Review** tab, in the comment box, enter additional information about the check.

6 Select the check box to indicate that you have justified this check.

The **Coding review progress** part of the window updates the ratios of errors reviewed to total errors.



Coding review progress	Count	Progress
Red IDP justified / to justify	1/1	100
Red justified / to justify	1/8	12
Gray justified / to justify	0/6	0
Orange justified / to justify	0/20	0
Software reliability indicator	215/249	86

Review Additional Examples of Checks

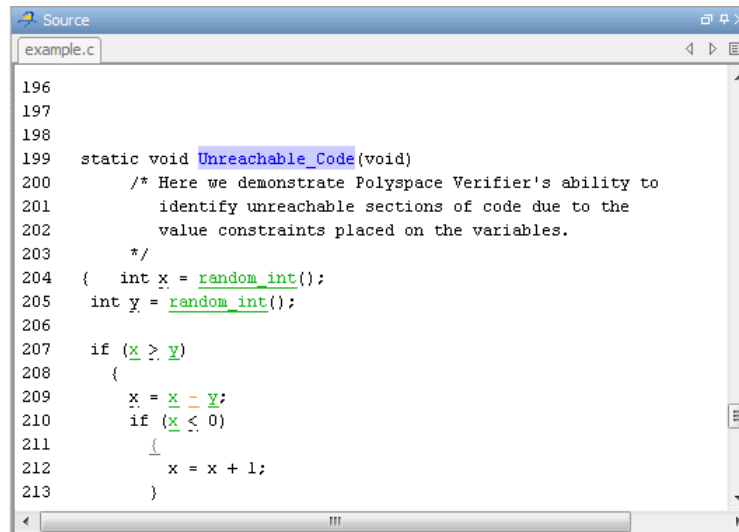
In this part of the tutorial, you learn about other types and categories of errors by reviewing the following examples in `example.c`:

- “Example: Unreachable Code” on page 4-16
- “Example: Arithmetic Error” on page 4-17
- “Example: A Function with No Errors” on page 4-18
- “Example: Division by Zero” on page 4-18

Example: Unreachable Code. Unreachable code is code that never executes. Polyspace software displays unreachable code in gray. In the following example, you look at an example of unreachable code.

1 In Procedural Entities, click `Unreachable_Code()`.

The source code view displays the source code for this function.



```
196
197
198
199 static void Unreachable_Code(void)
200     /* Here we demonstrate Polyspace Verifier's ability to
201        identify unreachable sections of code due to the
202        value constraints placed on the variables.
203        */
204     { int x = random_int();
205       int y = random_int();
206
207       if (x >= y)
208       {
209         x = x - y;
210         if (x < 0)
211         {
212             x = x + 1;
213         }
214     }
```

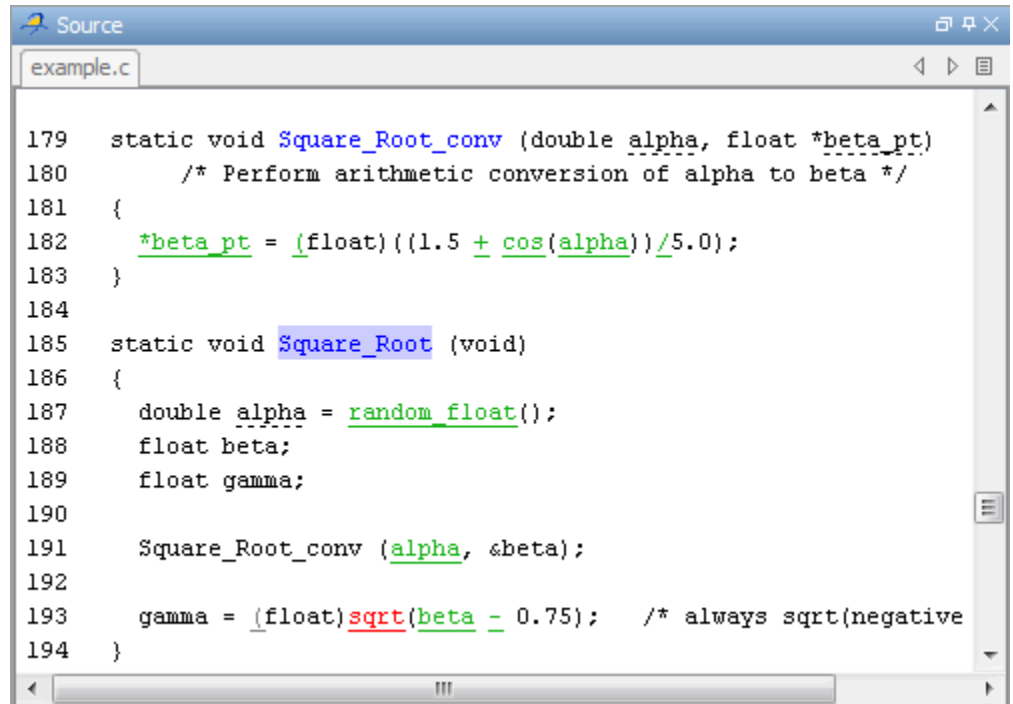
2 Examine the source code.

At line 210, the condition `x < 0` is always false. The curly bracket `{` is gray because the branch is never executed.

Example: Arithmetic Error. In the following example, Polyspace software detects a memory corruption error:

- 1 In the Procedural entities view, expand the red `Square_Root()` function.

The source code view displays the source code for this function.



```
179 static void Square_Root_conv (double alpha, float *beta_pt)
180     /* Perform arithmetic conversion of alpha to beta */
181 {
182     *beta_pt = (float)((1.5 + cos(alpha))/5.0);
183 }
184
185 static void Square_Root (void)
186 {
187     double alpha = random_float();
188     float beta;
189     float gamma;
190
191     Square_Root_conv (alpha, &beta);
192
193     gamma = (float)sqrt(beta - 0.75); /* always sqrt(negative
194 }
```

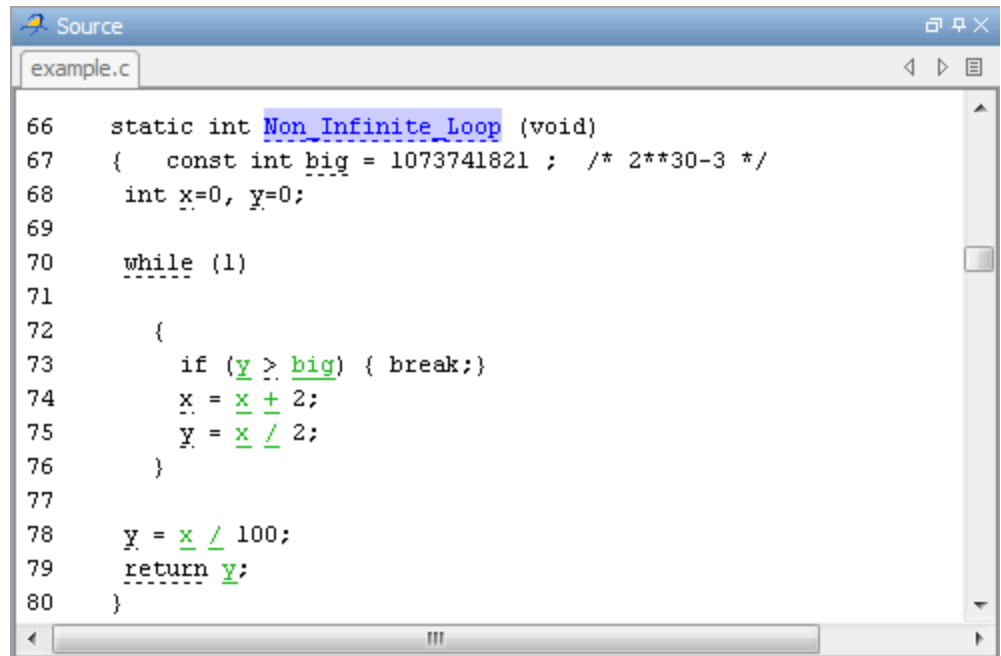
- 2 Examine the source code.

Because `beta` is always less than 0.75, the argument to the `sqrt()` function at line 193 is always negative.

Example: A Function with No Errors. In the following example, Polyspace software verifies code with a large number of iterations, and determines that the loop terminates and a variable does not overflow:

- 1 In Procedural entities, click the green `Non_Infinite_Loop()` function.

The source code view displays the source code for this function.



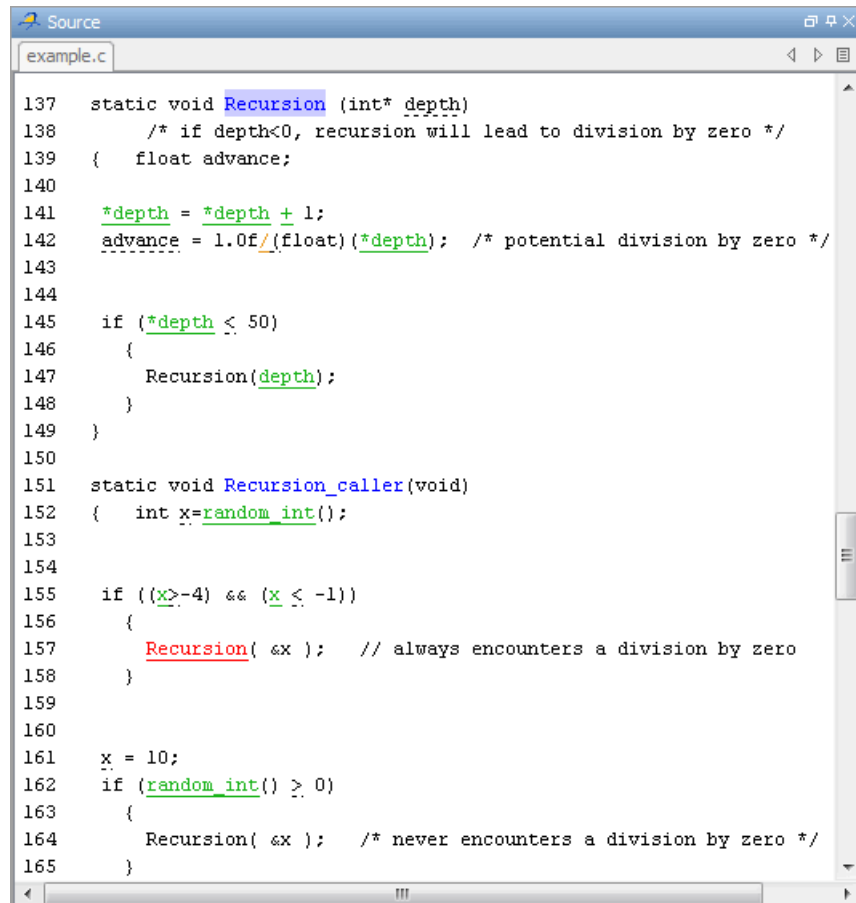
```
66 static int Non_Infinite_Loop (void)
67 { const int big = 1073741821 ; /* 2**30-3 */
68   int x=0, y=0;
69
70   while (1)
71   {
72     if (y > big) { break;}
73     x = x + 2;
74     y = x / 2;
75   }
76
77   y = x / 100;
78   return y;
79 }
80 }
```

- 2 Examine the source code. The variable `x` never overflows because the `while` loop at line 70 terminates before `x` can overflow.

Example: Division by Zero. In the following example, Polyspace software detects division by zero:

- 1 In Procedural entities, expand `Recursion()`.

The source code view displays the source code for this function.



```
137 static void Recursion (int* depth)
138     /* if depth<0, recursion will lead to division by zero */
139 {   float advance;
140
141     *depth = *depth + 1;
142     advance = 1.0f/(float)(*depth); /* potential division by zero */
143
144
145     if (*depth <= 50)
146     {
147         Recursion(depth);
148     }
149 }
150
151 static void Recursion_caller(void)
152 {   int x=random_int();
153
154
155     if ((x>=4) && (x <= -1))
156     {
157         Recursion( &x ); // always encounters a division by zero
158     }
159
160
161     x = 10;
162     if (random_int() >= 0)
163     {
164         Recursion( &x ); /* never encounters a division by zero */
165     }
```

2 Examine the Recursion() function.

When Recursion() is called with depth less than zero, the code at line 142 results in division by zero. The orange color indicates that this operation is a potential error (depending on the value of depth).

3 Examine the red Recursion_caller function.

The first call to Recursion() at line 157 is red because it calls Recursion() with depth less than zero, causing a division by zero. The

second call to `Recursion()` at line 164 does not cause division by zero because it calls `Recursion()` with `depth` greater than zero.

Filter Checks

To focus on certain checks, you can filter checks that you see in the Results Manager perspective. Polyspace software allows you to filter your results in several ways. You can filter:

- Run-time checks or coding rule violations
- Run-time checks by category (for example, ZDV, IDP, and NIP)
- Violations of selected coding rules
- Run-time checks by color (gray, orange, green)
- Justified or unjustified checks
- Checks by classification
- Checks by status

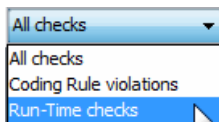
To filter checks, on the **Results Explorer** or **Results Summary** toolbar, select one of the following filters.



The tooltip for a filter button indicates what the filter does.


Example: Filter Coding Rule Violations

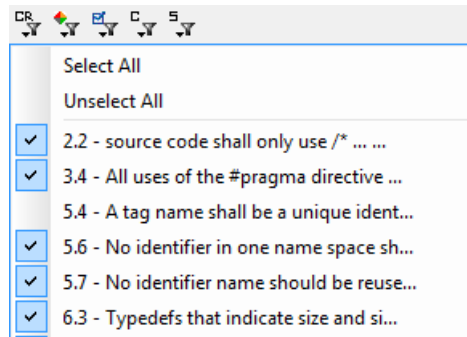
To hide all coding rule violations, on the **Results Explorer** or **Results Summary** toolbar, from the drop-down list of the first filter, select Run-time checks.



The software hides all coding rule violations and displays only run-time checks.

To filter violations of a specific coding rule:

- 1 On the **Results Explorer** or **Results Summary** toolbar, click the coding rules filter icon .
- 2 From the drop-down list, clear the check box for the coding rule, for example, 5.4.



The software does not display violations of this rule.

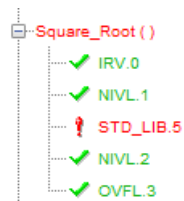
Example: Filter IRV Checks

You can use an RTE filter to hide a given check category, such as IRV. When a filter is enabled, you do not see that check category.

To filter IRV checks:

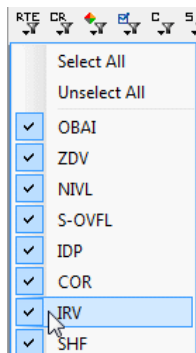
- 1 Expand `Square_Root()`.

`Square_Root()` has five checks: four are green and one is red.

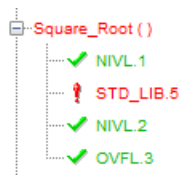


- 2 Click the **RTE filter** icon .

3 Clear the IRV option.



The software hides the IRV check for `Square_Root()`.



4 Select the IRV option to redisplay the IRV check.

When you filter a check category, red checks of that category are not hidden. For example, if you filter IDP checks, you still see IDP.9 under `Pointer_Arithmetic()`.

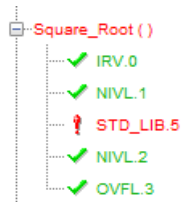
Example: Filter Green Checks


You can use a Color filter to hide certain color checks. When a filter is enabled, you do not see that color check.

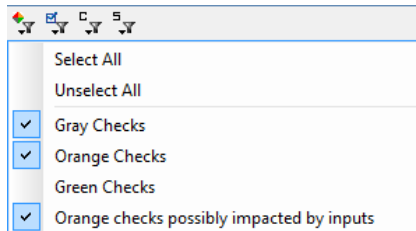
To filter green checks:

1 Expand `Square_Root()`.

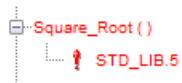
`Square_Root()` has five checks: four are green and one is red.



- 2 Click the **Color filter** icon .
- 3 Clear the **Green Checks** option.



The software hides the green checks.



Review Results Systematically

- “Review Checks at Level 0” on page 4-23
- “Review Checks at Levels 1, 2, and 3” on page 4-24
- “Review Checks Progressively” on page 4-26

Review Checks at Level 0

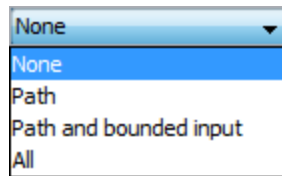
In addition to red and gray checks, at this level you can focus on orange checks that Polyspace identifies as potential run-time errors. These potential run-time errors fall into three categories:

- **Path** — The software identifies orange checks that are path-related issues, which are not dependent on input values.

- **Path and bounded input** — Default. In addition to orange checks that are path-related issues, the software identifies orange checks that are related to bounded input values.
- **All** — In addition to path-related and bounded input orange checks, the software identifies orange checks that are related to unbounded input values.

To specify the potential run-time error category for level 0:

- 1** In the Polyspace verification environment, select **Options > Preferences**.
- 2** In the Polyspace Preferences dialog box, select the **Review configuration** tab.
- 3** From the **Level** drop-down list, select your category.



The default is **Path and bounded input**. If you select **None**, the software displays only red and gray checks.

- 4** Click **OK** to save your options and close the Polyspace Preferences dialog box.

To select review level 0, on the Results Manager toolbar, move the Review Level slider to **0**.

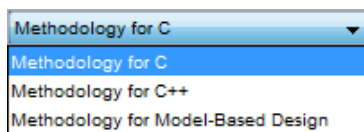
Review Checks at Levels 1, 2, and 3

In addition to red, gray, and green checks, the software displays orange checks according to values specified on the **Review Configuration** tab in the Polyspace Preferences dialog box. See “View Methodology Requirements for Levels 1, 2, and 3” on page 4-25.

You can use either a predefined or a custom methodology to specify the number of orange checks per check category.

To select a predefined methodology and review level:

- 1** From the Results Manager perspective, select **Options > Preferences**.
- 2** In the Polyspace Preferences dialog box, select the **Review configuration** tab.
- 3** From the **Methodology** drop-down list, select, for example, Methodology for C.



- 4** Move the Review Level slider to your required level, for example, level **1**.



View Methodology Requirements for Levels 1, 2, and 3. In this part of the tutorial, you examine **Methodology for C**, which defines the number of orange checks that you review at levels 1, 2, or 3.

To examine the configuration for **Methodology for C**:

- 1** In the Polyspace verification environment, select **Options > Preferences**.
- 2** In the Polyspace Preferences dialog box, select the **Review configuration** tab.
- 3** From the **Methodology** drop-down list, select Methodology for C.


In the section **Levels 1, 2, and 3**, a table shows the number of orange checks that you review for a given level and check category.

Levels 1, 2, and 3			
	Level 1	Level 2	Level 3
Common			
ZDV	5	20	ALL
NIVL	10	50	ALL
S-OVFL	10	50	ALL
COR		10	10
NIV		0	10
F-OVFL	5	10	20
ASRT		5	20
C & C++ only			
OBAI	10	20	ALL
SHF	5	10	ALL
IDP		10	20
NIP		10	20
STD_LIB			
C only			
IRV	5	20	ALL

For example, the table specifies that you review five orange ZDV checks when you select level 1. The number of checks increases as you move from level 1 to level 3, reflecting the changing review requirements as you move through the development process.

4 Click **OK** to close the dialog box.

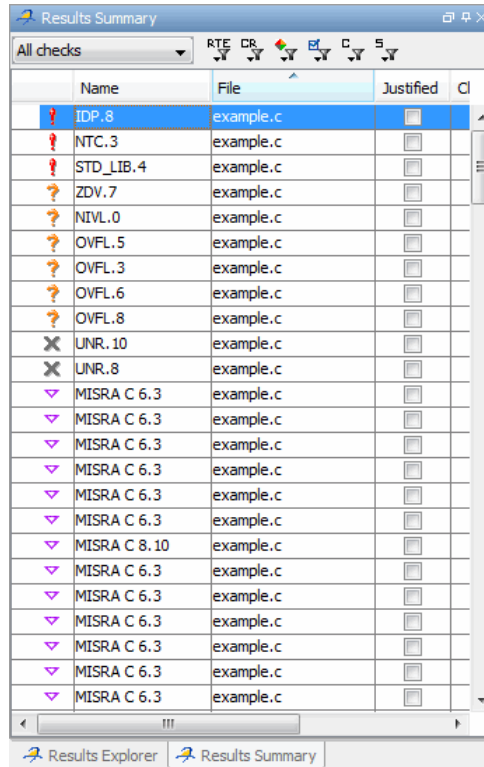
Review Checks Progressively


On the Results Manager perspective toolbar, use the forward arrow  to move to the next unjustified check. The software takes you through checks in the following order:

- All red checks
- All gray checks (the first check in each unreachable function).
- Orange checks — the number of orange checks is determined by the methodology and review level that you select.

Earlier in this tutorial, you selected Methodology for C, criterion 1. In this part of the tutorial, you review the checks for `example.c` using this methodology and criterion. To navigate through these checks:

- 1 Select the **Results Summary** view.



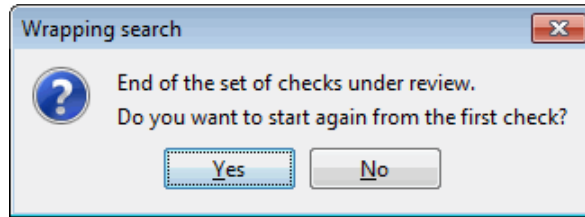
- 2 Click the forward arrow  to move to the next unjustified check.

The **Source** pane displays the source code for this check. The **Check Details** pane displays information about this check.

You can display the calling sequence and track review progress, as described in “Review Results” on page 4-7.

- 3 Continue to click the forward arrow until you have gone through all of the checks.

After the last check, a dialog box opens asking if you want to start again from the first check.



4 Click No.

Automatically Test Unproven Code

Reviewing orange code to find true errors is a time-consuming task. You can use the Automatic Orange Tester to automatically create and run test cases to identify errors in the orange code. The workflow for using the Automatic Orange Tester is:

- 1 Set an option to indicate that you want the software to run the Automatic Orange Tester at the end of the verification.
- 2 Run the verification. The software uses results from the Automatic Orange Tester to identify potential run-time errors.
- 3 If you want to perform further dynamic tests on the code, run the Automatic Orange Tester manually.
- 4 Review the results.

To learn how to use the Automatic Orange Tester, see “Automatically Test Orange Code”.

Generate Reports of Verification Results

- “Polyspace Report Generator” on page 4-28
- “Generate Report for example.c” on page 4-30

Polyspace Report Generator

The Polyspace Report Generator allows you to generate reports about your verification results, using predefined report templates.

The Polyspace Report Generator provides the following report templates:

- **Coding Rules Report** – Provides information about compliance with MISRA C Coding Rules, as well as Polyspace configuration settings for the verification.
- **Developer Report** – Provides information useful to developers, including summary results, detailed lists of red, orange, and gray checks, and Polyspace configuration settings for the verification. Detailed results are sorted by type of check (Proven Run-Time Violations, Proven Unreachable Code Branches, Unreachable Functions, and Unproven Run-Time Checks).
- **Developer Review Report** – Provides the same information as the Developer Report, but reviewed results are sorted by review classification (High, Medium, Low, Not a defect) and status. Untagged checks are sorted by file location.
- **Developer with Green Checks Report** – Provides the same content as the Developer Report, but also includes a detailed list of green checks.
- **Quality Report** – Provides information useful to quality engineers, including summary results, statistics about the code, graphs showing distributions of checks per file, and Polyspace configuration settings for the verification.
- **Software Quality Objectives Report** – Provides comprehensive information on software quality objectives (SQO), including code metrics, code analysis (coding-rules checker results), code verification (run-time checks), and the configuration settings for the verification. The code metrics section provides the same information that is displayed in the Polyspace Metrics web interface.

The Polyspace Report Generator allows you to generate verification reports in the following formats:

- HTML
- PDF
- RTF
- DOC (Microsoft Word)
- XML

Note Microsoft Word format is not available on UNIX platforms. If you select Word format on a UNIX platform, the software uses RTF format instead.

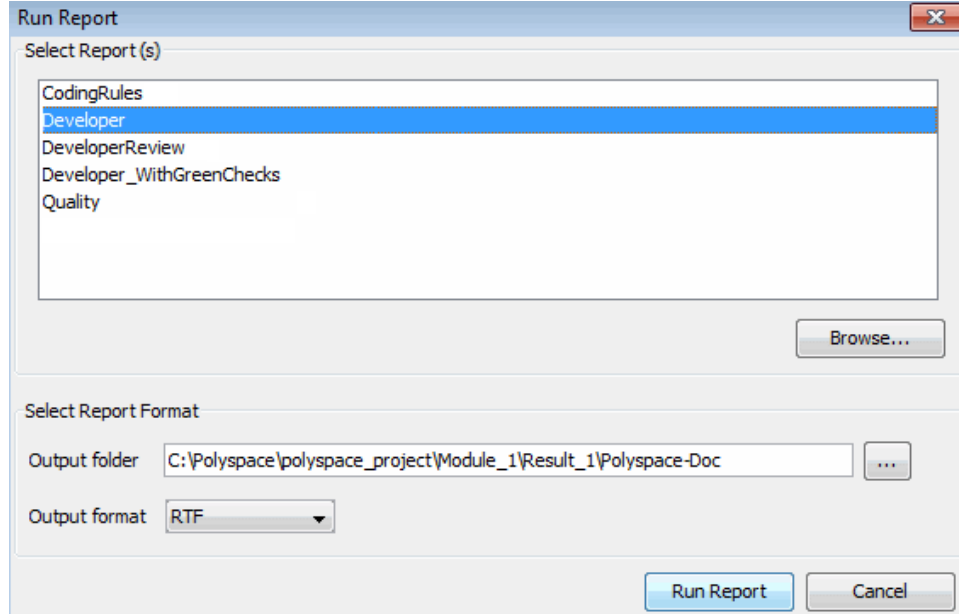
Generate Report for example.c

You can generate reports for any verification results using the Polyspace Report Generator.

To generate a verification report:

- 1 If your verification results are not already open, open them.
- 2 Select **Run > Run Report > Run Report...**

The Run Report dialog box opens.



- 3** In the Select Report Template section, select **Developer.rpt**.
- 4** In the Output folder section, select the \polyspace_project folder.
- 5** Select PDF Output format.
- 6** Click **Run Report**.

The software creates the specified report. When report generation is complete, the report opens.

Check Compliance with Coding Rules

Check Compliance with Coding Rules

In this section...
“Tutorial Overview” on page 5-2
“Before You Start” on page 5-3
“Create New Module for Coding Rules Checking” on page 5-3
“Set MISRA C Checking Option” on page 5-10
“Select Coding Rules to Check” on page 5-11
“Exclude Files from MISRA C Checking” on page 5-14
“Run a Verification with Coding Rules Checking” on page 5-15
“Examine MISRA C Violations” on page 5-16
“Open MISRA-C Report” on page 5-19

Tutorial Overview

Polyspace software allows you to analyze code to demonstrate compliance with established C or C++ coding standards (MISRA C 2004, MISRA C++:2008, or JSF++:2005).²

Applying coding rules can both reduce the number of orange checks in your verification results and improve the quality of your code. Coding rules are the most efficient way to reduce orange checks.

To check compliance with coding rules, you set an option in your project and then run a verification. Polyspace software finds the violations during the compile phase of a verification. When you have addressed all coding rule violations, you run the verification again.

For more information on the coding rules checker, see “Overview of Polyspace Code Analysis”.

In this tutorial, you learn how to:

2. MISRA and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

- 1 Create a module within your project.
- 2 Set an option for checking MISRA C compliance.
- 3 Select MISRA C rules to check.
- 4 Run a verification with MISRA C checking.
- 5 View coding rules violations using the Coding Rules perspective.

Before You Start

Before you start this tutorial, you must complete “Set Up Polyspace Project” on page 2-2. For this tutorial, you use the folders from that tutorial.

Create New Module for Coding Rules Checking

- “Open Your Example Project” on page 5-3
- “Create New Module for MISRA C Checking” on page 5-4
- “Configure Text and XML Editors” on page 5-9

Open Your Example Project

For this tutorial, you modify the project in `example_project.cfg` to include MISRA C checking. You use the Project Manager perspective to modify the project.


To open `example_project.cfg`:

- 1 Select **File > Open Project**.
- 2 In the Open a Polyspace project file dialog box, navigate to `polyspace_project`.
- 3 Select `example_project.cfg`.
- 4 Click **Open** to open the file and close the dialog box.

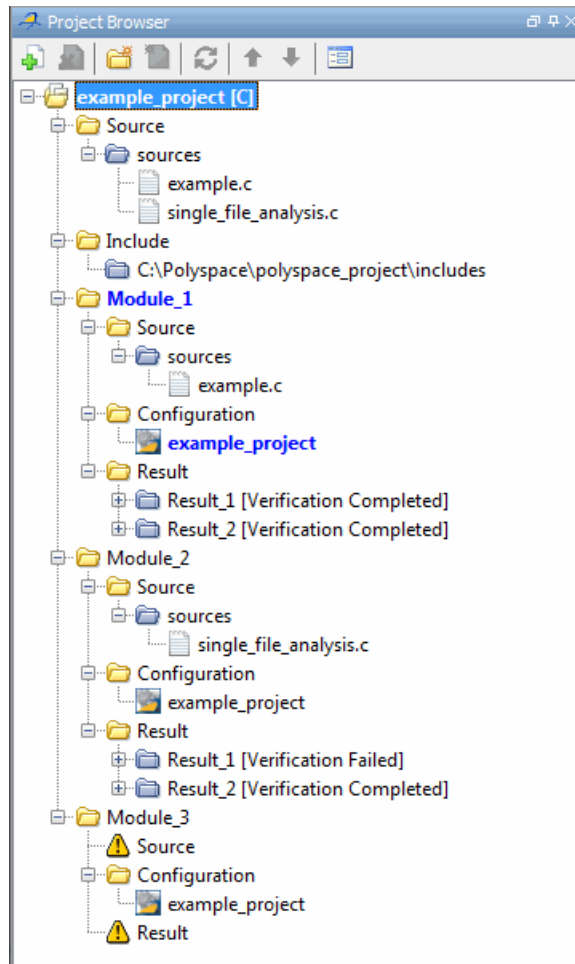
Create New Module for MISRA C Checking

A Polyspace project can contain multiple modules. Each of these modules can verify a specific set of source files using a specific set of analysis options. In this section, you create a third module to check coding rules compliance for the `example.c` file.

To create a new module in `example_project`:

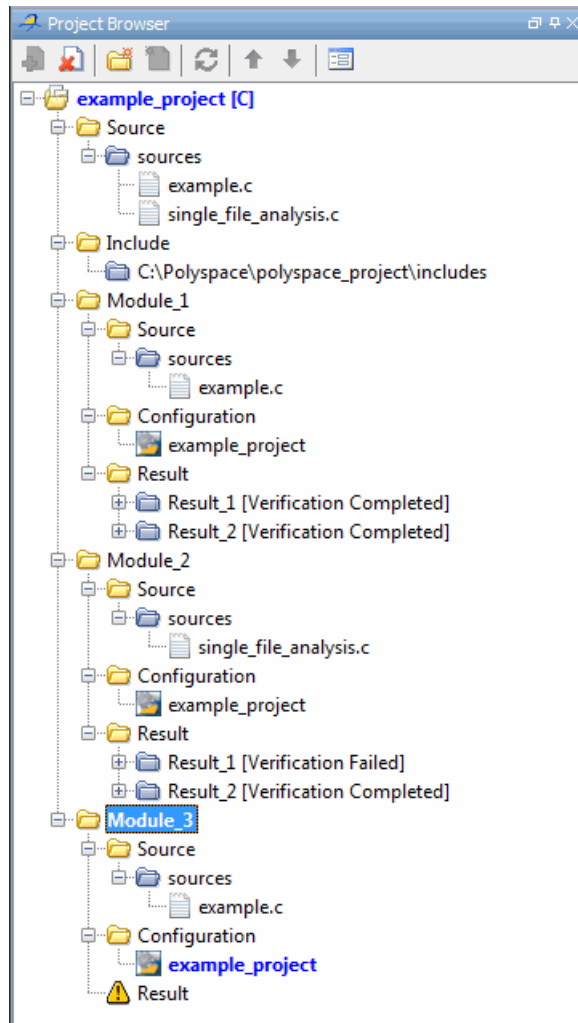
- 1 In the Project Browser, select **example_project [C]**.
- 2 Click the **Create a new module** icon  in the Project Browser toolbar.

A new verification, `Module_3`, appears in the Project Browser.



- 3 In the Project Browser Source tree, right-click `example.c`, and select **Copy Source File to > Module_2**.

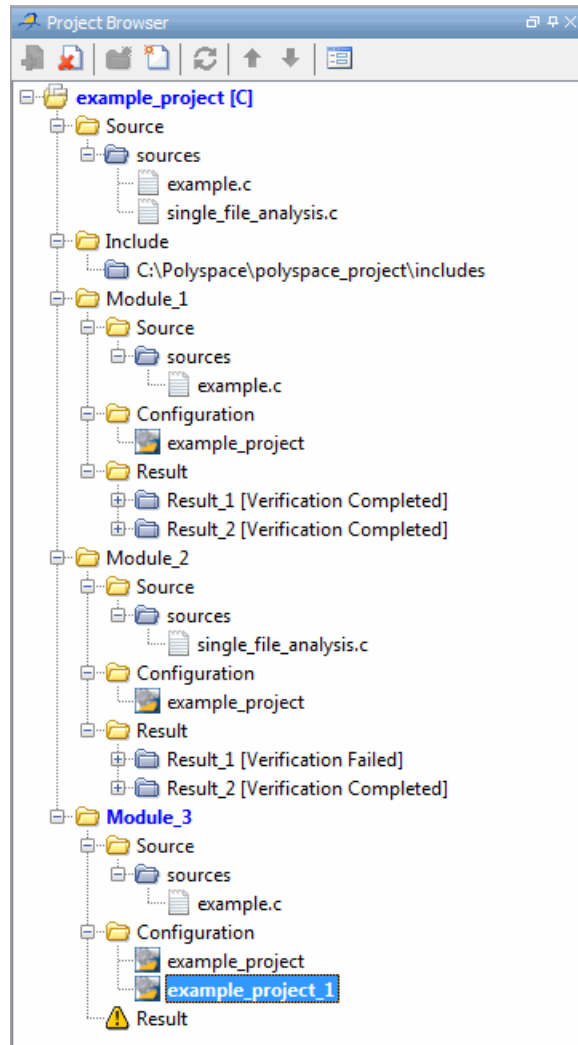
The `example.c` file appears in the Source tree of `Module_2`.



4 Right-click the Configuration folder in Module_3, and select **Create New Configuration**.

5 Right-click the example_project_1 configuration, and select **Set as Active Configuration**.

The Project Browser now looks like the following figure.

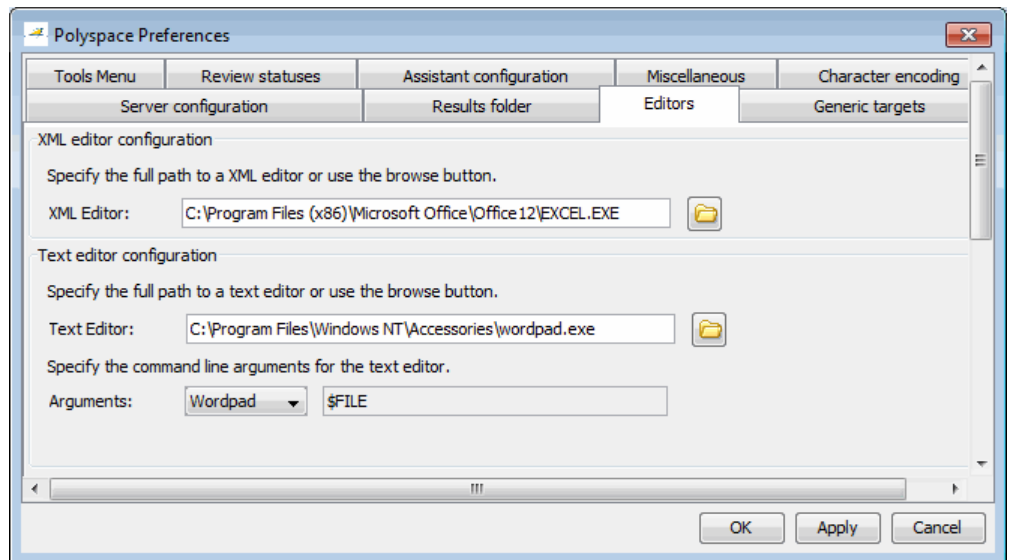


Configure Text and XML Editors

Before you check MISRA rules, configure your text and XML editors in the Polyspace Preferences dialog box. Configuring text and XML editors allows you to view source files and MISRA reports directly from the Results Manager perspective.

To configure your text and .XML editors:

- 1 From the Polyspace verification environment toolbar, select **Options > Preferences**.
- 2 In the Polyspace Preferences dialog box opens, select the **Editors** tab.



- 3 Specify an **XML editor** to use to view MISRA-C reports. For example:

C:\Program Files\MSOffice\Office12\EXCEL.EXE

- 4 Specify a **Text Editor** to use to view source files from the Project Manager logs. For example:

C:\Program Files\Windows NT\Accessories\wordpad.exe

5 From the **Arguments** drop-down list, select your text editor to automatically specify the command-line arguments for that editor.

- Emacs
- Notepad++ — Windows only
- UltraEdit
- VisualStudio
- WordPad — Windows only
- gVim

If you are using another text editor, select **Custom** from the drop-down list, and specify the command-line arguments for the text editor.

6 Click **OK**.

Set MISRA C Checking Option

You set up MISRA C checking by setting an analysis option and then selecting the rules to check. To set the MISRA C checking option:

- 1** In the Project Browser, select the `example_project_1` configuration.
- 2** Select the **Configuration > Coding Rules & Code Complexity Metrics** pane.
- 3** Select the **Check MISRA C rules** check box.
- 4** Use the corresponding drop-down list to specify the rules. For example, select `required-rules`.
- 5** You can also specify the following options:
 - **Files and folders to ignore** — Files, if any, to exclude from the checking.
 - **Effective boolean types** — Data types that you want Polyspace to consider as Boolean.
 - **Allowed pragmas** — Undocumented pragma directives for which rule MISRA C 3.4 must not be applied.

Select Coding Rules to Check

You must have a rules file to run a verification with MISRA C checking. You can use an existing file or create a new one. You create a new rules file for this tutorial by:

- “Creating a MISRA C Rules File” on page 5-11
- “Set All Rules to Off” on page 5-13
- “Selecting Rules to Check” on page 5-13

Creating a MISRA C Rules File

To open a new rules file:

- 1** In the Project Manager perspective, select the **Configuration > Coding Rules & Code Complexity Metrics** pane.
- 2** Select the **Check MISRA C rules** check box.
- 3** From the corresponding drop-down list, select custom.
- 4** Click the **Edit** button. The New File dialog box opens, displaying a table of rules.

5 Check Compliance with Coding Rules

Set the following state to all MISRA C rules : **Error** Apply

Rule	Error	Warning	Off	Comment
MISRA C rules				
Number of rules by mode:	0	130	12	
1 Environment				
2 Language extensions				
2.1 Assembly language shall be encapsulated and isolated	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	
2.2 source code shall only use /* ... */ style comments	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	
2.3 The character sequence /* shall not be used within a comment	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	
2.4 Sections of code should not be 'commented out'	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	Not implemented
3 Documentation				
4 Character sets				
5 Identifiers				
6 Types				
7 Constants				
8 Declarations and definitions				
9 Initialization				
10 Arithmetic type conversions				
11 Pointer type conversions				
12 Expressions				
13 Control statement expressions				
14 Control flow				
15 Switch statements				
16 Functions				
17 Pointers and arrays				
18 Structures and unions				
19 Preprocessing directives				
20 Standard libraries				
21 Run-time failures				

OK Cancel

5 For each rule, specify one of the following states.

State	Causes the verification to ...
Error	End after the compile phase when this rule is violated.
Warning	Display warning message and continue verification when this rule is violated.
Off	Skip checking of this rule.

The default state for most rules is **Warning**. The state for rules that have not yet been implemented is **Off**. Some rules have a fixed state of **Error**, which you cannot change.

6 Click **OK**.

7 Use the Save as dialog box to save your rules file.

Set All Rules to Off

In this tutorial, you select only a few rules. Therefore, first set the state of all rules to **Off**. Later, you can select the specific rules that you want to check.

To set the state of all rules to **Off**:

1 In the New File dialog box, from the **Set the following state to all MISRA rules** drop-down list, select **Off**.

2 Click **Apply**.

Selecting Rules to Check

To select the rules to check for this tutorial:

1 Expand the set of rules named **16 Functions**.

2 Select the **Error** column for **16.3**.

3 Expand the set of rules named **17 Pointers and arrays**.

4 Select the **Warning** column for **17.4**.

The completed rules table looks like the following figure.

Rules	Error	Warning	Off
MISRA C rules			
Number of rules by mode :	1	1	140
1 Environment			
2 Language extensions			
3 Documentation			
4 Character sets			
5 Identifiers			
6 Types			
7 Constants			
8 Declarations and definitions			
9 Initialisation			
10 Arithmetic type conversions			
11 Pointer type conversions			
12 Expressions			
13 Control statement expressions			
14 Control flow			
15 Switch statements			
16 Functions			
16.1 Functions shall not be defined with variable numbers of arguments.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
16.2 Functions shall not call themselves, either directly or indirectly.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
16.3 Identifiers shall be given for all of the parameters in a function prototype.	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
16.4 The identifiers used in the declaration and definition of a function shall match.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
16.5 Functions with no parameters shall be declared with parameter type void.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
16.6 The number of arguments passed to a function shall match the number in the prototype.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
16.7 A pointer parameter in a function prototype should be declared as pointer to void.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
16.8 All exit paths from a function with non-void return type shall have an explicit return statement.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
16.9 A function identifier shall only be used with either a preceding &, or with a pointer to void.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
16.10 If a function returns error information, then that error information shall be of type void.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
17 Pointer and arrays			
17.1 Pointer arithmetic shall only be applied to pointers that address an array of the same type.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
17.2 Pointer subtraction shall only be applied to pointers that address elements of the same type.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
17.3 >, >=, <, <= shall not be applied to pointer types except where they are used to compare pointers to the same object.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
17.4 Array indexing shall be the only allowed form of pointer arithmetic.	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
17.5 The declaration of objects should contain no more than 2 levels of pointer indirection.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
17.6 The address of an object with automatic storage shall not be assigned to a pointer.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
18 Structures and unions			


5 Click **OK** to save the rules and close the window.

6 In the Save as dialog box, in the **File**, field, enter `misrac.txt`

7 Click **OK** to save the file and close the dialog box.

Exclude Files from MISRA C Checking

You can exclude files from MISRA C checking. You might want to exclude some included files. To exclude `math.h` from the MISRA C checking of the project `example_project.cfg`:


- 1 In the Project Manager perspective, select the **Configuration > Coding Rules & Code Complexity Metrics** pane.
- 2 Select the **Files and folders to ignore** check box.
- 3 From the corresponding drop-down list, select **custom**.
- 4 In the **File/Folder** view, click .
- 5 Use the Open File dialog box to navigate to the folder `polyspace_project\includes`.
- 6 Select the file `math.h`.
- 7 Click **Open**.

You see the file `math.h` in the **File/Folder** view.

Run a Verification with Coding Rules Checking

When you run a verification with the MISRA C option selected, the software checks most of the MISRA C rules during the compile phase.³

To start the verification:

- 1 In the Project Browser, select your project configuration, for example, `example_project_1`.
- 2 On the Project Manager toolbar, click the **Run** button .

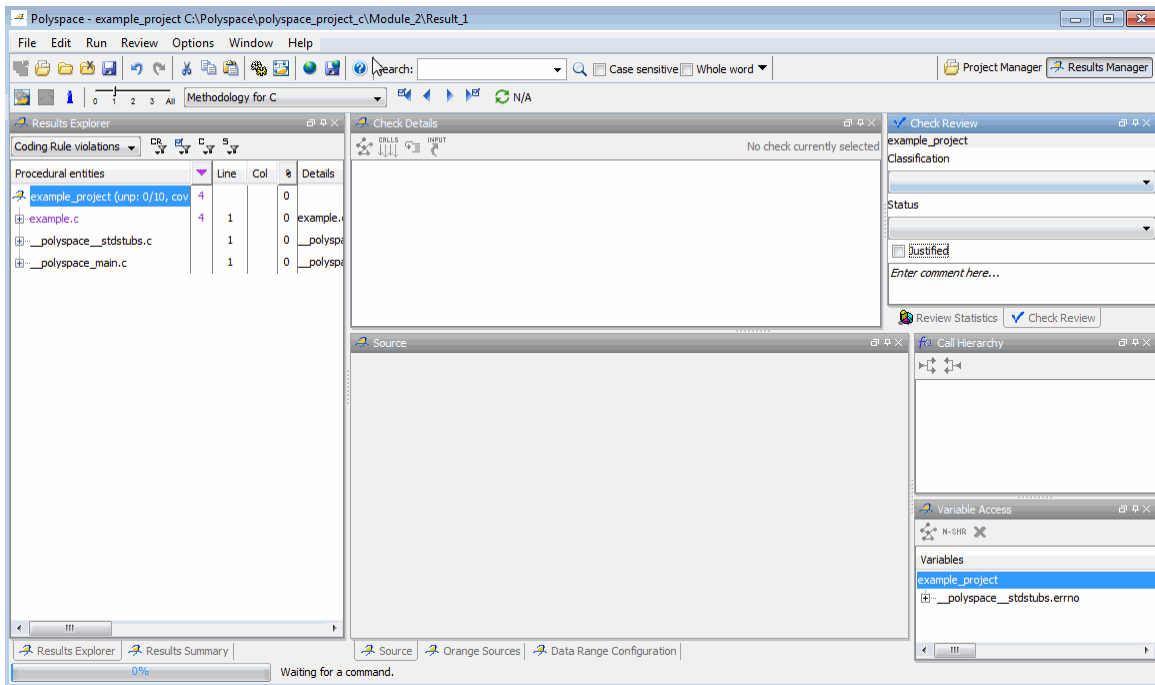
The verification fails because of MISRA C violations. You see messages in the **Full Log**, and the **Output Summary** indicates that the verification has detected MISRA errors. If a rule with state **Error** is violated, the verification stops.

3. MISRA and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

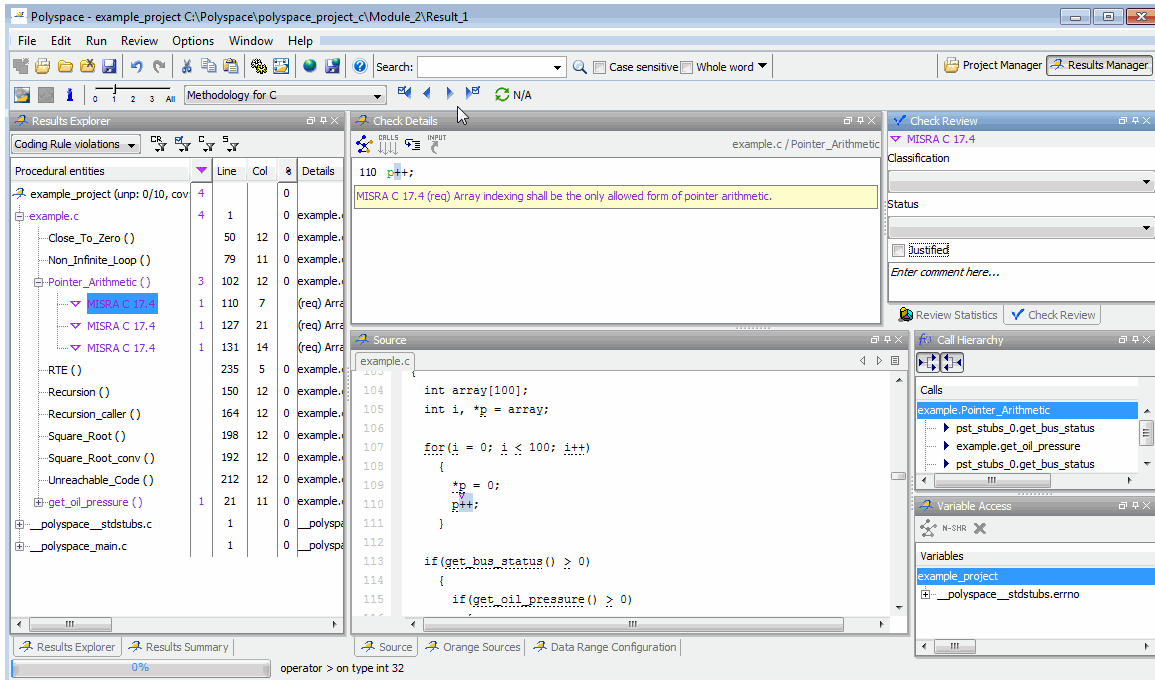
Examine MISRA C Violations

To examine the MISRA C violations:

- 1 In the Project Browser Result folder, double-click **MISRA-C-report.xml**, which opens the Results Manager perspective.
- 2 On the **Results Explorer** toolbar, from the drop-down list of the first filter, select **Coding Rule violations**.



- 3 Click any violation.

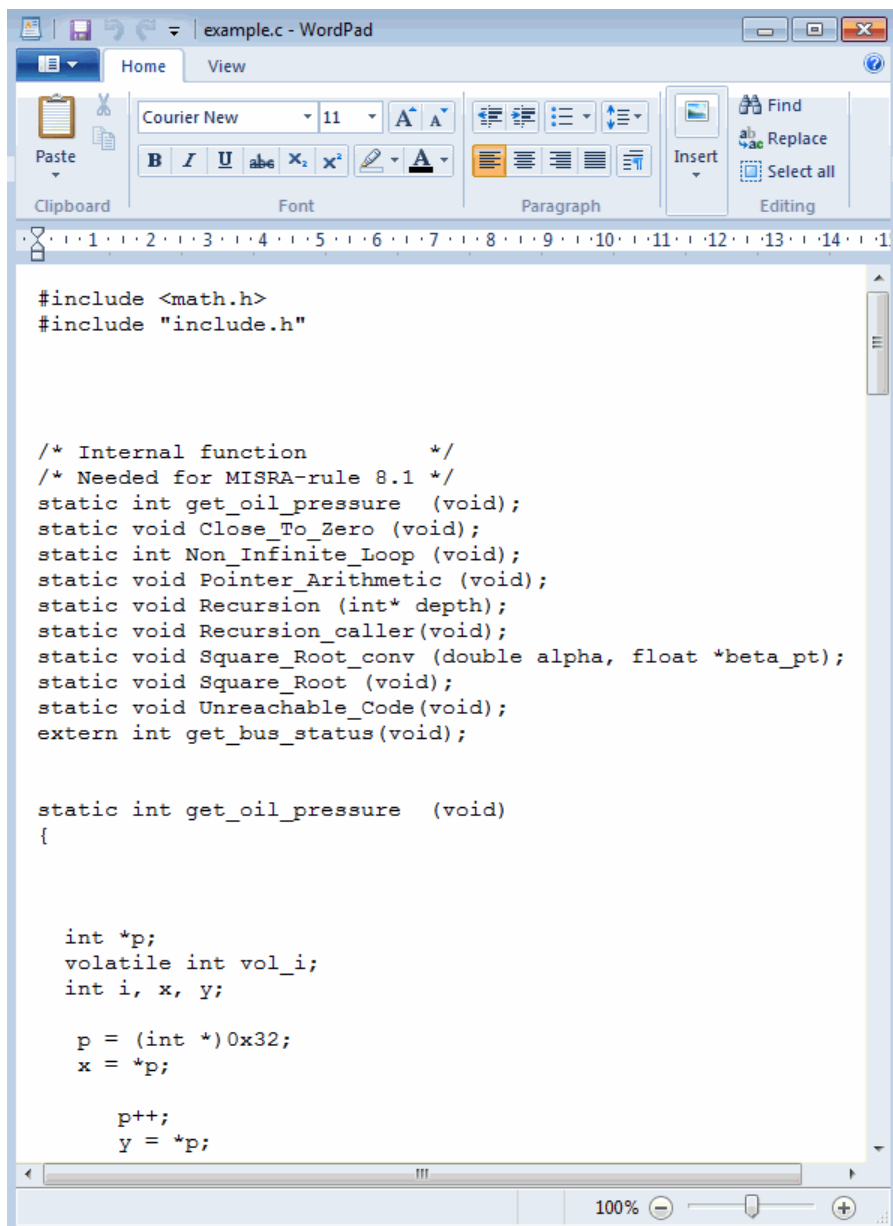


In the **Check Details** pane, you see a description of the violated rule and the name of the file in which the violation was found. In the **Source** pane, you see the source code that contains the violation.

The code uses a form of pointer arithmetic that is not allowed, a violation of rule 17.4.

- 4 In the **Source** pane, right-click the highlighted code containing the violation of rule 17.4. From the context menu, select **Open Source File**. Before you can open source files, you must configure a text editor. See “Configure Text and XML Editors” on page 5-9.

The `example.c` file opens in your text editor.



The image shows a screenshot of the WordPad application window titled "example.c - WordPad". The window displays a C program with several MISRA-style comments. The code is as follows:

```
#include <math.h>
#include "include.h"

/* Internal function          */
/* Needed for MISRA-rule 8.1 */
static int get_oil_pressure (void);
static void Close_To_Zero (void);
static int Non_Infinite_Loop (void);
static void Pointer_Arithmetic (void);
static void Recursion (int* depth);
static void Recursion_caller(void);
static void Square_Root_conv (double alpha, float *beta_pt);
static void Square_Root (void);
static void Unreachable_Code(void);
extern int get_bus_status(void);

static int get_oil_pressure (void)
{

    int *p;
    volatile int vol_i;
    int i, x, y;

    p = (int *)0x32;
    x = *p;

    p++;
    y = *p;
```


- 5 Fix the MISRA violation and run the verification again. The results are the same as those from the tutorial in “Run Verification” on page 3-2.

Open MISRA-C Report

After you check MISRA rules, you can generate a report containing all the errors and warnings reported by the MISRA-C checker. Before you can open a MISRA-C report, you must configure an editor. See “Configure Text and XML Editors” on page 5-9.

To view the MISRA-C report:

- 1 Navigate to the folder that contains your coding rules report, for example, `C:\Polyspace\polyspace_project\Module_3\Result_1\Polyspace-Doc`.
- 2 Double-click the coding rules report, for example, `example_project_CodingRules.rtf`. The report opens in your selected editor.

A

- analysis options 2-12
 - MISRA C compliance 5-10
- ANSI compliance 3-9
- AOT. *See* Automatic Orange Tester
- Automatic Orange Tester
 - overview 4-28

C

- call graph 4-12
- call tree view 4-4
- calling sequence 4-12
- cfg. *See* configuration file
- client 1-7 3-2
 - installation 1-12
 - verification on 3-20
- coding review progress view 4-4 4-12
- coding rules compliance 1-4
- color-coding of verification results 1-4 to 1-5 4-5
- compile log
 - Project Manager 3-11 3-22
 - Spooler 3-12
- compile phase 3-9
- compliance
 - ANSI 3-9
 - coding rules 1-4
 - MISRA C 5-1
- configuration file
 - definition 2-2

D

- default folder
 - changing location 2-6
- division by zero
 - example 4-18
- downloading
 - results 3-16

E

- expert mode
 - filters 4-20

F

- files
 - includes 2-9
 - source 2-9
- filters 4-20
- folders
 - includes 2-9
 - sources 2-9

H

- hardware requirements 3-18
- help
 - accessing 1-16

I

- installation
 - Polyspace Client for C/C++ 1-12
 - Polyspace products 1-12
 - Polyspace Server for C/C++ 1-12

L

- licenses
 - obtaining 1-12
- logs
 - compile
 - Project Manager 3-11 3-22
 - Spooler 3-12
 - full
 - Project Manager 3-11 3-22
 - Spooler 3-12
 - stats
 - Project Manager 3-11 3-22
 - Spooler 3-12

- viewing
 - Project Manager 3-11 3-22
 - Spooler 3-12

M

- manual mode
 - selection 4-9
 - use 4-7
- MISRA C compliance
 - analysis option 5-10
 - checking 5-1
 - file exclusion 5-14
 - rules file 5-11
 - violations 5-16

P

- Polyspace Client for C/C++
 - installation 1-12
 - license 1-12
- Polyspace products for C
 - installation 1-12
 - licenses 1-12
 - related products 1-17
 - workflow 1-13
- Polyspace products for C/C++
 - components 1-7
 - overview 1-4
 - user interface 1-7
- Polyspace Queue Manager Interface. *See* Spooler
- Polyspace Server for C/C++
 - installation 1-12
 - license 1-12
- Polyspace verification environment
 - opening 2-4
- preferences
 - Project Manager
 - default server mode 3-9
 - server detection 3-18

- procedural entities view 4-4
- product overview 1-4
- progress bar
 - Project Manager window 3-11 3-22
- project
 - creation 2-2 2-6
 - definition 2-2
 - file types
 - configuration file 2-2
 - folders
 - includes 2-3
 - results 2-3
 - sources 2-3
 - opening 3-3
 - saving 2-12
- Project Manager
 - monitoring verification progress 3-11 3-22
 - opening 2-4
 - overview 2-4
 - perspective 2-4
 - starting verification on client 3-20
 - starting verification on server 3-9
 - viewing logs 3-11 3-22
 - window
 - progress bar 3-11 3-22
- Project Manager perspective 1-7

R

- related products 1-17
 - Polyspace products for linking to Models 1-17
 - Polyspace products for verifying Ada code 1-17
- reports
 - generation 4-28
- results
 - downloading from server 3-16
 - opening 4-3
 - report generation 4-28
 - reviewing 4-1

Results Manager perspective 1-7
 call tree view 4-4
 coding review progress view 4-4
 opening 4-3
 overview 4-3
 procedural entities view 4-4
 selected check view 4-4
 source code view 4-4
 variables view 4-4
rte view. *See* procedural entities view

S

selected check view 4-4
server 1-7 3-2
 detection 3-18
 information in preferences 3-18
 installation 1-12 3-18
 verification on 3-9
source code view 4-4
Spooler 1-7
 monitoring verification progress 3-12
 removing verification from queue 3-16
 use 3-12
 viewing log 3-12

T

target environment 2-11
troubleshooting failed verification 3-17

U

unreachable code
 example 4-16

V

variables view 4-4
verification
 Ada code 1-17
 C/C++ code 1-4
 client 3-2
 compile phase 3-9
 failed 3-17
 monitoring progress
 Project Manager 3-11 3-22
 Spooler 3-12
 phases 3-9
 results
 color-coding 1-4 to 1-5
 opening 4-3
 report generation 4-28
 reviewing 4-1
 running 3-2
 running on client 3-20
 running on server 3-9
 starting
 from Project Manager 3-2 3-9 3-20
 stopping 3-24
 troubleshooting 3-17
 with MISRA C checking 5-15
Verification
 stopping 3-23

W

workflow
 basic 1-13
 in this guide 1-14