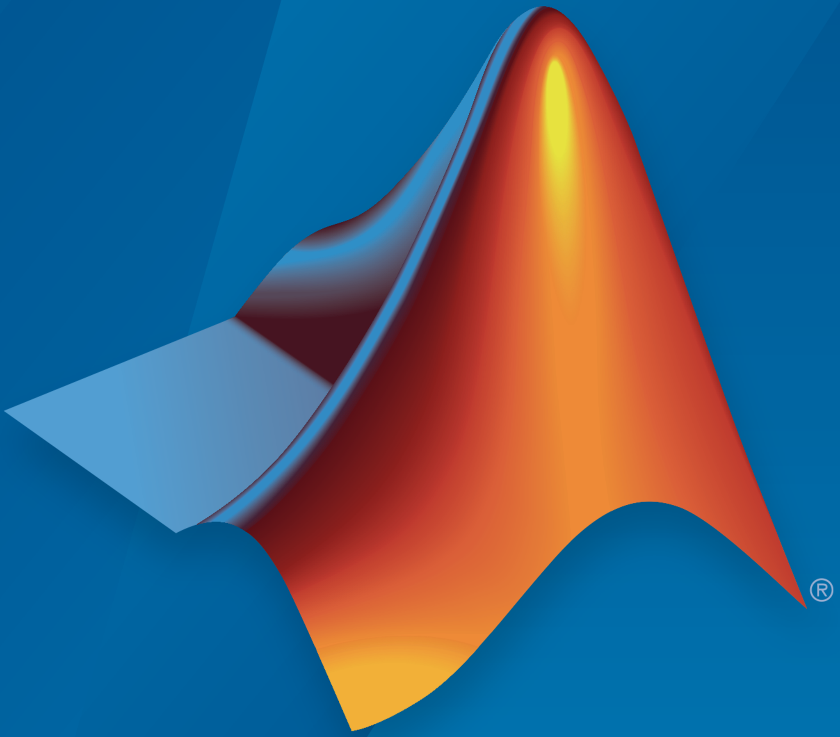
Polyspace® Bug Finder™

User's Guide



MATLAB®&SIMULINK®

R2015a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Polyspace® Bug Finder™ User's Guide

Revision History

September 2013	Online only	New for Version 1.0 (Release 2013b)
March 2014	Online Only	Revised for Version 1.1 (Release 2014a)
October 2014	Online only	Revised for Version 1.2 (Release 2014b)
March 2015	Online only	Revised for Version 1.3 (Release 2015a)

Contents

1 Project Configuration

6 Command-Line Analysis

7 Polyspace Bug Finder Analysis in Simulink

Project Configuration

What Is a Project?

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

- Source files
- Include folders
- A configuration, specifying a set of analysis options

In the Polyspace interface, use the Project Browser and Configuration panes to create and modify a project.

What is a Project Template?

A **Project Template** is a predefined set of analysis options for a specific compilation environment. When creating a new project, you have the option to:

- Use an existing template to automatically set analysis options for your compiler.

  Polyspace software provides predefined templates for common compilers such as `IAR`, `Kiel`, and `VxWorks` `Aonix`, `Rational`, and `Greenhills`. For additional templates, see Polyspace Compiler Templates .
- Set analysis options manually. You can save your options to a custom template and reuse them later. For more information, see “Save Analysis Options as Project Template”.

Open Polyspace Bug Finder

After you install MATLAB® and Polyspace, you can open Polyspace Bug Finder™ from the desktop shortcut created during installation. Other ways to open Polyspace are:

- via MATLAB.
  - In the apps gallery, select Polyspace Bug Finder.
  - In the Command Window, enter:

    ```
    polyspaceBugFinder
    ```
- via the command-line.
  - DOS: `*MATLAB Install*\polyspace\bin\polyspace-bug-finder`
  - UNIX: `*MATLAB Install*/polyspace/bin/polyspace-bug-finder`

  Where *`MATLAB Install`* is your MATLAB installation folder.

Polyspace Bug Finder can be opened simultaneously with Polyspace Code Prover™ or a second instance of Polyspace Bug Finder. However, only one code analysis can be run at a time.

If you try to run Polyspace processes from multiple windows, you will get a `License Error –4,0`. To avoid this error, close any additional Polyspace windows before running an analysis.

Create New Project

This example shows how to create a new project in Polyspace Bug Finder. Before you create a project, you must know:

- Location of source files
- Location of include files
- Location where analysis results will be stored

For the three locations, you will find it convenient to create three subfolders under a common project folder. For instance, under the folder `polyspace_project`, you can create three subfolders `sources`,`includes` and `results`.

1. Select **File** > **New Project**.
2. In the Project – Properties dialog box, enter the following information:
   - **Project name**
   - **Location**: Folder where you will store the project file with extension `.psprj`. You can use this file to open an existing project.

     The software assigns a default location to your project. You can change this default on the **Project and Results Folder** tab in the Polyspace Preferences dialog box.
   - **Project language**
3. Add source files and include folders to your project.
   - Navigate to the location where you stored your source files. Select the source files for your project. Click **Add Source Files**.
   - The software automatically adds the standard include files to your project. To use custom include files, navigate to the *folder* containing your include files. Click **Add Include Folders**.
4. Click **Finish**.

   The new project opens in the **Project Browser** pane. Your source files are automatically copied to the first module in the project.
5. Save the project. Select **File** > **Save** or enter **Ctrl+S**.

   To close the project at any time, in the **Project Browser**, right-click the project node and select **Close**.

Create Project Automatically

If you use build automation scripts to build your source code, you can automatically setup a Polyspace project from your scripts. The automatic project setup runs your automation scripts to determine:

- Source files.
- Includes.
- Target & compiler options. For more information on these options, see:
  - C Code: “Target & Compiler”
  - C++ Code: “Target & Compiler”

1 Select **File** > **New Project**.
2 On the Project – Properties dialog box, after specifying the project name, location and language, under **Project configuration**, select **Create from build command**.
3 On the next window, enter the following information:

Field	Description
Specify command used for building your source files	If you use an IDE such as Visual Studio® or Eclipse™ to build your project, specify the full path to your IDE executable or navigate to it using the 📁 button. For a tutorial using Visual Studio, see “Trace Visual Studio Build”. **Example:** `"C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VCExpress.exe"` If you use command-line tools to build your project, specify the appropriate command. **Example:** • `make -B -f` *`makefileName`* or `make -W` *`makefileName`* • `"mingw32-make.exe -B -f makefilename"`
Specify working directory for	Specify the folder from which you run your build automation script.

Field	Description
running build command	If you specify the full path to your executable in the previous field, this field is redundant. Specify any folder.
Add advanced configure options	Specify additional options for advanced tasks such as incremental build. For the full list of options, see the `-options value` argument for `polyspaceConfigure`.

4 Click Run.

- If you entered your build command, Polyspace runs the command and sets up a project.
- If you entered the path to an executable, the executable runs. Build your source code and close the executable. Polyspace traces your build and sets up a project.

  For example, in Visual Studio 2010, use **Tools** > **Rebuild Solution** to build your source code. Then close Visual Studio.

After you click **Finish**, the new project appears on the **Project Browser** pane. To close the project at any time, in the **Project Browser**, right-click the project node and select **Close**.

5 If you updated your build command, you can recreate the Polyspace project from the updated command. To recreate an existing project, on the **Project Browser**, right-click the project name and select **Update Project**.

Note:

- In the Polyspace interface, it is possible that the created project will not show implicit defines or includes. The configuration tool does include them. However, they are not visible.
- By default, Polyspace assigns the latest dialect for your compiler. If you have compilation errors in your project, check the dialect. If it does not apply to you, change it to a more appropriate one.
- If your build process requires user interaction, you cannot run the build command from the Polyspace user interface and do an automatic project setup.

Related Examples

- “Trace Visual Studio Build”

More About

- “Requirements for Project Creation from Build Systems”
- “Your Compiler Is Not Supported”

Requirements for Project Creation from Build Systems

For automatic project creation from build systems, your build commands or makefiles must meet the following requirements:

- Your compiler must be called locally.

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

  If your compiler performs only an incremental build, use appropriate options to build all your source files. For example, if you use `gmake`, append the `-B` or `-W` *`makefileName`* option to force a clean build. For the list of options allowed with the GNU® `make`, see make options.
- Your compiler configuration must be available to Polyspace. The compilers currently supported include the following:

  - Visual C++® compiler
  - `gcc`
  - `clang`
  - `MinGW` compiler
  - `IAR` compiler

  If your compiler configuration is not available to Polyspace:

  - Write a compiler configuration file for your compiler in a specific format. For more information, see “Your Compiler Is Not Supported”.
  - Contact MathWorks Technical Support. For more information, see “Contact Technical Support”.
- In Linux®, only UNIX® shell (sh) commands must be used. If your build uses advanced commands such as commands supported only by bash, tcsh or zsh, Polyspace cannot trace your build.

  In Windows®, only DOS commands must be used. If your build uses advanced commands such as commands supported only by Powershell, Polyspace cannot trace your build.

- Your build command must not use aliases.

  The `alias` command is used in Linux to create an alternate name for commands. If your build command uses those alternate names, Polyspace cannot recognize them.
- Your build command must be executable completely on the current machine and must not require privileges of another user.

  If your build uses `sudo` to change user privileges or `ssh` to remotely login to another machine, Polyspace cannot trace your build.
- If you use Cygwin™ to build your source code, Polyspace cannot trace your build. You can either use MinGW to build your source and have the software trace your build, or do the following:

  1. Build your source code using the process that you usually follow. Copy the command lines that executed during the build to a file.

     For instance, on `make` systems, use the flag `-B` or `-W` *`makefileName`* to build your entire source and `-n` to view the commands. For more information, see make options.
  2. Save the file as a Windows batch file. A batch file is a file that can contain one or more commands. It has a `.bat` extension. For more information, see batch files.
  3. Run the batch file to make sure your build commands work.

     For example, if your batch file is called `myBuild.bat`, at a DOS command prompt, enter:

     ```
     cmd.exe /C myBuild.bat
     ```
  4. Create a project from the batch file.

     If you ran the command in the previous step, at a DOS command prompt, enter:

     ```
     polyspace-configure cmd.exe /C myBuild.bat
     ```
- If your build command uses redirection with the `>` or `|` character, the redirection occurs after Polyspace traces the command. Therefore, Polyspace does not handle the redirection.

  For example, if your command occurs as

  ```
  command1 | command2
  ```
  And you enter

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

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

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

See Also

polyspaceConfigure

Related Examples

- “Create Project Automatically”

Your Compiler Is Not Supported

For automatic project creation from your build system, your compiler configuration must be available to Polyspace. For information on supported compilers, see "Requirements for Project Creation from Build Systems". If your compiler is not supported, you can write your own compiler configuration file to enable support.

Tip To quickly see if your compiler configuration file works, run the automatic project setup on a sample build that does not take much time to complete. After you have set up a project successfully with your compiler configuration file, you can use this file for larger builds.

---

1. Copy one of the existing configuration files from *`matlabroot`*`\polyspace\configure\compiler_configuration\`.
2. Save the file as *`my_compiler`*`.xml`. *`my_compiler`* can be any name that helps you identify the file.

   To edit the file, save it outside the installation folder. After you have finished editing, you must copy the file back to *`matlabroot`*`\polyspace\configure\compiler_configuration\`.
3. Edit the contents of the file to represent your compiler. Replace the entries between the XML elements with appropriate content.

   The following table lists the XML elements in the file with a description of what the content within the element represents.

XML Element	Content Description	Content Example for GNU C Compiler

`<compiler_names><name> ...`
 	
`</name><compiler_names>`
	Name of the compiler executable. This executable transforms your `.c` files into object files. You can add several binary names, each in a separate `<name>...</name>` element. `polyspaceConfigure` checks	• `gcc`
		• `gpp`

XML Element	Content Description	Content Example for GNU C Compiler
	for each of the provided names and uses the compiler name for which it finds a match. You must not specify the linker binary inside the `<name>...</name>` elements.	

`<include_options><opt> ...`
 	
`</opt></include_options>`
	The option that you use with your compiler to specify include folders.	`-I`

`<system_include_options>`
 	
`<opt> ... </opt>`
 	
`</system_include_options>`
	The option that you use with your compiler to specify system headers.	`-isystem`

`<preinclude_options><opt> ...`
 	
`</opt></preinclude_options>`
	The option that you use with your compiler to force inclusion of a file in the compiled object.	`-include`

`<define_options><opt> ...`
 	
`</opt></define_options>`
	The option that you use with your compiler to predefine a macro.	`-D`

`<undefine_options><opt> ...`
 	
`</opt></undefine_options>`
	The option that you use with your compiler to undo any previous definition of a macro.	`-U`

`<semantic_options><opt> ...`
 	
`</opt></semantic_options>`
	The options that you use to modify the compiler behavior. These options specify the language settings to which the code must conform.	• `-ansi`
		• `-std =C90`
		• `-std =c++11`
		• `-fun signed -char`

XML Element	Content Description	Content Example for GNU C Compiler
`<dialect> ... </dialect>`	The options that specify the Polyspace dialect used by your compiler. For the complete list of dialects, on the **Configuration** pane, select **Target & Compiler**.	`gnu4.7`
`<preprocess_options_list>` `<opt> ... </opt>` `</preprocess_options_list>`	The options that specify how your compiler generates a preprocessed file.	`-E`
`<src_extensions><ext> ...` `</ext></src_extensions>`	The file extensions for source files.	• `c` • `cpp` • `c++`
`<obj_extensions><ext> ...` `</ext></obj_extensions>`	The file extensions for object files.	
`<precompiled_header_extensions> ...` `</precompiled_header_extensions>`	The file extensions for precompiled headers (if available).	
`<polyspace_c_extra_options_list>` `<opt> ... </opt>` `</polyspace_c_extra_options_list>`	Additional options that will be added to your project configuration	To avoid compilation errors due to non-ANSI® extension keywords, enter `-D` *`keyword`*. For more information, see “Preprocessor definitions (C/C++)”.

XML Element	Content Description	Content Example for GNU C Compiler
`<polyspace_cpp_extra_options_list>` `<opt> ... </opt>` `</polyspace_cpp_extra_options_list>`	Additional options that will be added to your C++ project configuration	To avoid compilation errors due to non-ANSI extension keywords, enter `-D` *`keyword`*. For more information, see “Preprocessor definitions (C/C++)”.

4 After saving the edited XML file to *`matlabroot`*`\polyspace\configure\compiler_configuration\`, create a project automatically using your build command. For more information, see:

- “Create Project Automatically”
- “Create Project Automatically at Command Line”
- “Create Project Automatically from MATLAB Command Line”

Create Project Using Visual Studio Information

In this section...
"Trace Visual Studio Build" on page 1-17
"Import Visual Studio Project" on page 1-20

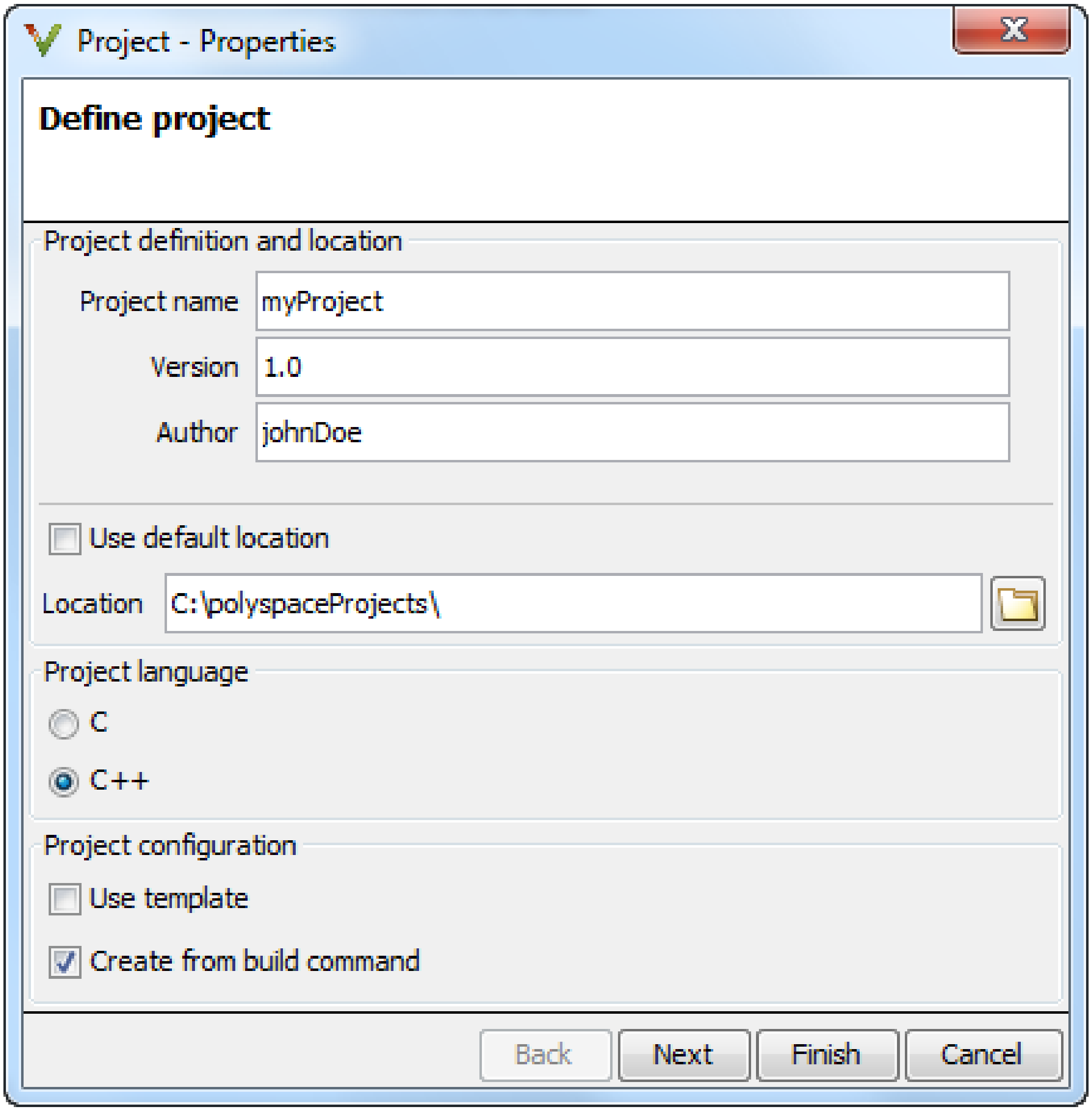
Trace Visual Studio Build

To create a Polyspace project, you can trace your Visual Studio build. For Polyspace to trace your Visual Studio build, you must install both `x86` and `x64` versions of the Visual C++ Redistributable for Visual Studio 2012 from the Microsoft website.

1. In the Polyspace interface, select **File** > **New Project**.
2. In the Project – Properties window, enter your project information.

   a. Choose **C++** as **Project Language**.

   b. Under **Project Configuration**, select **Create from build command** and click **Next**.


Project - Properties

Define project

Project definition and location

Project name: myProject

Version: 1.0

Author: johnDoe

☐ Use default location

Location: C:\polyspaceProjects\

Project language

○ C

● C++

Project configuration

☐ Use template

☑ Create from build command

3 In the field **Specify command used for building your source files**, enter the full path to the Visual Studio executable. For instance, "`C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\VCExpress.exe`".

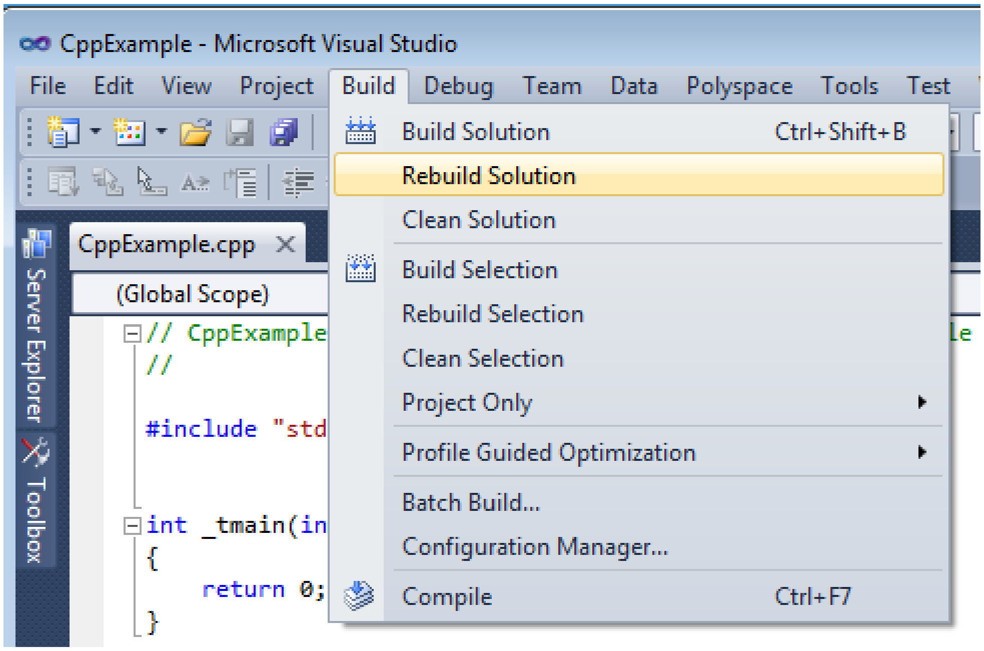
4 In the field **Specify working directory for running build command**, enter `C:\`.

Click Run.

This action opens the Visual Studio environment.

5 In the Visual Studio environment, create and build a Visual Studio project.

If you already have a Visual Studio project, open the existing project and build a clean solution. To build a clean solution in Visual Studio 2012, select **BUILD** > **Rebuild Solution**.



```
// CppExample
//

#include "std

int _tmain(in
{
    return 0;
}
```

6 After the project builds, close Visual Studio.

Polyspace traces your Visual Studio build and creates a Polyspace project.

The Polyspace project contains the source files from your Visual Studio build and the relevant **Target & Compiler** options.

7 If you update your Visual Studio project, to update the corresponding Polyspace project, on the **Project Browser**, right-click the project name and select **Update Project**.

Import Visual Studio Project

Note: This feature is will be removed in a future release.

You can directly create a Polyspace project from a Visual Studio project file with extension `.vcproj`. The Visual Studio import retrieves the following information from a Visual Studio project:

- **Source** files
- **Include** folders
- Some **Target & Compiler** options
- **Preprocessor Macros**

Note: For Visual Studio 2010 or Visual Studio 2012, you cannot directly import your project.

1. In the Polyspace interface, select **Tools** > **Import Visual Studio Project**.
2. In the Import Visual Studio dialog box, specify the **Visual Studio project** that you want to use.
3. You can:
   - **Create new Polyspace project**: Enter full path to a new Polyspace project.
   - **Update existing Polyspace project**: The dropdown list contains all projects currently open in the **Project Browser**. Select the project you want to update.
4. Click **Import**.

More About

- “Troubleshooting Project Creation from Visual Studio Build”

Troubleshooting Project Creation from Visual Studio Build

In this section...
"Cannot Create Project from Visual Studio Build" on page 1-21
"Compilation Error After Creating Project from Visual Studio Build" on page 1-21

Cannot Create Project from Visual Studio Build

If you are trying to import a Visual Studio 2010 or Visual Studio 2012 project and `polyspace-configure` does not work properly, do the following:

1. Stop the `MSBuild.exe` process.
2. Set the environment variable `MSBUILDDISABLENODEREUSE` to 1.
3. Specify `MSBuild.exe` with the`/nodereuse:false` option.
4. Restart the Polyspace configuration tool:

   ```
   polyspace-configure.exe -lang cpp <MSVS path>/msbuild sample.sln
   ```

Compilation Error After Creating Project from Visual Studio Build

If you automatically set up your project from a Visual Studio 2010 build, you can face compilation errors. By default, Polyspace assigns the latest dialect `visual11.0` to your project. This assignment can cause compilation errors. For more information on the **Dialect** option, see istian.

To avoid the errors, do one of the following:

- After automatic project setup:

  1. Open the project in the user interface. On the **Configuration** pane, select **Target & Compiler**.
  2. Check the **Dialect**. If it is set to `visual11.0`, change it to `visual10`.

  Note: If you are creating an options file from your Visual Studio 2010 build, check the `-dialect` argument. If it is set to `visual11.0`, change it to `visual10`.

- Before automatic project setup:

1. Open the file `cl.xml` in *`matlabroot`*`\polyspace\configure\compiler_configuration\` where *`matlabroot`* is your MATLAB installation folder such as `C:\Program Files\R2015a`.
2. Change the line

   ```
   <dialect>visual11.0</dialect>
   ```

   to

   ```
   <dialect>visual10</dialect>
   ```
3. Add the following lines:

   ```
   <polyspace_cpp_extra_options_list>
   <opt>-OS-target Visual</opt>
   </polyspace_cpp_extra_options_list>
   ```
4. Create your project or options file. The dialect is already assigned to `visual10`.

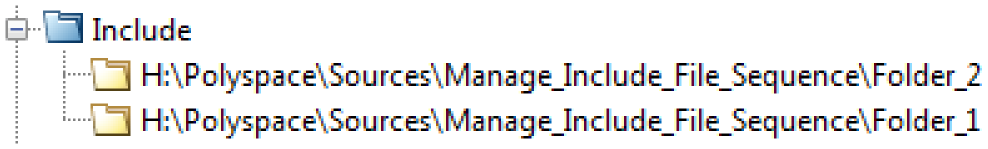
Add Source Files and Include Folders

This example shows how to add source files and include folders to an existing project.

Manage Include File Sequence

You can change the order of include folders to manage the sequence in which include files are compiled. When multiple include files by the same name exist in different folders, it is convenient to change the order of include folders instead of reorganizing the contents of your folders. For a particular include file name, the software includes the file in the first include folder under ***`Project_Name`*** > **Include**.

In the following figure, `Folder_1` and `Folder_2` contain the same include file `include.h`. If your source code includes this header file, during compilation, `Folder_2/include.h` is included in preference to `Folder_1/include.h`.



```
Include
    H:\Polyspace\Sources\Manage_Include_File_Sequence\Folder_2
    H:\Polyspace\Sources\Manage_Include_File_Sequence\Folder_1
```

To change the order of include folders:

1. In the **Project Browser**, expand the **Include** folder.
2. Select the include folder that you want to move.
3. To move the folder, click either ⇧ or ⇩ on the Project Browser toolbar.

Related Examples

- "Specify Results Folder"
- "Create New Project"

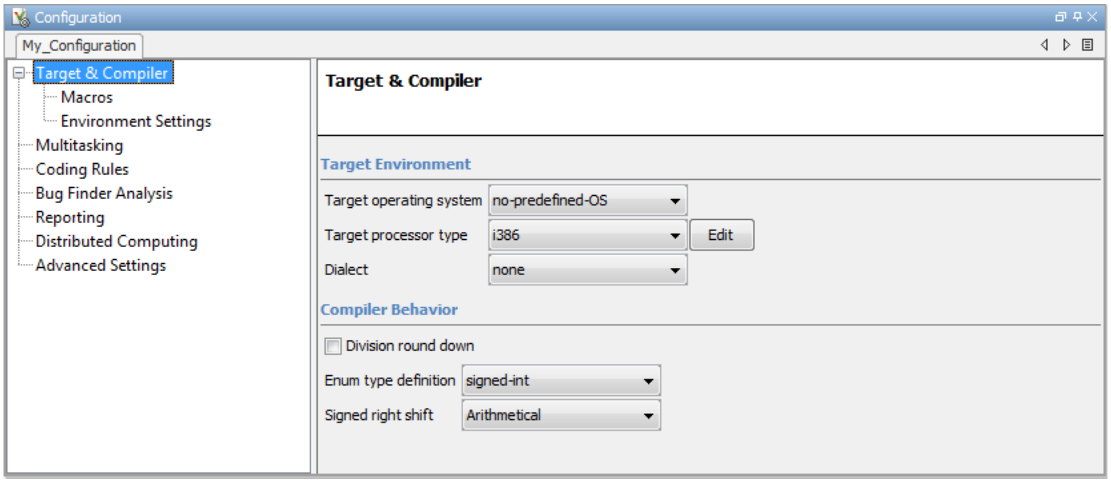
Specify Analysis Options

You can either retain the default analysis options used by the software or change them to your requirements.

In this section...
"Specify Options in User Interface" on page 1-24
"Specify Options from DOS and UNIX Command Line" on page 1-25
"Specify Options from MATLAB Command Line" on page 1-25

Specify Options in User Interface

To specify analysis options, use the different nodes on the **Configuration** pane.



Target operating system: no-predefined-OS
Target processor type: i386
Dialect: none
Division round down: ☐
Enum type definition: signed-int
Signed right shift: Arithmetical

For instance:

- To specify the target processor, select **Target & Compiler** in the **Configuration** tree view. Select your processor from the **Target processor type** drop-down list.
- To check for violation of MISRA C® rules, select **Coding Rules**. Check the **Check MISRA C Rules** box. To check for a subset of rules, select an option from the drop-down list.

Specify Options from DOS and UNIX Command Line

At the DOS or UNIX command-line, append analysis options to the `polyspace-bug-finder-nodesktop` command. For instance:

- To specify the target processor, use the `-target` option. For instance, to specify the `m68k` processor for your source file `file.c`, use the command:

  ```
  polyspace-bug-finder-nodesktop -sources "file.c" -lang c -target m68k
  ```
- To check for violation of MISRA C rules, use the `-misra2` option. For instance, to check for only the required MISRA C rules on your source file `file.c`, use the command:

  ```
  polyspace-bug-finder-nodesktop -sources "file.c" -misra2 required-rules
  ```

Specify Options from MATLAB Command Line

At the MATLAB command-line, enter analysis options and their values as string arguments to the `polyspaceBugFinder` function. For instance:

- To specify the target processor, use the `-target` option. For instance, to specify the `m68k` processor for your source file `file.c`, enter:

  ```
  polyspaceBugFinder('-sources','file.c','-lang','c','-target','m68k')
  ```
- To check for violation of MISRA C rules, use the `-misra2` option. For instance, to check for only the required MISRA C rules on your source file `file.c`, enter:

  ```
  polyspaceBugFinder('-sources','file.c','-misra2','required-rules')
  ```

See Also

polyspaceBugFinder

Related Examples

- "Save Analysis Options as Project Template"

More About

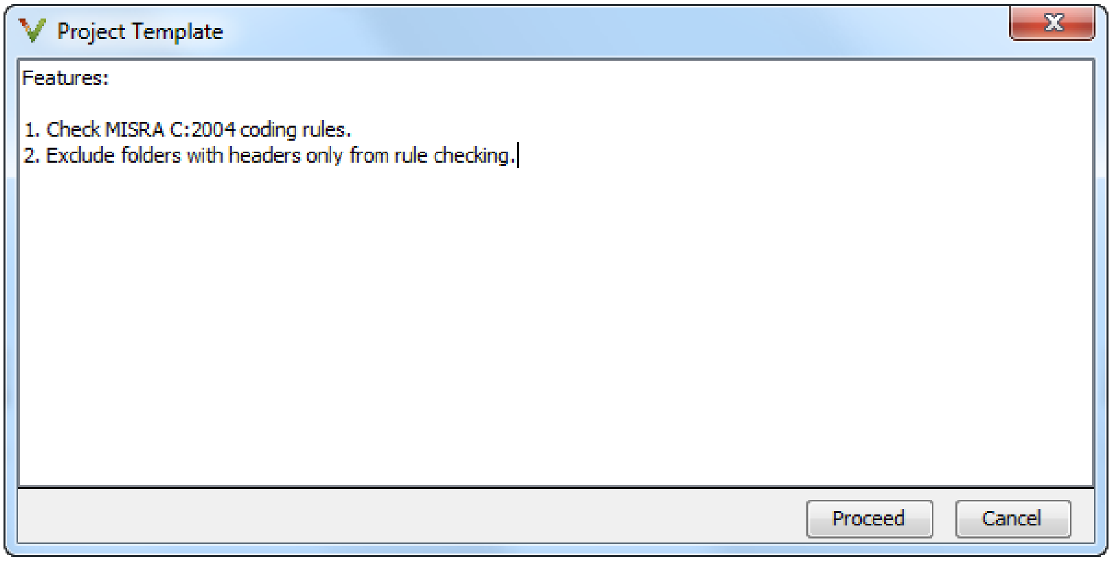
- "Analysis Options for C"
- "Analysis Options for C++"

Save Analysis Options as Project Template

This example shows how to save your analysis options for use in other projects. Once you have configured analysis options for a project, you can save the configuration as a **Project Template**. You can use this saved configuration to automatically set up analysis options for other projects.

- To create a **Project Template** from an open project:

  1. Right-click the configuration that you want to use, and then select **Save As Template**.
  2. Enter a description for the template, then click **Proceed**. Save your Template file.

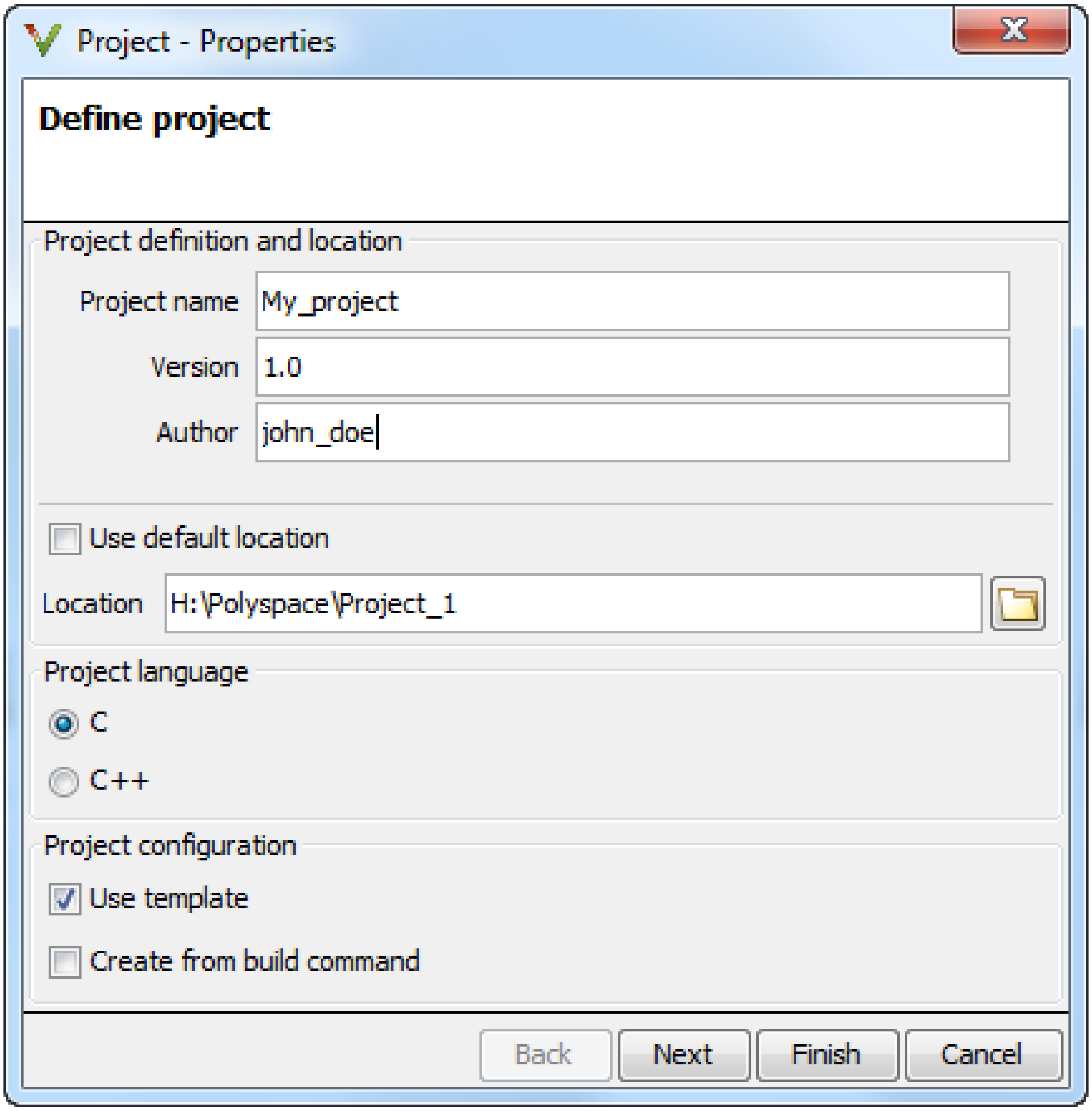


Features:

1. Check MISRA C:2004 coding rules.
2. Exclude folders with headers only from rule checking.

- When you create a new project, to use a saved template:

  1. Under **Project configuration**, check the **Use template** box. Click **Next**.



Define project

Project definition and location

Project name: My_project

Version: 1.0

Author: john_doe

☐ Use default location

Location: H:\Polyspace\Project_1

Project language

● C

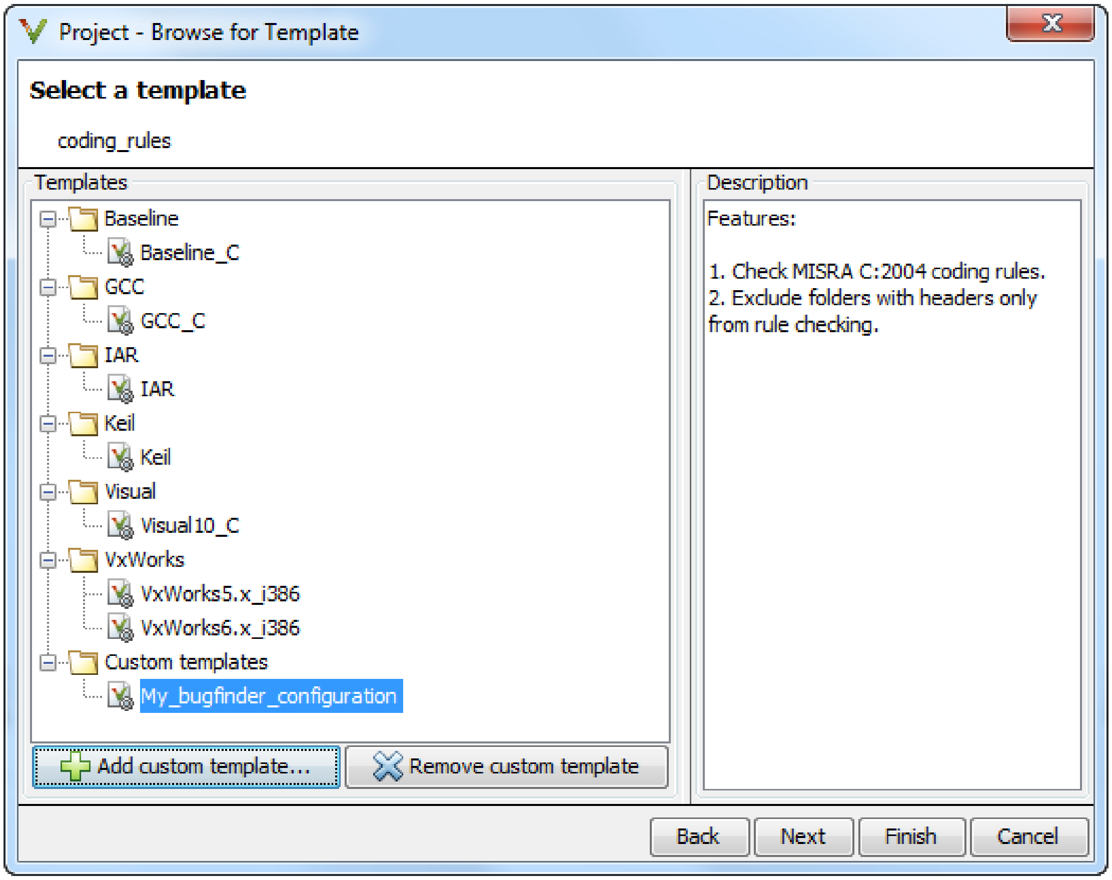
○ C++

Project configuration

☑ Use template

☐ Create from build command

2 Select Add custom template... . Navigate to the template that you saved earlier, and then click **Open**. The new template appears in the **Custom templates** folder on the **Templates** browser. Select the template for use.



Project - Browse for Template

Select a template

coding_rules

Templates

- Baseline
  - Baseline_C
- GCC
  - GCC_C
- IAR
  - IAR
- Keil
  - Keil
- Visual
  - Visual10_C
- VxWorks
  - VxWorks5.x_i386
  - VxWorks6.x_i386
- Custom templates
  - My_bugfinder_configuration

Description

Features:

1. Check MISRA C:2004 coding rules.
2. Exclude folders with headers only from rule checking.

Related Examples

- "Specify Analysis Options"

More About

- "Analysis Options for C"

- “Analysis Options for C++”

Specify External Text Editor

This example shows how to change the default text editor for opening source files from the Polyspace interface. By default, if you open your source file from the user interface, it opens on a **Code Editor** tab. If you prefer editing your source files in an external editor, you can change this default behavior.

1 Select **Tools** > **Preferences**.
2 On the Polyspace Preferences dialog box, select the **Editors** tab.
3 From the **Text editor** drop-down list, select **External**.
4 In the **Text editor** field, specify the path to your text editor. For example:

   ```
   C:\Program Files\Windows NT\Accessories\wordpad.exe
   ```
5 To make sure that your source code opens at the correct line and column in your text editor, specify command-line arguments for the editor using Polyspace macros, `$FILE`, `$LINE` and `$COLUMN`. Once you specify the arguments, when you right-click a check on the **Results Summary** pane and select **Open Editor**, your source code opens at the location of the check.

   Polyspace has already specified the command-line arguments for the following editors:

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

   If you are using one of these editors, select it from the **Arguments** drop-down list. If you are using another text editor, select `Custom` from the drop-down list, and enter the command-line options in the field provided.
6 To revert back to the built-in editor, on the **Editors** tab, from the **Text editor** drop-down list, select **Built In**.

For console-based text editors, you must create a terminal. For example, to specify `vi`:

1 In the **Text Editor** field, enter `/usr/bin/xterm`.

2. From the **Arguments** drop-down list, select `Custom`.
3. In the field to the right, enter `-e /usr/bin/vi $FILE`.

Change Default Font Size

This example shows how to change the default font size in the Polyspace user interface.

1 Select **Tools** > **Preferences**.

2 On the **Miscellaneous** tab:

   - To increase the font size of labels on the user interface, select a value for **GUI font size**.

     For example, to increase the default size by 1 point, select `+1`.
   - To increase the font size of the code on the **Source** pane and the **Code Editor** pane, select a value for **Source code font size**.

3 Click **OK**.

When you restart Polyspace, you see the increased font size.

Define Custom Review Status

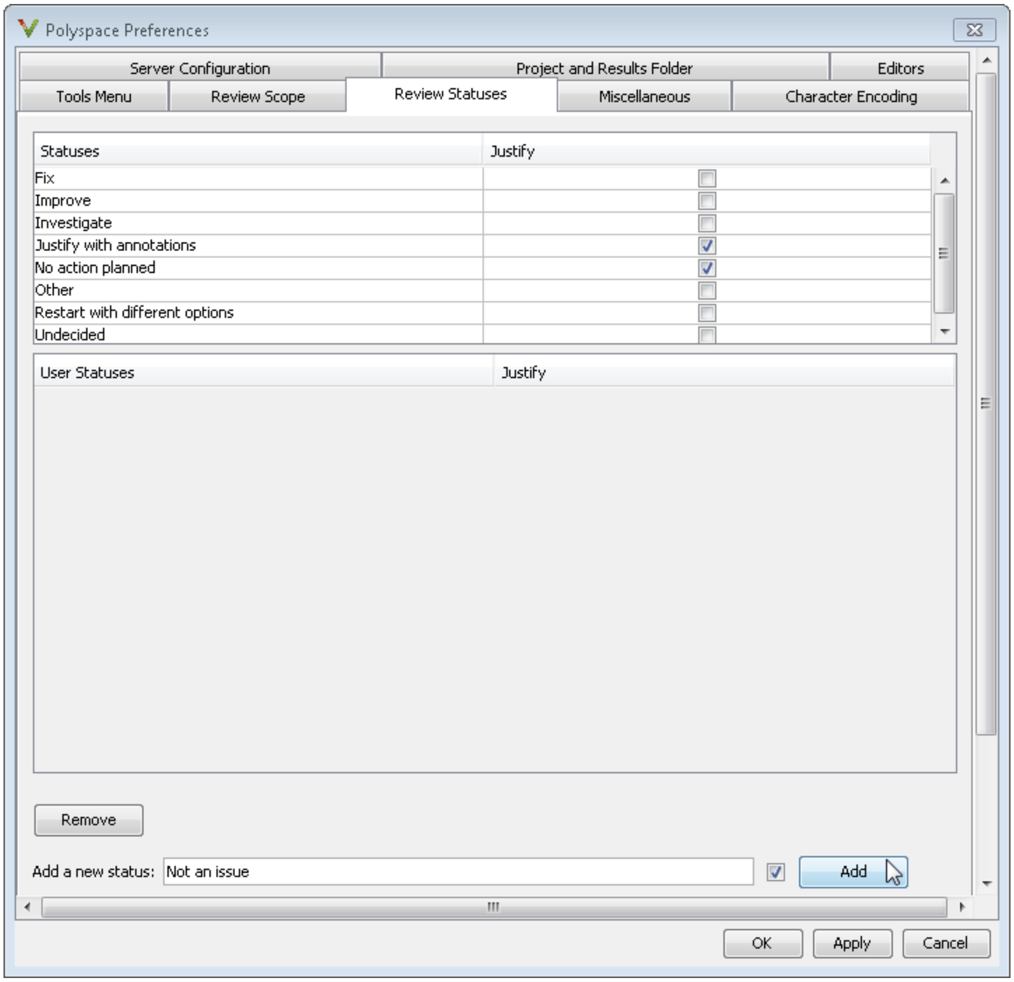
This example shows how to customize the statuses you assign on the **Results Summary** pane.

Define Custom Status

1 Select **Tools** > **Preferences**.

2 Select the **Review Statuses** tab.

3 Enter your new status at the bottom of the dialog box, then click **Add**.



Statuses	Justify
Fix	☐
Improve	☐
Investigate	☐
Justify with annotations	✓
No action planned	✓
Other	☐
Restart with different options	☐
Undecided	☐

User Statuses	Justify

Add a new status: Not an issue ✓

The new status appears in the **Status** list.

4 Click **OK** to save your changes and close the dialog box.

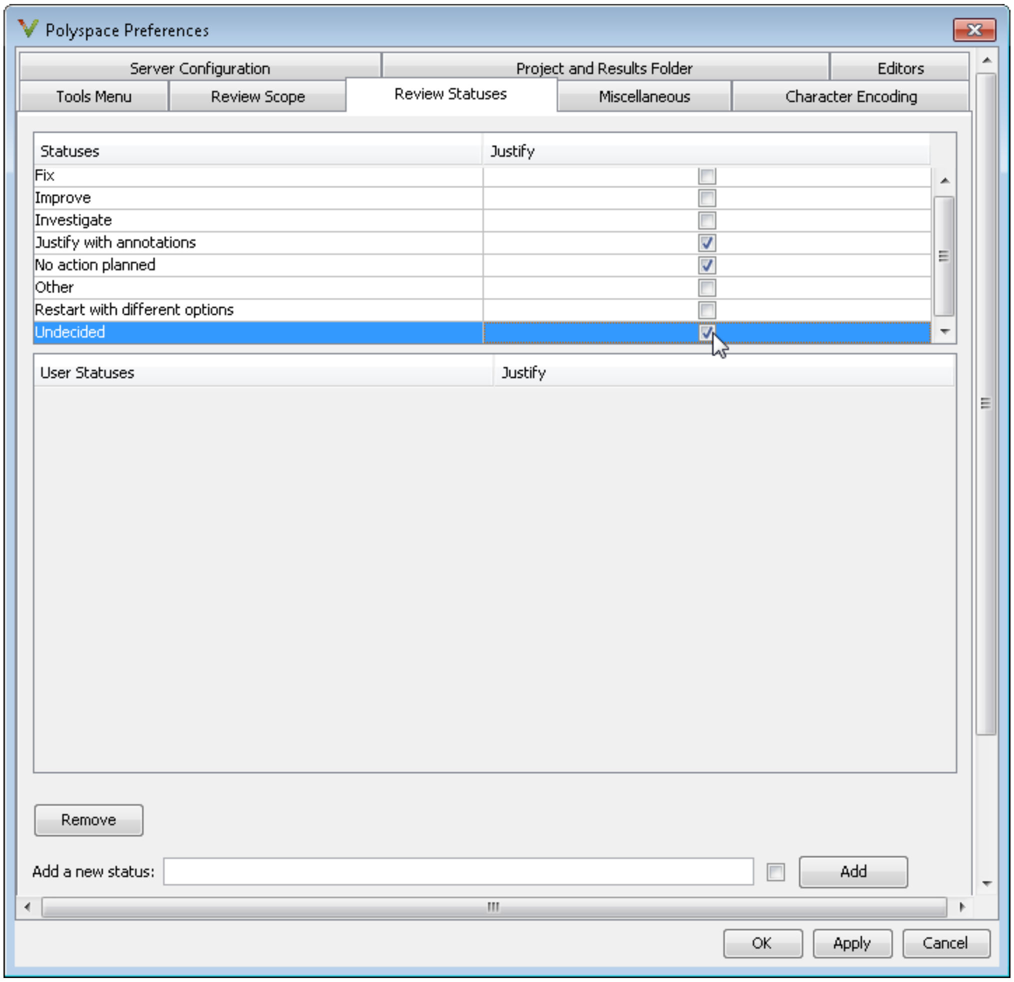
When reviewing checks, you can select the new status from the **Status** drop-down list on the **Results Summary** pane.

Add Justification to Existing Status

By default, a check is automatically justified if you assign the status, `Justified` or `No action planned`. However, you can change this default setting so that a check is justified when you assign one of the other existing statuses.

To add justification to existing status `Improve`:

1. Select **Tools** > **Preferences**.
2. Select the **Review Statuses** tab. For the `Improve` status, select the check box in the **Justify** column. Click **OK**.



Statuses	Justify
Fix	☐
Improve	☐
Investigate	☐
Justify with annotations	✓
No action planned	✓
Other	☐
Restart with different options	☐
Undecided	✓

User Statuses	Justify

If you assign the `Improve` status to a check on the **Results Summary** pane, the check gets automatically justified.

Compilation Errors

During a Polyspace Bug Finder analysis, the software first compiles the project and looks for coding rule errors. If the files have compilation errors, a message appears in the Output Summary pane and the offending files are ignored during the later analysis stages.

Consequently, Bug Finder produces results for all source files that do not have compilation errors. Files with compilation problems do not appear in the results.

For complete analysis results, fix compilation errors before rerunning the analysis.

Set Up Multitasking Analysis

This example shows how to prepare for analysis of multitasking code. If your code has functions that are intended for concurrent execution, you must specify them before analysis. If these functions operate on a common variable, you must also specify protection mechanisms for those operations. You can then use Polyspace Bug Finder to check if the protection mechanisms are well designed. For this example, save the following code in a file `multi.c`:

```
int a;

begin_critical_section();
end_critical_section();

void performTaskCycle(void) {
 begin_critical_section();
 a++;
 end_critical_section();
}

void task1(void) {
  while(1) {
    performTaskCycle();
   }
}

void task2(void) {
   while(1) {
    performTaskCycle();
   }
}

void task3() {
  a=0;
}
```

Specify Entry Points

If you want `task1`, `task2`, and `task3` to run concurrently:

- In the user interface, do the following:

  1 On the **Configuration** pane, select the **Multitasking** node.

2. For **Entry points**, specify `task1`, `task2`, and `task3`, each on its own line.

- At the DOS or UNIX command prompt, use the option `-entry-points` with the `polyspace-bug-finder-nodesktop` command. For example:

```
polyspace-bug-finder-nodesktop -sources multi.c
                -entry-points task1,task2,task3
```

- At the MATLAB command prompt, specify the following arguments to the `polyspaceBugFinder` function.

```
polyspaceBugFinder('-sources','multi.c',...
        '-entry-points','task1,task2,task3')
```

Specify Critical Sections

If you do not want the operation `a++` from `task1` to interrupt the same operation from `task2`, you can place the operation inside a critical section. Polyspace requires that a critical section must lie between two function calls. The functions that begin and end the critical section must have the prototype `void func(void)`. In this example, to specify `begin_critical_section` and `end_critical_section` as the required functions:

- In the user interface, do the following:

  1. On the **Configuration** pane, select the **Multitasking** node.
  2. For **Critical section details**, specify `begin_critical_section` as **Starting procedure** and `end_critical_section` as **Ending procedure**.

- At the DOS or UNIX command prompt, use the options `-critical-section-begin` and `-critical-section-end` with the `polyspace-bug-finder-nodesktop` command. For example:

```
polyspace-bug-finder-nodesktop -sources multi.c
   -entry-points task1,task2,task3
   -critical-section-begin begin_critical_section:cs1
   -critical-section-end end_critical_section:cs1
```

- At the MATLAB command prompt, specify the following arguments to the `polyspaceBugFinder` function.

```
polyspaceBugFinder('-sources','multi.c',...
    '-entry-points','task1,task2,task3',...
    '-critical-section-begin','begin_critical_section:cs1',...
    '-critical-section-end','end_critical_section:cs1')
```

Specify Temporally Exclusive Tasks

To specify that `task3` must not interrupt `task1` or `task2`:

- In the user interface, do the following:

  1. On the **Configuration** pane, select the **Multitasking** node.
  2. For **Temporally exclusive tasks**, specify `task1 task3` and `task2 task3`, each on its own line.

- At the DOS or UNIX command prompt, do the following:

  1. In a text file, enter the following lines:

     ```
     task1 task3
     task2 task3
     ```

  2. Save the file as `tasklist.txt`.
  3. At the command prompt, use the option `-temporal-exclusions-file` with the `polyspace-bug-finder-nodesktop` command. For example:

     ```
     polyspace-bug-finder-nodesktop -sources multi.c
        -entry-points task1,task2,task3
        -critical-section-begin begin_critical_section:cs1
        -critical-section-end end_critical_section:cs1
        -temporal-exclusions-file tasklist.txt
     ```

- At the MATLAB command prompt, do the following.

  1. In a text file, enter the following lines:

     ```
     task1 task3
     task2 task3
     ```

  2. Save the file as `tasklist.txt`.
  3. Specify the following arguments to the `polyspaceBugFinder` function.

     ```
     polyspaceBugFinder('-sources','multi.c',...
         '-entry-points','task1,task2,task3',...
         '-critical-section-begin','begin_critical_section:cs1',...
         '-critical-section-end','end_critical_section:cs1',...
         '-temporal-exclusions-file','tasklist.txt')
     ```

Specify Concurrency Defects to Check

To specify that Polyspace Bug Finder must check for **Data race** and **Deadlock** errors:

- In the user interface, do the following.

  1. On the **Configuration** pane, select the **Bug Finder Analysis** node.
  2. From the **Find defects** list, select `custom`.
  3. Under the **Concurrency** node, select **Data race** and **Deadlock**.
- At the DOS or UNIX command prompt, use the option `-checkers` with the `polyspace-bug-finder-nodesktop` command. For example:

  ```
  polyspace-bug-finder-nodesktop -sources multi.c
     -entry-points task1,task2,task3
     -critical-section-begin begin_critical_section:cs1
     -critical-section-end end_critical_section:cs1
     -temporal-exclusions-file tasklist.txt
     -checkers data_race,deadlock
  ```
- At the MATLAB command prompt, specify the following arguments to the `polyspaceBugFinder` function.

  ```
  polyspaceBugFinder('-sources','multi.c',...
      '-entry-points','task1,task2,task3',...
      '-critical-section-begin','begin_critical_section:cs1',...
      '-critical-section-end','end_critical_section:cs1',...
      '-temporal-exclusions-file','tasklist.txt',...
      '-checkers','data_race,deadlock')
  ```

See Also

"Entry points (C/C++)" | "Critical section details (C/C++)" | "Temporally exclusive tasks (C/C++)" | "Find defects (C/C++)"

Related Examples

- "Review Concurrency Defects"

More About

- "Concurrency"

Review Concurrency Defects

This example shows how to review defects that arise only in a multitasking analysis. For this example, use the results in the demo **Bug_Finder_Example.psprj**.

To load the demo in your **Project Browser**, under **Help**, select **Examples** > **Bug_Finder_Example.psprj**.

Filter Concurrency Defects

1. Right-click any column header and select **Category**.
2. On the **Category** column, select the icon.
3. From the filter menu, clear **All**. Select **Concurrency**.

Review Data Race Defects

1. Select the first **Data race** defect.

   The **Check Details** pane lists the variable `bad_glob1` that is:

   - Shared between multiple tasks and written in at least one of the tasks
   - Not protected against concurrent operations

   On the **Source** pane, the variable declaration appears highlighted.
2. To navigate to each operation involving `bad_glob1` in the source code, on the **Check Details** pane, click the row corresponding to the operation in the table. The lines with the operations are also highlighted in blue on the **Source** pane.

   a. To see if the access is in a critical section, use the **Access Protections** column. If one of the accesses is in a critical section, to fix the **Data race** defect, you can use the same critical section for the other accesses.

   b. To see which function contains the access, use the **Scope** column.
3. Select the second **Data race** defect.

   The **Check Details** pane lists the variable `bad_glob2` involved in the defect. You can view similar information as the first **Data race** defect.

   However, for this defect, the **Access** column on the **Check Details** pane lists why the operation can be non-atomic.

Review Locking Defects

1 Select the **Deadlock** defect.

The **Check Details** pane lists the sequence of operations that cause the **Deadlock**. You can see:

- The function call through which each task involved in the **Deadlock** enters a critical section.
- The function call through which each task attempts to enter a critical section that is already entered by another task.

2 To navigate to each operation in the source code, on the **Check Details** pane, click the row corresponding to the operation in the table.

3 Select the **Double lock** defect.

The **Check Details** pane lists the sequence of operations that cause the **Double lock**. You can see:

- The function call through which a task enters a critical section.
- The function call through which the task attempts to enter the same critical section.

4 To navigate to each operation in the source code, on the **Check Details** pane, click the row corresponding to the operation in the table.

5 Select the **Missing unlock** defect.

- The **Source** pane shows the function call that begins a critical section.
- On the **Check Details** pane, under the **Event** column, you can see which task contains the critical section.

See Also

Data race including atomic operations | Data race | Deadlock | Double lock | Double unlock | Missing lock | Missing unlock

Related Examples

- “Set Up Multitasking Analysis”

More About

- “Concurrency”

Annotate Code for Known Defects

How to Add Annotations

You can place annotations in your code that inform Polyspace software of known or acceptable defects. Through the use of these annotations, you can:

- Identify results from previous analyses.
- Categorize reviewed results.
- Highlight defects that are not significant.

You can add annotations in one of the following ways:

- When you are reviewing results in the Polyspace user interface, you can:
  1. Enter a **Classification**, **Status** and **Comment** for each defect on the **Results Summary** pane.
  2. Copy the information you entered and paste it in your source code in a syntax that Polyspace can read later. For more information, see “Copy and Paste Annotations”.
- You can directly open your source file in a text editor and enter comments in a syntax that Polyspace can read later. For more information, see “Syntax for Code Annotations”.

After you have placed the annotations in your code:

- Polyspace populates the **Status**, **Classification** and **Comment** fields for the defect.
- You or another reviewer can avoid reviewing the defect. You can either ignore the known defects or filter them from the **Results Summary** pane. For more information on filtering, see “Filter and Group Results”.

Syntax for Code Annotations

Polyspace applies the annotations, which are case-insensitive, to the first non-commented line of C code that follows the annotation.

To apply annotations to a single line of code, use the following syntax:

```
/* polyspace<Defect:Kind1[,Kind2] : [Classification] : [Status] >
[Additional comments] */
```

To apply annotations to a section of code, use the following syntax:

```
/* polyspace:begin<Defect:Kind1[,Kind1] : [Classification] : [Status] >
[Additional text] */

... Code section ...

/* polyspace:end<Defect:Kind1[,Kind1] : [Classification] : [Status] >  */
```

Square brackets *[ ]* indicate optional information.

Replace	Replace with
Kind1,Kind2,...	Specific defect abbreviations such as `MEM_LEAK`, `FREED_PTR`, etc. If you want the comment to apply to all defects on the following line, specify `ALL`.
Classification	• `Unset` • `High` • `Medium` • `Low` • `Not a defect`
Status	Action for correcting the defect in your code. Possible values are: • `Fix` • `Improve` • `Investigate` • `Justified` • `No action planned` • `Other`
Additional text	Additional comments.

Syntax Examples:

Defect:

```
polyspace<Defect:USELESS_WRITE : Low : No Action Planned > Known issue
```

Annotate Code for Rule Violations

How to Add Annotations

You can place annotations in your code that inform Polyspace software of known or acceptable coding rule violations. Through the use of these annotations, you can:

- Identify results from previous analyses.
- Categorize reviewed results.
- Highlight rule violations that are not significant.

Note: Source code annotations do not apply to code comments. Therefore, the following coding rules cannot be annotated:

- MISRA-C Rules 2.2 and 2.3
- MISRA-C++ Rule 2-7-1
- JSF++ Rules 127 and 133

You can add annotations in one of the following ways:

- When you are reviewing results in the Polyspace user interface, you can:
  1. Enter a **Classification**, **Status** and **Comment** for each coding-rule violation on the **Results Summary** pane.
  2. Copy the information you entered and paste it in your source code in a syntax that Polyspace can read later. For more information, see “Copy and Paste Annotations”.
- You can directly open your source file in a text editor and enter comments in a syntax that Polyspace can read later. For more information, see “Syntax for Code Annotations”.

After you have placed the annotations in your code:

- Polyspace populates the **Status**, **Classification** and **Comment** fields for the coding-rule violation.

- You or another reviewer can avoid reviewing the rule violation. You can either ignore the known rule violations or filter them from the **Results Summary** pane. For more information on filtering, see "Filter and Group Coding Rule Violations".

Syntax for Code Annotations

Polyspace applies the annotations, which are case-insensitive, to the first non-commented line of C code that follows the annotation.

To apply annotations to a single line of code, use the following syntax:

```
/* polyspace<Rule_set:Rule1[,Rule2] : [Classification] : [Status] >
[Additional comments] */
```

To apply annotations to a section of code, use the following syntax:

```
/* polyspace:begin<Rule_Set:Rule1[,Rule2] : [Classification] : [Status] >
[Additional text] */

... Code section ...

/* polyspace:end<Rule_Set:Rule1[,Rule2] : [Classification] : [Status] >  */
```

Square brackets *[ ]* indicate optional information.

Replace	Replace with
Rule_Set	• `MISRA-C` • `MISRA-AC-AGC` • `MISRA-CPP` • `JSF` • `Custom` If you want the comment to apply to all coding rule violations on the following line, specify `ALL`.
Rule1,Rule2,...	Rule number. For more information, see: • "MISRA C:2004 Coding Rules" • "MISRA C++ Coding Rules" • "JSF C++ Coding Rules" • "Custom Coding Rules"

Replace	Replace with
`Classification`	• `Unset` • `High` • `Medium` • `Low` • `Not a defect`
`Status`	Action for correcting the coding rule violation. Possible values are: • `Fix` • `Improve` • `Investigate` • `Justified` • `No action planned` • `Other`
`Additional text`	Additional comments.

Syntax Examples:

MISRA C rule violation:

```
polyspace<MISRA-C:6.3 : Low : Justified> Known issue
```

JSF® rule violation:

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

Copy and Paste Annotations

This example shows how to place annotations in your code to mark defects that you are already aware of but do not intend to fix immediately. Using your comments, Polyspace populates the defect **Classification**, **Status** and **Comment** fields on the **Results Summary** pane. After you have placed your comments in your code, you or another reviewer can avoid reviewing the same defect twice.

1. On the **Results Summary** pane, assign a **Classification**, **Status** and **Comment** to a defect or coding rule violation.

   a. Select the defect or coding rule violation.
   b. Under the columns, **Classification** and **Status**, select options from the drop down lists.
   c. Under the column **Comment**, enter a comment that helps you recognize the defect easily.

2. Copy the **Classification**, **Status** and **Comment**.

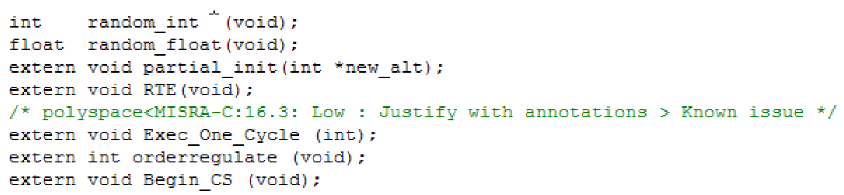
   a. On the **Results Summary** pane, right-click the defect or coding rule violation.
   b. Select **Add Pre-Justification to Clipboard**. The software copies the justification string to the clipboard.

3. Paste the **Classification**, **Status** and **Comment** in your source code.

   a. On the **Results Summary** pane, right-click the defect or coding rule violation and select **Open Editor**.

      Your source file opens on the **Code Editor** pane or an external text editor depending on your **Preferences**. The current line is the line containing the defect.
   b. Using the paste option in the text editor, paste the justification template string on the line immediately before the line containing the defect or coding rule violation.

      You can see your **Classification**, **Status** and **Comment** as a code comment in a format that Polyspace can read later.



```
int    random_int  (void);
float  random_float(void);
extern void partial_init(int *new_alt);
extern void RTE(void);
/* polyspace<MISRA-C:16.3: Low : Justify with annotations > Known issue */
extern void Exec_One_Cycle (int);
extern int orderregulate (void);
extern void Begin_CS (void);
```

c Save your source file.

4 Run the analysis again. Open your results.

On the **Results Summary** pane, the software populates the **Classification**, **Status** and **Comment** fields for the defect or rule violation. You can either ignore these findings, or filter them from the **Results Summary** pane. For more information on filtering, see “Filter and Group Results”.

Modify Predefined Target Processor Attributes

You can modify certain attributes of the predefined target processors. If your specific processor is not listed, you may be able to specify a similar processor and modify its characteristics to match your processor. The settings that you can modify for each target are shown in [brackets] in the target processor settings. See “Target processor type (C)” or “Target processor type (C++)”.

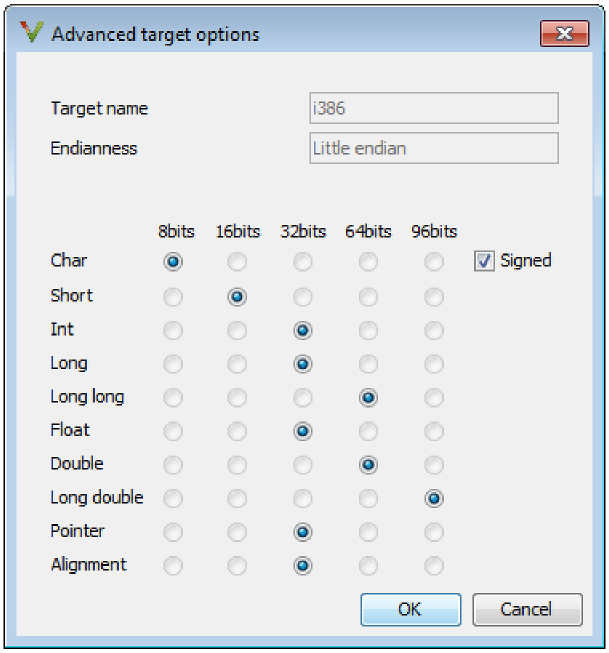
To modify target processor attributes:

1 On the **Configuration** pane, select the **Target & Compiler** node.

2 From the **Target processor type** drop-down list, select the target processor that you want to use.

3 To the right of the **Target processor type** field, click **Edit**.

The Advanced target options dialog box opens.



Target name: i386

Endianness: Little endian

	8bits	16bits	32bits	64bits	96bits	
Char	●	○	○	○	○	☑ Signed
Short	○	●	○	○	○	
Int	○	○	●	○	○	
Long	○	○	●	○	○	
Long long	○	○	○	●	○	
Float	○	○	●	○	○	
Double	○	○	○	●	○	
Long double	○	○	○	○	●	
Pointer	○	○	●	○	○	
Alignment	○	○	●	○	○	

4 Modify the attributes as required.

For information on each target option, see “Generic target options (C/C++)”.

5 Click **OK** to save your changes.

Specify Generic Target Processors

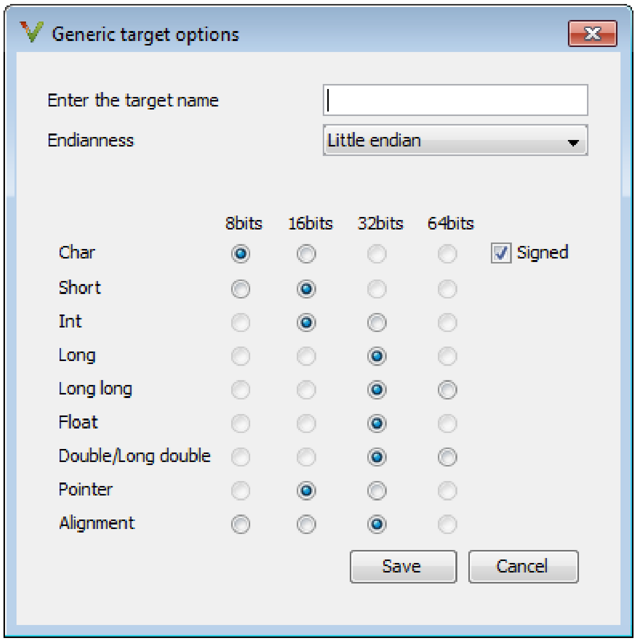
Define Generic Target

If your application is designed for a custom target processor, you can configure many basic characteristics of the target by selecting the selecting the `mcpu... (Advanced)` target, and specifying the characteristics of your processor.

To configure a generic target:

1. On the **Configuration** pane, select the **Target & Compiler** node.
2. From the **Target processor type** drop-down list, select `mcpu... (Advanced)`.

   The Generic target options dialog box opens.



Enter the target name

Endianness: Little endian

	8bits	16bits	32bits	64bits	
Char	●	○	○	○	✓ Signed
Short	○	●	○	○	
Int	○	●	○	○	
Long	○	○	●	○	
Long long	○	○	●	○	
Float	○	○	●	○	
Double/Long double	○	○	●	○	
Pointer	○	●	○	○	
Alignment	○	○	●	○	

3. In the **Enter the target name** field, enter a name, for example, `MyTarget`.
4. Specify the parameters for your target, such as the size of basic types, and alignment with arrays and structures.

For example, when the alignment of basic types within an array or structure is always 8, it implies that the storage assigned to arrays and structures is strictly determined by the size of the individual data objects (without fields and end padding).

Note: For information on each target option, see "Generic target options (C/C++)".

5 Click **Save** to save the generic target options and close the dialog box.

Common Generic Targets

The following tables describe the characteristics of common generic targets.

ST7 (Hiware C compiler : HiCross for ST7)

ST7	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16	32	32	32	32	32	16/32	unsigned	Big
alignmen	8	16/8	16/8	32/16/8	32/16/8	32/16/8	32/16/8	32/16/8	32/16/8	N/A	N/A

ST9 (GNU C compiler : gcc9 for ST9)

ST9	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16	32	32	32	64	64	16/64	unsigned	Big
alignmen	8	8	8	8	8	8	8	8	8	N/A	N/A

Hitachi H8/300, H8/300L

Hitachi H8/300, H8/300L	char	short	int	long	long long	float	double	long double	ptr	char is	endian
size	8	16	16/32	32	64	32	654	64	16	unsigned	Big
alignmen	8	16	16	16	16	16	16	16	16	N/A	N/A

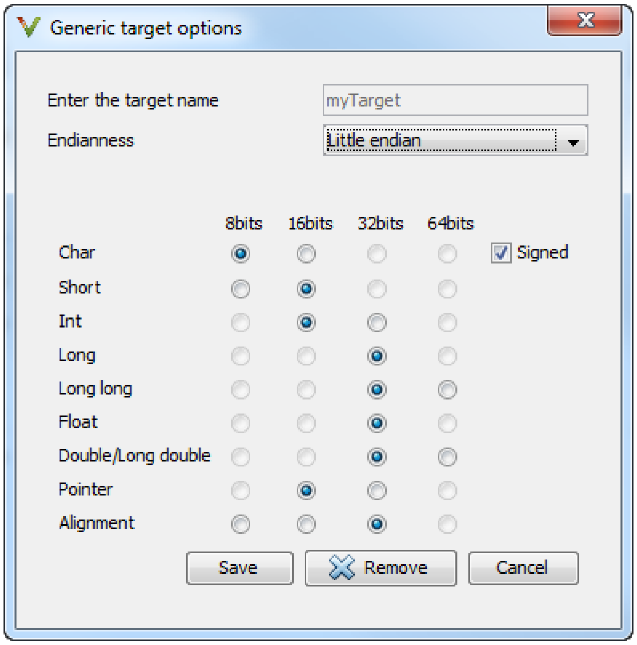
Hitachi H8/300H, H8S, H8C, H8/Tiny

Hitachi H8/300H, H8S, H8C, H8/Tiny	**char**	**short**	**int**	**long**	**long long**	**float**	**double**	**long double**	**ptr**	**char is**	**endian**
size	8	16	16/ 32	32	64	32	64	64	32	unsigned	Big
alignmen	8	16	32/ 16	32/16	32/16	32/16	32/16	32/16	32/16	N/A	N/A

View or Modify Existing Generic Targets

To view or modify generic targets that you previously created:

1. On the **Configuration** pane, select the **Target & Compiler** node.
2. From the **Target processor type** drop-down list, select your target, for example, `myTarget`.
3. Click **Edit**. The Generic target options dialog box opens, displaying your target attributes.



Enter the target name: myTarget

Endianness: Little endian

	8bits	16bits	32bits	64bits	
Char	●	○	○	○	☑ Signed
Short	○	●	○	○	
Int	○	●	○	○	
Long	○	○	●	○	
Long long	○	○	●	○	
Float	○	○	●	○	
Double/Long double	○	○	●	○	
Pointer	○	●	○	○	
Alignment	○	○	●	○	

4 If required, specify new attributes for your target. Then click **Save**.

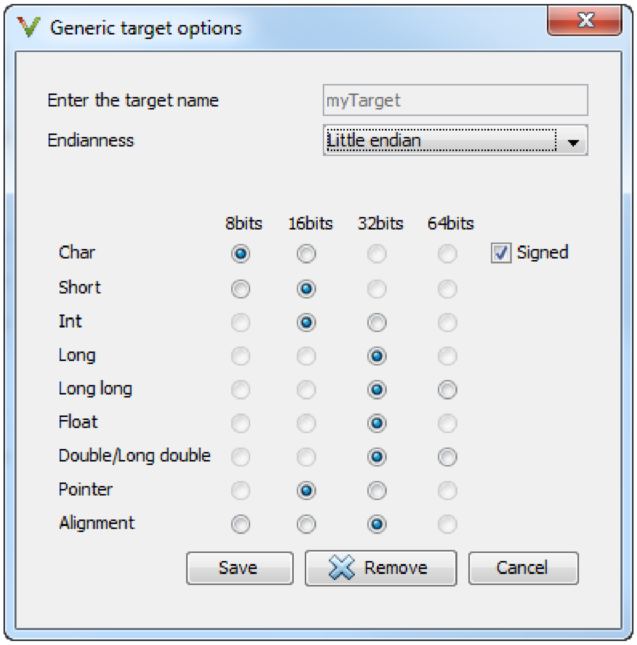
5 Otherwise, click **Cancel**.

Delete Generic Target

To delete a generic target:

1 On the **Configuration** pane, select the **Target & Compiler** node.

2 From the **Target processor type** drop-down list, select the target that you want to remove, for example, `myTarget`.



Enter the target name: myTarget

Endianness: Little endian

	8bits	16bits	32bits	64bits	
Char	●	○	○	○	☑ Signed
Short	○	●	○	○	
Int	○	●	○	○	
Long	○	○	●	○	
Long long	○	○	●	○	
Float	○	○	●	○	
Double/Long double	○	○	●	○	
Pointer	○	●	○	○	
Alignment	○	○	●	○	

3 Click **Remove**. The software removes the target from the list.

Compile Operating System-Dependent Code

This section describes the options required to compile and analyze code designed to run on specific operating systems. It contains the following:

In this section...

Predefined Compilation Flags for C Code

These flags concern the predefined **OS-target** options: `no-predefined-OS`, `linux`, `vxworks`, `Solaris` and `visual` (`-OS-target` option).

OS-target	Compilation flags	`-include` file and content
no predefined OS	`-D__STDC__`	
visual	`-D__STDC__`	`-include <product_dir>/cinclude/pst-visual.h`
vxworks	`-D__STDC__ -DANSI_PROTOTYPES -DSTATIC= -DCONST=const -D__GNUC__=2 -Dunix -D__unix -D__unix__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4__ -D__SVR4`	`-include <product_dir>/cinclude/pst-vxworks.h`
linux	`-D__STDC__ -D__GNUC__=2 -D__GNUC_MINOR__=6`	`<product_dir>/cinclude/pst-linux.h`

OS-target	Compilation flags	`-include` file and content
	`-D__GNUC__=2 -D__GNUC_MINOR__=6 -D__ELF__ -Dunix -D__unix -D__unix__ -Dlinux -D__linux -D__linux__ -Di386 -D__i386 -D__i386__ -Di686 -D__i686 -D__i686__ -Dpentiumpro -D__pentiumpro -D__pentiumpro__`	
Solaris	`-D__STDC__ -D__GNUC__=2 -D__GNUC_MINOR__=8 -D__GNUC__=2 -D__GNUC_MINOR__=8 -D__GCC_NEW_VARARGS__ -Dunix -D__unix -D__unix__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__ -D__svr4__ -D__SVR4`	No `-include` file mentioned

Note: The use of the `-OS-target` option is equivalent to the following alternative approaches.

- Setting the same -D flags manually, or
- Using the `-include` option on a copied and modified pst-*OS-target*.h file

Predefined Compilation Flags for C++ Code

The following table shown for each –`OS-target`, the list of compilation flags defined by default, including pre-include header file (see also `-include`):

-OS-target	Compilation flags	-include file	Minimum set of options
Linux	`-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline-D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_SPECIALIZATION -D__GNU_SOURCE -D__STDC__ -D__ELF__ -Dunix -D__unix -D__unix__ -Dlinux -D__linux -D__linux__ -Di386 -D__i386 -D__i386__ -Di686 -D__i686 -D__i686__ -Dpentiumpro -D__pentiumpro -D__pentiumpro__`	`<product_dir>/cinclude/pst-linux.h`	`polyspace-[desktop-]cpp -OS-target Linux \ -I <polyspace_install>/include/ include-linux \ -I <product_dir>/include/include-linux/next` Where the Polyspace product has been installed in the folder `<polyspace_install>`
vxWorks	`-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline-D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_SPECIALIZATION -DANSI_PROTOTYPES-DSTATIC=-DCONST=const-D__STDC-D__GNU_SOURCE -Dunix -D__unix -D__unux__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__-D__svr4-D__SVR4`	`<product_dir>/cinclude/pstvxworks. h`	`polyspace-[desktop-]cpp \ -OS-target vxworks \ -I /your_path_to/Vxworks_include_folders`

-OS-target	Compilation flags	-include file	Minimum set of options
visual / visual6	`-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__STRICT_ANSI__ -D__inline__=inline-D__signed__=signed -D__gnuc_va_list=va_list -D_POSIX_SOURCE -D__STL_CLASS_PARTIAL_SPECIALIZATION`	`<product_dir>/cinclude/pstvisual. h`	
Solaris	`-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__inline__=inline-D__signed__=signed -D__gnuc_va_list=va_list -D__STL_CLASS_PARTIAL_SPECIALIZATION -D__GNU_SOURCE -D__STDC-D__GCC_NEW_VARARGS__ -Dunix -D__unix -D__unux__ -Dsparc -D__sparc -D__sparc__ -Dsun -D__sun -D__sun__-D__svr4-D__SVR4`		If Polyspace runs on a Linux machine: `polyspace-bug-finder-no-desktop \ -OS-target Solaris \ -I /your_path_to_solaris_include` If Polyspace runs on a Solaris machine: `polyspace-bug-finder-no-desktop \ -OS-target Solaris \ -I /usr/include`
no-predefin OS	`-D__SIZE_TYPE__=unsigned -D__PTRDIFF_TYPE__=int -D__STRICT_ANSI__ -D__inline__=inline-D__signed__=signed -D__gnuc_va_list=va_list -D_POSIX_SOURCE -D__STL_CLASS_PARTIAL_SPECIALIZATION`		`polyspace-bug-finder-no-desktop \ -OS-target no-predefined-OS \ -I /your_path_to/MyTarget_include_folders`

Note: This list of compiler flags is written in every log file.

My Target Application Runs on Linux

The minimum set of options is as follows:

```
polyspace-bug-finder-no-desktop \
 -OS-target Linux \
 -I Polyspace_Install/polyspace/verifier/cxx/include/include-libc \

 ...
```

where the Polyspace product has been installed in the folder `Polyspace_Install`.

If your target application runs on Linux but you are launching your analysis from Windows, the minimum set of options is as follows:

```
polyspace-bug-finder-no-desktop \
 -OS-target Linux \
 -I Polyspace_Install\polyspace\verifier\cxx\include\include-libc \

 ...
```

where the Polyspace product has been installed in the folder `Polyspace_Install`.

My Target Application Runs on Solaris

If Polyspace software runs on a Linux machine:

```
polyspace-bug-finder-no-desktop \
 -OS-target Solaris \
 -I /your_path_to_solaris_include
```

If Polyspace software runs on a Solaris™ machine:

```
polyspace-bug-finder-no-desktop \
 -OS-target Solaris \
 -I /usr/include
```

My Target Application Runs on Vxworks

If Polyspace software runs on either a Solaris or a Linux machine:

```
polyspace-bug-finder-no-desktop \
 -OS-target vxworks \
 -I /your_path_to/Vxworks_include_folders
```

My Target Application Does Not Run on Linux, vxworks nor Solaris

If Polyspace software does not run on either a Solaris or a Linux machine:

```
polyspace-bug-finder-no-desktop \
 -OS-target no-predefined-OS \
 -I /your_path_to/MyTarget_include_folders
```

Address Alignment

Polyspace software handles address alignment by calculating `sizeof` and alignments. This approach takes into account 3 constraints implied by the ANSI standard which ensure that:

- that global `sizeof` and `offsetof` fields are optimum (i.e. as short as possible);
- the alignment of addressable units is respected;
- global alignment is respected.

Consider the example:

```
struct foo {char a; int b;}
```

- Each field must be aligned; that is, the starting offset of a field must be a multiple of its own size[1]
- So in the example, `char a` begins at offset 0 and its size is 8 bits. `int b` cannot begin at 8 (the end of the previous field) because the starting offset must be a multiple of its own size (32 bits). Consequently, `int b` begins at offset=32. The size of the `struct foo` before global alignment is therefore 64 bits.
- The global alignment of a structure is the maximum of the individual alignments of each of its fields;
- In the example, `global_alignment = max (alignment char a, alignment int b) = max (8, 32) = 32`
- The size of a struct must be a multiple of its global alignment. In our case, `b` begins at 32 and is 32 long, and the size of the struct (64) is a multiple of the `global_alignment` (32), so `sizeof` is not adjusted.

1. except in the cases of "double" and "long" on some targets.

Ignore or Replace Keywords Before Compilation

You can ignore noncompliant keywords, for example, `far` or `0x`, which precede an absolute address. The template `myTpl.pl` (listed below) allows you to ignore these keywords:

1. Save the listed template as `C:\Polyspace\myTpl.pl`.
2. Select the **Configuration** > **Target & Compiler** > **Environment Settings** pane.
3. To the right of the **Command/script to apply to preprocessed files** field, click on the file icon.
4. Use the Open File dialog box to navigate to `C:\Polyspace`.
5. In the **File name** field, enter `myTpl.pl`.
6. Click **Open**. You see `C:\Polyspace\myTpl.pl` in the **Command/script to apply to preprocessed files** field.

For more information, see “Command/script to apply to preprocessed files (C/C++)”.

Content of myTpl.pl file

```
#!/usr/bin/perl

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

$version = 0.1;

$INFILE = STDIN;
$OUTFILE = STDOUT;

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

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

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

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

 # Convert current line to lower case
# $_ =~ tr/A-Z/a-z/;

 # Print the current processed line
 print $OUTFILE $_;
}
```

Perl Regular Expression Summary

```
#########################################################
# Metacharacter What it matches
#########################################################
# Single Characters
# . Any character except newline
# [a-z0-9] Any single character in the set
# [^a-z0-9] Any character not in set
# \d A digit same as
# \D A non digit same as [^0-9]
# \w An Alphanumeric (word) character
# \W Non Alphanumeric (non-word) character
#
# Whitespace Characters
# \s Whitespace character
# \S Non-whitespace character
# \n newline
# \r return
# \t tab
# \f formfeed
# \b backspace
#
# Anchored Characters
# \B word boundary when no inside []
# \B non-word boundary
# ^ Matches to beginning of line
# $ Matches to end of line
#
# Repeated Characters
# x? 0 or 1 occurrence of x
# x* 0 or more x's
# x+ 1 or more x's
# x{m,n} Matches at least m x's and no more than n x's
# abc Exactly "abc"
# to|be|great One of "to", "be" or "great"
#
# Remembered Characters
# (string) Used for back referencing see below
# \1 or $1 First set of parentheses
# \2 or $2 First second of parentheses
# \3 or $3 First third of parentheses
##########################################################
# Back referencing
#
# e.g. swap first two words around on a line
# red cat -> cat red
# s/(\w+) (\w+)/$2 $1/;
#
##########################################################
```

Analyze Keil or IAR Dialects

Typical embedded control applications frequently read and write port data, set timer registers and read input captures. To deal with this without using assembly language, some microprocessor compilers have specified special data types like `sfrand sbit`. Typical declarations are:

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

These declarations reside in header files such as `regxx.h` for the basic `80Cxxx` micro processor. The definition of `sfr` in these header files customizes the compiler to the target processor.

When accessing a register or a port, using `sfr` data is then simple, but is not part of standard ANSI C:

```
int status,P0;

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

You can analyze this type of code using the **Dialect** option . This option allows the software to support the Keil or IAR C language extensions even if some structures, keywords, and syntax are not ANSI standard. The following tables summarize what is supported when analyzing code that is associated with the Keil or IAR dialects.

The following table summarizes the supported Keil C language extensions:

Example: `-dialect keil -sfr-types sfr=8`

Type/Language	Description	Example	Restrictions

Type `bit`	• An expression to type bit gives values in range [0,1].	`bit x = 0, y = 1,`
		` z = 2;`
		`assert(x == 0);`
		`assert(y == 1);`
		`assert(z == 1);`
			pointers to bits and arrays of bits are not allowed

Type/Language	Description	Example	Restrictions
	• Converting an expression in the type, gives 1 if it is not equal to 0, else 0. This behavior is similar to c ++ `bool`type.	`assert(sizeof(bit) == sizeof(int));`	
Type `sfr`	• The `-sfr-types` option defines unsigned types **name** and size in bits. • The behavior of a variable follows a variable of type integral. • A variable which overlaps another one (in term of address) will be considered as volatile.	`sfr x = 0xf0; // declaration of variable x at address 0xF0` `sfr16 y = 0x4EEF;` For this example, options need to be: `-dialect keil` `-sfr-types sfr=8, sfr16=16`	sfr and sbit types are only allowed in declarations of external global variables.
Type `sbit`	• Each read/write access of a variable is replaced by an access of the corresponding sfr variable access. • Only external global variables can be mapped with a sbit variable. • Allowed expressions are integer variables, cells of array of int and struct/union integral fields. • a variable can also be declared as extern bit in an another file.	`sfr x = 0xF0;` `sbit x1 = x ^ 1; // 1st bit of x` `sbit x2 = 0xF0 ^ 2; // 2nd bit of x` `sbit x3 = 0xF3; // 3rd bit of x` `sbit y0 = t[3] ^ 1;` `/* file1.c */` `sbit x = P0 ^ 1;` `/* file2.c */` `extern bit x;` `x = 1; // set the 1st bit of P0 to 1`	

Type/Language	Description	Example	Restrictions

Absolute variable location	Allowed constants are integers, strings and identifiers.	`int var _at_ 0xF0`
		`int x @ 0xFE ;`
		`static const`
		`int y @ 0xA0 = 3;`
			Absolute variable locations are ignored (even if declared with a `#pragma` location).

Interrupt functions	A warnings in the log file is displayed when an interrupt function has been found: "interrupt handler detected : <name>" or "task entry point detected : <name>"	`void foo1 (void)`
		`interrupt XX = YY`
		`using 99 {…}`
		`void foo2 (void) _`
		`task_ 99 _priority_`
		`2 {…}`
			Entry points and interrupts are not taken into account as `-entry-points`.

Keywords ignored	alien, bdata, far, idata, ebdata, huge, sdata, small, compact, large, reentrant. Defining -D __C51__, keywords large code, data, xdata, pdata and xhuge are ignored.		

The following table summarize the IAR dialect:

Example: `-dialect iar -sfr-types sfr=8`

Type/Language	Description	Example	Restrictions

Type `bit`	• An expression to type bit gives values in range [0,1].
	• Converting an expression in the type, gives 1 if it is not equal to 0, else 0. This behavior is similar to c ++ bool type.
	• If initialized with values 0 or 1, a variable of type bit is a simple variable (like a c++ bool).
	• A variable of type bit is a register bit
		`union {`
		`  int v;`
		`  struct {`
		`    int z;`
		`  } y;`
		`} s;`
		
		`void f(void) {`
		`  bit y1 = s.y.z . 2;`
		`  bit x4 = x.4;`
		`  bit x5 = 0xF0 . 5;`
		`  y1 = 1;`
		` // 2nd bit of s.y.z`
		` // is set to 1`
		`};`
			pointers to bits and arrays of bits are not allowed

Type/Language	Description	Example	Restrictions
	variable (mapped with a bit or a sfr type)		
Type `sfr`	• The `-sfr-types` option defines unsigned types name and size. • The behavior of a variable follows a variable of type integral. • A variable which overlaps another one (in term of address) will be considered as volatile.	`sfr x = 0xf0; // declaration of variable x at address 0xF0`	sfr and sbit types are only allowed in declarations of external global variables.

Individual `bit` access	• Individual bit can be accessed without using sbit/bit variables.
	• Type is allowed for integer variables, cells of integer array, and struct/union integral fields.
		`int x[3], y;`
		`x[2].2 = x[0].3 + y.1;`	

Absolute variable location	Allowed constants are integers, strings and identifiers.	`int var @ 0xF0;`
		`int xx @ 0xFE ;`
		`static const int y   \`
		` @0xA0 = 3;`
			Absolute variable locations are ignored (even if declared with a #pragma location).

Interrupt functions	• A warning is displayed in the log file when an interrupt function has been found: "interrupt handler detected : funcname"	`interrupt [1]    \`
		` using [99] void   \`
		` foo1(void) { ... };`
		
		`monitor [3] void   \`
		` foo2(void) { ... };`
			Entry points and interrupts are not taken into account as `-entry-points`.

Type/Language	Description	Example	Restrictions
	• A monitor function is a function that disables interrupts while it is executing, and then restores the previous interrupt state at function exit.		
Keywords ignored	`saddr`, `reentrant`, `reentrant_idata`, `non_banked`, `plm`, `bdata`, `idata`, `pdata`, `code`, `data`, `xdata`, `xhuge`, `interrupt`, `__interrupt` and `__intrinsic`		

Unnamed struct/ union	• Fields of unions/ structs without a tag or a name can be accessed without naming their parent struct.
	• Option `-allow-unnamed-fields` need to be used to allow anonymous struct fields.
	• On a conflict between a field of an anonymous struct with other identifiers:
	  • with a variable name, field name is hidden
	  • with a field of another anonymous struct at different scope, closer scope is chosen
		`union { int x; };`
		`union { int y; struct { int z; }; } @ 0xF0;`	

Type/Language	Description	Example	Restrictions
	• with a field of another anonymous struct at same scope: an error "anonymous struct field name <name> conflict“ is displayed in the log file.		

`no_init` attribute	• a global variable declared with this attribute is handled like an external variable.
	• It is handled like a type qualifier.
		`no_init int x;`
		`no_init union`
		`{ int y; } @ 0xFE;`
			The `#pragma no_init` does not affect the code.

The option `-sfr-types` defines the size of a `sfr` type for the Keil or IAR dialect.

The syntax for an `sfr` element in the list is `type-name=typesize`.

For example:

```
-sfr-types sfr=8,sfr16=16
```

defines two `sfr` types: `sfr` with a size of 8 bits, and `sfr16` with a size of 16-bits. A value type-name must be given only once. 8, 16 and 32 are the only supported values for `type-size`.

Note: As soon as an `sfr` type is used in the code, you must specify its name and size, even if it is the keyword `sfr`.

Note: Many IAR and Keil compilers currently exist that are associated to specific targets. It is difficult to maintain a complete list of those supported.

Supported C++ 2011 Standards

Standard	Description	Supported
C++2011-N2118	Rvalue references	Yes
C++2011-N2439	Rvalue references for *this	Yes
C++2011-N1610	Initialization of class objects by rvalues	Yes
C++2011-N2756	Non-static data member initializers	Yes
C++2011-N2242	Variadic templates	Yes
C++2011-N2555	Extending variadic template template parameters	Yes
C++2011-N2672	Initializer lists	Yes
C++2011-N1720	Static assertions	Yes
C++2011-N1984	auto-typed variables	Yes
C++2011-N1737	Multi-declarator auto	Yes
C++2011-N2546	Removal of auto as a storage-class specifier	Yes
C++2011-N2541	New function declarator syntax	Yes
C++2011-N2927	New wording for C++0x lambdas	Yes
C++2011-N2343	Declared type of an expression	Yes
C++2011-N3276	decltype and call expressions	Yes

Standard	Description	Supported
C++2011-N1757	Right angle brackets	Yes
C++2011-DR226	Default template arguments for function templates	Yes
C++2011-DR339	Solving the SFINAE problem for expressions	Yes
C++2011-N2258	Template aliases	Yes
C++2011-N1987	Extern templates	Yes
C++2011-N2431	Null pointer constant	Yes
C++2011-N2347	Strongly-typed enums	Yes
C++2011-N2764	Forward declarations for enums	Yes
C++2011-N2761	Generalized attributes	Yes
C++2011-N2235	Generalized constant expressions	Yes
C++2011-N2341	Alignment support	Yes
C++2011-N1986	Delegating constructors	Yes
C++2011-N2540	Inheriting constructors	Yes
C++2011-N2437	Explicit conversion operators	Yes
C++2011-N2249	New character types	Yes
C++2011-N2442	Unicode string literals	Yes

Standard	Description	Supported
C++2011-N2442	Raw string literals	Yes
C++2011-N2170	Universal character name literals	No
C++2011-N2765	User-defined literals	Yes
C++2011-N2342	Standard Layout Types	Not applicable[1]
C++2011-N2346	Defaulted and deleted functions	Yes
C++2011-N1791	Extended friend declarations	Yes
C++2011-N2253	Extending sizeof	Yes
C++2011-N2535	Inline namespaces	Yes
C++2011-N2544	Unrestricted unions	Yes
C++2011-N2657	Local and unnamed types as template arguments	Yes
C++2011-N2930	Range-based for	Yes
C++2011-N2928	Explicit virtual overrides	Yes
C++2011-N3050	Allowing move constructors to throw [noexcept]	Yes
C++2011-N3053	Defining move special member functions	Yes
C++2011-N2239	Concurrency - Sequence points	Not applicable[1]
C++2011-N2427	Concurrency - Atomic operations	No

Standard	Description	Supported
C++2011-N2748	Concurrency - Strong Compare and Exchange	No
C++2011-N2752	Concurrency - Bidirectional Fences	No
C++2011-N2429	Concurrency - Memory model	Not applicable[1]
C++2011-N2664	Concurrency - Data-dependency ordering: atomics and memory model	No
C++2011-N2179	Concurrency - Propagating exceptions	No
C++2011-N2440	Concurrency - Abandoning a process and at_quick_exit	Yes
C++2011-N2547	Concurrency - Allow atomics use in signal handlers	No
C++2011-N2659	Concurrency - Thread-local storage	No
C++2011-N2660	Concurrency - Dynamic initialization and destruction with concurrency	No
C++2011-N2340	__func__ predefined identifier	Yes
C++2011-N1653	C99 preprocessor	Yes
C++2011-N1811	long long	Yes
C++2011-N1988	Extended integral types	Not applicable[1]

[1] This C++11 requirement is not a factor in a Polyspace analysis.

See Also

"C++11 Extensions (C++)"

Gather Compilation Options Efficiently

The code is often tuned for the target (as discussed in "Analyze Keil or IAR Dialects" on page 1-68). Rather than applying minor changes to the code, create a single `polyspace.h` file which contains target specific functions and options. The `-include` option can then be used to force the inclusion of the `polyspace.h` file in the source files.

Where there are missing prototypes or conflicts in variable definition, writing the expected definition or prototype within such a header file will yield several advantages.

Direct benefits:

- The error detection is much faster since it will be detected during compilation rather than in the link or subsequent phases.
- The position of the error will be identified more precisely.
- Original source files will not need to be modified.

Indirect benefits:

- The file is automatically included as the very first file in the original .c files.
- The file can contain much more powerful macro definitions than simple -D options.
- The file is reusable for other projects developed under the same environment.

Example

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

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

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

// Generic definitions, reusable from one project to another
#define far
#define at(x)

// A prototype may be positioned here to aid in the solution of
// a link phase conflict between
// declaration and definition. This will allow detection of the
// same error at compilation time instead of at link time.
// Leads to:
// - earlier detection
// - precise localisation of conflict at compilation time
void f(int);

// The same also applies to variables.
extern int x;

// Standard library stubs can be avoided,
// and OS standard prototypes redefined.

#define POLYSPACE_NO_STANDARD_STUBS // use this flag to prevent the
              //automatic stubbing of std functions
#define __polyspace_no_sscanf
#define __polyspace_no_fgetc
void sscanf(int, char, char, char, char, char);
void fgetc(void);
```

Specify Constraints

This example shows how to specify constraints on variables in your code. Polyspace uses the code that you provide to make assumptions about variable ranges, allowed buffer size for pointers, and other items. However, sometimes the assumptions are broader than what you expect because:

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

Because of these broad assumptions, Polyspace can sometimes produce false positives.

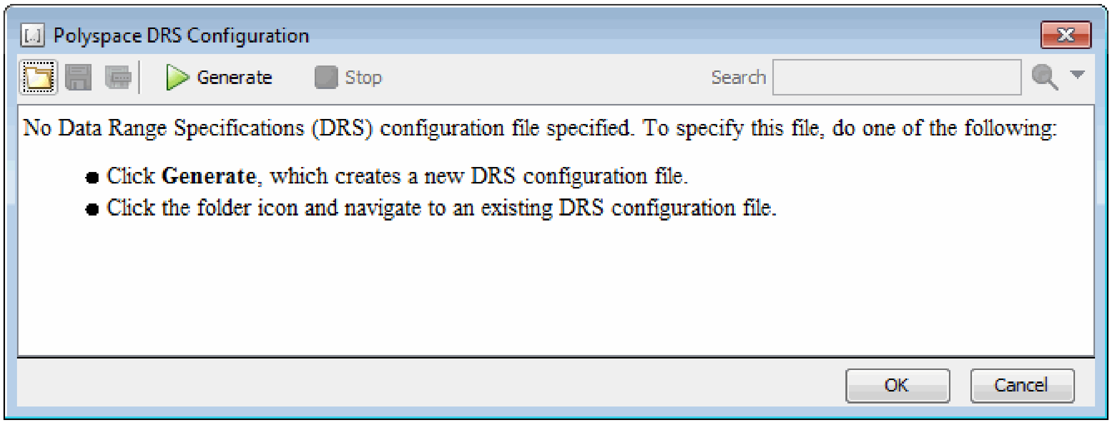
To reduce the number of such false positives, you can specify additional constraints on variables using a **Data Range Specification** or DRS template. After you create a template, you can save the template as an XML file and use it for subsequent verifications. If your source code changes, you can update the previous template. You do not have to create another template.

In this section...

Create Constraint Template

1. On the **Configuration** pane, select **Inputs & Stubbing**.
2. To the right of **Variable/function range setup**, click the **Edit** button.

   The Polyspace DRS Configuration dialog box opens.



No Data Range Specifications (DRS) configuration file specified. To specify this file, do one of the following:

- Click **Generate**, which creates a new DRS configuration file.
- Click the folder icon and navigate to an existing DRS configuration file.

3 Click **Generate**. The software compiles your project and creates a template.

The template contains a list of all variables on which you can provide constraints.

4 Specify your constraints and save the template as an XML file. For more information, see "Constraints".

5 Click **OK**.

You see the full path to the template XML file in the **Variable/function range setup** field. If you run a verification, Polyspace uses this template for extracting variable constraints.

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

To specify constraints in your code, you can use:

- Appropriate error handling tests in your code.
  Polyspace checks if the errors can actually occur. If they do not occur, the test blocks appear as **Unreachable code**.
- The assert macro. For example, to constrain a variable `var` in the range [0,10], you can use `assert(var >= 0 && var <=10);`.

Polyspace checks your `assert` statements to see if the condition can be false. Following the `assert` statement, Polyspace considers that the `assert` condition is true. Therefore, if you use appropriate `assert` statements, for the remaining code in the same scope, your variables are constrained. For examples, see User assertion.

Update Existing Template

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

See Also

"Variable/function range setup (C/C++)"

Related Examples

- "Constrain Global Variables"

Constraints

The Polyspace DRS Configuration interface allows you to specify constraints for:

- Global Variables.
- User-defined Functions.
- Stubbed Functions.

For more information, see “Specify Constraints”.

The following table lists the constraints that can be specified through this interface.

Column	Settings
Name	Displays the list of variables and functions in your Project for which you can specify data ranges. This Column displays three expandable menu items: • **Globals** – Displays global variables in the project. • **User defined functions** – Displays user-defined functions in the project. Expand a function name to see its inputs. • **Stubbed functions** – Displays a list of stub functions in the project. Expand a function name to see the inputs and return values.
File	Displays the name of the source file containing the variable or function.
Attributes	Displays information about the variable or function. For example, static variables display `static`.
Data Type	Displays the variable type.
Main Generator Called	Applicable only for user-defined functions. Specifies whether the main generator calls the function: • `MAIN GENERATOR` – Main generator may call this function, depending on the value of the `-functions-called-in-loop` (C) or `-main-generator-calls` (C++) parameter. • `NO` – Main generator will not call this function.

Column	Settings
	• **`YES`** – Main generator will call this function.
Init Mode	Specifies how the software assigns a range to the variable: • **`MAIN GENERATOR`** – Variable range is assigned depending on the settings of the main generator options `-variables-written-before-loop` and `-no-def-init-glob`. (For C++, the options are `-main-generator-writes-variables`, and `-no-def-init-glob`.) • **`IGNORE`** – Variable is not assigned to any range, even if a range is specified. • **`INIT`** – Variable is assigned to the specified range only at initialization, and keeps the range until first write. • **`PERMANENT`** – Variable is permanently assigned to the specified range. If the variable is assigned outside this range during the program, no warning is provided. Use the `globalassert` mode if you need a warning. User-defined functions support only `INIT` mode. Stub functions support only `PERMANENT` mode. For C verifications, global pointers support `MAIN GENERATOR`, `IGNORE`, or `INIT` mode. • **`MAIN GENERATOR`** – Pointer follows the options of the main generator. • **`IGNORE`** – Pointer is not initialized • **`INIT`** – Specify if the pointer is NULL, and how the pointed object is allocated (**Initialize Pointer** and **Init Allocated** options).

Column	Settings

Init Range	Specifies the minimum and maximum values for the variable.
	
	You can use the keywords `min` and `max` to denote the minimum and maximum values of the variable type. For example, for the type long, `min` and `max` correspond to -2^31 and 2^31-1 respectively.
	
	You can also use hexadecimal values. For example: `0x12..0x100`
	
	For `enum` variables, you cannot specify ranges directly using the enumerator constants. Instead use the values represented by the constants.
	
	For `enum` variables, you can also use the keywords `enum_min` and `enum_max` to denote the minimum and maximum values that the variable can take. For example, for an `enum` variable of the type defined below, `enum_min` is 0 and `enum_max` is 5:
	
	`enum week{ sunday, monday=0, tuesday,`
	`     wednesday, thursday, friday, saturday};`

Initialize Pointer	Applicable only to pointers. Enabled only when you specify **Init Mode**:`INIT`. Specifies whether the pointer should be NULL: • **`May-be NULL`** – The pointer could potentially be a NULL pointer (or not). • **`Not Null`** – The pointer is never initialized as a null pointer. • **`Null`** – The pointer is initialized as NULL. **Note:** Not applicable for C++ projects.

Column	Settings
Init Allocated	Applicable only to pointers. Enabled only when you specify **Init Mode**:`INIT`. Specifies how the pointed object is allocated: • **`MAIN GENERATOR`** – The pointed object is allocated by the main generator. • **`None`** – Pointed object is not written. • **`SINGLE`** – Write the pointed object or the first element of an array. (This setting is useful for stubbed function parameters.) • **`MULTI`** – All objects (or array elements) are initialized. See . **Note:** Not applicable for C++ projects.
# Allocated Objects	Applicable only to pointers. Specifies how many objects are pointed to by the pointer (the pointed object is considered as an array). **Note:** The Init Allocated parameter specifies how many allocated objects are actually initialized. See . **Note:** Not applicable for C++ projects.
Global Assert	Specifies whether to perform an assert check on the variable at global initialization, and after each assignment.
Global Assert Range	Specifies the minimum and maximum values for the range you want to check.
Comment	Remarks that you enter, for example, justification for your DRS values.

Storage of Polyspace Preferences

The software stores the settings that you specify through the Polyspace Preferences dialog box in the following file:

- Windows: `$Drive\Users\$User\AppData\Roaming\MathWorks \MATLAB \$Release\Polyspace\polyspace.prf`
- Linux: `/home/$User/.matlab/$Release/Polyspace/polyspace.prf`

Here, *`$Drive`* is the drive where the operating system files are located such as `C:`, *`$User`* is the username and *`$Release`* is the release number.

The following file stores the location of all installed Polyspace products across various releases:

- Windows: `$Drive\Users\$User\AppData\Roaming\MathWorks \MATLAB \AppData\Roaming\MathWorks\MATLAB \polyspace_shared \polyspace_products.prf`
- Linux : `/home/$User/.matlab/polyspace_shared/polyspace_products.prf`

Coding Rule Sets and Concepts

Rule Checking

Polyspace Coding Rule Checker

Polyspace software allows you to analyze code to demonstrate compliance with established C and C++ coding standards:

- MISRA C 2004
- MISRA C 2012
- MISRA® C++:2008
- JSF++:2005

Applying coding rules can reduce the number of defects and improve the quality of your code.

While creating a project, you specify both the coding standard, and which rules to enforce. Polyspace software performs rule checking before and during the analysis. Violations appear in the **Results Summary** pane.

If any source files in the analysis do not compile, coding rules checking will be incomplete. The coding rules checker results:

- May not contain full results for files that did not compile
- May not contain full results for the files that did compile as some rules are checked only after compilation is complete

Note: When you enable the Compilation Assistant *and* coding rules checking, the software does not report coding rule violations if there are compilation errors.

Differences Between Bug Finder and Code Prover

Coding rule checker results can differ between Polyspace Bug Finder and Polyspace Code Prover. The rule checking engines are identical in Bug Finder and Code Prover, but the context in which the checkers execute is not the same. If a project is launched from Bug Finder and Code Prover with the same source files and same configuration options, the coding rule results can differ. For example, the main generator used in Code Prover activates global variables, which causes the rule checkers to identify such global variables as initialized. The Bug Finder does not have a main generator, so handles the initialization of the global variables differently. Another difference is how violations are reported. The coding rules violations found in header files are not reported to the user in Bug Finder, but these violations are visible in Code Prover.

This difference can occur in MISRA C:2004, MISRA C:2012, MISRA C++, and JSF++. See the **Polyspace Specification** column or the **Description** for each rule.

Even though there are differences between rules checkers in Bug Finder and Code Prover, both reports are valid in their own context. For quick coding rules checking, use Polyspace Bug Finder.

Polyspace MISRA C 2004 and MISRA AC AGC Checkers

The Polyspace MISRA C:2004 checker helps you comply with the MISRA C 2004 coding standard.[2]

When MISRA C rules are violated, the MISRA C checker enables Polyspace software to provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis.

The MISRA C checker can check nearly all of the **142** MISRA C:2004 rules.

The MISRA AC AGC checker checks rules from the OBL (obligatory) and REC (recommended) categories specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation.*

There are subsets of MISRA coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your results. When you set up rule checking, you can select these subsets directly. These subsets are defined in:

- “Software Quality Objective Subsets (C:2004)” on page 2-5
- “Software Quality Objective Subsets (AC AGC)” on page 2-10

Note: The Polyspace MISRA checker is based on MISRA C:2004, which also incorporates MISRA C Technical Corrigendum (http://www.misra-c.com).

---

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

Software Quality Objective Subsets (C:2004)

In this section…

Rules in `SQO-Subset1`

In Polyspace Code Prover, the following set of coding rules will typically reduce the number of unproven results.

Rule number	Description
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.
11.3	A cast should not be performed between a pointer type and an integral type.
12.12	The underlying bit representations of floating-point values shall not be used.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a *for* statement shall not contain any objects of floating type.
13.5	The three expressions of a *for* statement shall be concerned only with loop control.
14.4	The *goto* statement shall not be used.
14.7	A function shall have a single point of exit at the end of the function.

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

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

Rules in `SQO-Subset2`

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

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

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

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

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

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

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

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

Software Quality Objective Subsets (AC AGC)

In this section...
"Rules in SQO-Subset1" on page 2-10
"Rules in SQO-Subset2" on page 2-11

Rules in `SQO-Subset1`

In Polyspace Code Prover, the following set of coding rules will typically reduce the number of unproven results.

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

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

Rules in `SQO-Subset2`

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

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

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

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

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

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

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

MISRA C:2004 Coding Rules

Supported MISRA C:2004 Rules

The following tables list MISRA C:2004 coding rules that the Polyspace coding rules checker supports. Details regarding how the software checks individual rules and any limitations on the scope of checking are described in the "Polyspace Specification" column.

Note: The Polyspace coding rules checker:

- Supports MISRA-C:2004 Technical Corrigendum 1 for rules 4.1, 5.1, 5.3, 6.1, 6.3, 7.1, 9.2, 10.5, 12.6, 13.5, and 15.0.
- Checks rules specified by *MISRA AC AGC Guidelines for the Application of MISRA-C:2004 in the Context of Automatic Code Generation*.

The software reports most violations during the compile phase of an analysis. However, the software detects violations of rules 9.1 (`Non-initialized variable`), 12.11 (one of the overflow checks) using `-scalar-overflows-checks signed-and-unsigned`), 13.7 (dead code), 14.1 (dead code), 16.2 and 21.1 during code analysis, and reports these violations as run-time errors.

Note: Some violations of rules 13.7 and 14.1 are reported during the compile phase of analysis.

Environment

N.	MISRA Definition	Messages in report file	Polyspace Specification
1.1	All code shall conform to ISO® 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.	The text *All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996* precedes each of the following messages: • ANSI C does not allow '#include_next' • ANSI C does not allow macros with variable arguments list	All the supported extensions lead to a violation of this MISRA rule. Standard compilation error messages do not lead to a violation of this MISRA rule and remain unchanged.

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• ANSI C does not allow ‘#assert’ • ANSI C does not allow '#unassert' • ANSI C does not allow testing assertions • ANSI C does not allow '#ident' • ANSI C does not allow '#sccs' • text following '#else' violates ANSI standard. • text following '#endif' violates ANSI standard. • text following '#else' or '#endif' violates ANSI standard.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
1.1 (cont.)		The text *All code shall conform to ISO 9899:1990 Programming languages C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/ COR2:1996* precedes each of the following messages: • ANSI C90 forbids 'long long int' type. • ANSI C90 forbids 'long double' type. • ANSI C90 forbids long long integer constants. • Keyword 'inline' should not be used. • Array of zero size should not be used. • Integer constant does not fit within unsigned long int. • Integer constant does not fit within long int. • Too many nesting levels of #includes: N_1. The limit is N_0. • Too many macro definitions: N_1. The limit is N_0. • Too many nesting levels for control flow: N_1. The limit is N_0.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• Too many enumeration constants: N_1. The limit is N_0.	

Language Extensions

N.	MISRA Definition	Messages in report file	Polyspace Specification
2.1	Assembly language shall be encapsulated and isolated.	Assembly language shall be encapsulated and isolated.	No warnings if code is encapsulated in `asm` functions or in `asm pragma` (only warning is given on asm statements even if it is encapsulated by a MACRO).
2.2	Source code shall only use /* */ style comments	C++ comments shall not be used.	C++ comments are handled as comments but lead to a violation of this MISRA rule **Note**: This rule cannot be annotated in the source code.
2.3	The character sequence /* shall not be used within a comment	The character sequence /* shall not appear within a comment.	This rule violation is also raised when the character sequence /* inside a C++ comment. **Note**: This rule cannot be annotated in the source code.

Documentation

Rule	MISRA Definition	Messages in report file	Polyspace Specification
3.4	All uses of the *#pragma* directive shall be documented and explained.	All uses of the #pragma directive shall be documented and explained.	To check this rule, the option `-allowed-pragmas` must be set to the list of pragmas that are allowed in source files. Warning if a pragma that does not belong to the list is found.

Character Sets

N.	MISRA Definition	Messages in report file	Polyspace Specification
4.1	Only those escape sequences which are defined in the ISO C standard shall be used.	\<character> is not an ISO C escape sequence Only those escape sequences which are defined in the ISO C standard shall be used.	
4.2	Trigraphs shall not be used.	Trigraphs shall not be used.	Trigraphs are handled and converted to the equivalent character but lead to a violation of the MISRA rule

Identifiers

N.	MISRA Definition	Messages in report file	Polyspace Specification
5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters	Identifier 'XX' should not rely on the significance of more than 31 characters.	All identifiers (global, static and local) are checked.
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	• Local declaration of XX is hiding another identifier. • Declaration of parameter XX is hiding another identifier.	Assumes that rule 8.1 is not violated.
5.3	A typedef name shall be a unique identifier	{typedef name}'%s' should not be reused. (already used as {typedef name} at %s:%d)	Warning when a typedef name is reused as another identifier name.
5.4	A tag name shall be a unique identifier	{tag name}'%s' should not be reused. (already used as {tag name} at %s:%d)	Warning when a tag name is reused as another identifier name
5.5	No object or function identifier with a static storage duration should be reused.	{static identifier/parameter name}'%s' should not be reused. (already used as {static identifier/parameter name} with static storage duration at %s:%d)	Warning when a static name is reused as another identifier name Bug Finder and Code Prover check this coding rule

N.	MISRA Definition	Messages in report file	Polyspace Specification
			differently. The analyses can produce different results.
5.6	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.	{member name}'%s' should not be reused. (already used as {member name} at %s:%d)	Warning when an `idf` in a namespace is reused in another namespace
5.7	No identifier name should be reused.	{identifier}'%s' should not be reused. (already used as {identifier} at %s:%d)	No violation reported when: • Different functions have parameters with the same name • Different functions have local variables with the same name • A function has a local variable that has the same name as a parameter of another function

Types

N.	MISRA Definition	Messages in report file	Polyspace Specification
6.1	The plain char type shall be used only for the storage and use of character values	Only permissible operators on plain chars are '=', '==' or '!=' operators, explicit casts to integral types and '?' (for the 2nd and 3rd operands)	Warning when a plain char is used with an operator other than =, ==, !=, explicit casts to integral types, or as the second or third operands of the ? operator.
6.2	Signed and unsigned char type shall be used only for the storage and use of numeric values.	• Value of type plain char is implicitly converted to signed char. • Value of type plain char is implicitly converted to unsigned char.	Warning if value of type plain char is implicitly converted to value of type signed char or unsigned char.

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• Value of type signed char is implicitly converted to plain char. • Value of type unsigned char is implicitly converted to plain char.	
6.3	*typedefs* that indicate size and signedness should be used in place of the basic types	typedefs that indicate size and signedness should be used in place of the basic types.	No warning is given in typedef definition.
6.4	Bit fields shall only be defined to be of type *unsigned int* or *signed int*.	Bit fields shall only be defined to be of type unsigned int or signed int.	
6.5	Bit fields of type *signed int* shall be at least 2 bits long.	Bit fields of type signed int shall be at least 2 bits long.	No warning on anonymous signed int bitfields of width 0 - Extended to all signed bitfields of size <= 1 (if Rule **6.4** is violated).

Constants

N.	MISRA Definition	Messages in report file	Polyspace Specification
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	• Octal constants other than zero and octal escape sequences shall not be used. • Octal constants (other than zero) should not be used. • Octal escape sequences should not be used.	

Declarations and Definitions

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	• Function XX has no complete prototype visible at call. • Function XX has no prototype visible at definition.	Prototype visible at call must be complete.
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated	Whenever an object or function is declared or defined, its type shall be explicitly stated.	
8.3	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.	Definition of function 'XX' incompatible with its declaration.	Assumes that rule 8.1 is not violated. The rule is restricted to compatible types. Can be turned to Off
8.4	If objects or functions are declared more than once their types shall be compatible.	• If objects or functions are declared more than once their types shall be compatible. • Global declaration of 'XX' function has incompatible type with its definition. • Global declaration of 'XX' variable has incompatible type with its definition.	Violations of this rule might be generated during the link phase. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.5	There shall be no definitions of objects or functions in a header file	• Object 'XX' should not be defined in a header file. • Function 'XX' should not be defined in a header file.	Tentative of definitions are considered as definitions.

N.	MISRA Definition	Messages in report file	Polyspace Specification
		• Fragment of function should not be defined in a header file.	
8.6	Functions shall always be declared at file scope.	Function 'XX' should be declared at file scope.	
8.7	Objects shall be defined at block scope if they are only accessed from within a single function	Object 'XX' should be declared at block scope.	Restricted to static objects.
8.8	An external object or function shall be declared in one file and only one file	Function/Object 'XX' has external declarations in multiples files.	Restricted to explicit extern declarations (tentative of definitions are ignored). Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.9	Definition: An identifier with external linkage shall have exactly one external definition.	• Procedure/Global variable XX multiply defined. • Forbidden multiple tentative of definition for object XX • Global variable has multiples tentative of definitions • Undefined global variable XX	Tentative of definitions are considered as definitions, no warning on predefined symbols. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required	Function/Variable XX should have internal linkage.	Assumes that 8.1 is not violated. No warning if 0 uses. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	MISRA Definition	Messages in report file	Polyspace Specification
8.11	The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage	static storage class specifier should be used on internal linkage symbol XX.	
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization	Size of array 'XX' should be explicitly stated.	

Initialization

N.	MISRA Definition	Messages in report file	Polyspace Specification
9.1	All automatic variables shall have been assigned a value before being used.		Checked during code analysis. Violations displayed as Non-initialized variable results. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
9.2	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.	
9.3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Arithmetic Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: • it is not a conversion to a wider integer type of the same signedness, or • the expression is complex, or • the expression is not constant and is a function argument, or • the expression is not constant and is a return expression	• Implicit conversion of the expression of underlying type XX to the type XX that is not a wider integer type of the same signedness. • Implicit conversion of one of the binary operands whose underlying types are XX and XX • Implicit conversion of the binary right hand operand of underlying type XX to XX that is not an integer type. • Implicit conversion of the binary left hand operand of underlying type XX to XX that is not an integer type.	ANSI C base types order (signed char, short, int, long) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. The same interpretation is applied on the unsigned version of base types. An expression of bool or enum types has int as underlying type. Plain char may have signed or unsigned underlying type (depending on Polyspace target configuration or option setting). The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not token into account and it assumes that only signed | unsigned int are used for bitfield (Rule 6.4).
10.1 (cont)		• Implicit conversion of the binary right hand operand of underlying type XX to XX that is not a wider integer type of the same signedness or Implicit conversion of the binary ? left hand operand of underlying	No violation reported when: • The implicit conversion is a type widening, without change of signedness if integer • The expression is an argument expression or a return expression

N.	MISRA Definition	Messages in report file	Polyspace Specification
		type XX to XX, but it is a complex expression. • Implicit conversion of complex integer expression of underlying type XX to XX. • Implicit conversion of non-constant integer expression of underlying type XX in function return whose expected type is XX. • Implicit conversion of non-constant integer expression of underlying type XX as argument of function whose corresponding parameter type is XX.	No violation reported when the following are all true: • Implicit conversion applies to a constant expression and is a type widening, with a possible change of signedness if integer • The conversion does not change the representation of the constant value or the result of the operation • The expression is an argument expression or a return expression or an operand expression of a non-bitwise operator

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.2	The value of an expression of floating type shall not be implicitly converted to a different type if • it is not a conversion to a wider floating type, or • the expression is complex, or • the expression is a function argument, or • the expression is a return expression	• Implicit conversion of the expression from XX to XX that is not a wider floating type. • Implicit conversion of the binary ? right hand operand from XX to XX, but it is a complex expression. • Implicit conversion of the binary ? right hand operand from XX to XX that is not a wider floating type or Implicit conversion of the binary ? left hand operand from XX to XX, but it is a complex expression. • Implicit conversion of complex floating expression from XX to XX. • Implicit conversion of floating expression of XX type in function return whose expected type is XX. • Implicit conversion of floating expression of XX type as argument of function whose corresponding parameter type is XX.	ANSI C base types order (float, double) defines that T2 is wider than T1 if T2 is on the right hand of T1 or T2 = T1. No violation reported when: • The implicit conversion is a type widening • The expression is an argument expression or a return expression.

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.3	The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression	Complex expression of underlying type XX may only be cast to narrower integer type of same signedness, however the destination type is XX.	• ANSI C base types order (signed char, short, int, long) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T1 = T2. The same methodology is applied on the unsigned version of base types. • An expression of bool or enum types has int as underlying type. • Plain char may have signed or unsigned underlying type (depending on target configuration or option setting). • The underlying type of a simple expression of struct.bitfield is the base type used in the bitfield definition, the bitfield width is not token into account and it assumes that only signed, unsigned int are used for bitfield (Rule 6.4).
10.4	The value of a complex expression of float type may only be cast to narrower floating type	Complex expression of XX type may only be cast to narrower floating type, however the destination type is XX.	ANSI C base types order (float, double) defines that T1 is narrower than T2 if T2 is on the right hand of T1 or T2 = T1.

N.	MISRA Definition	Messages in report file	Polyspace Specification
10.5	If the bitwise operator ~ and << are applied to an operand of underlying type *unsigned char* or *unsigned short*, the result shall be immediately cast to the underlying type of the operand	Bitwise [<<|~] is applied to the operand of underlying type [unsigned char| unsigned short], the result shall be immediately cast to the underlying type.	
10.6	The "U" suffix shall be applied to all constants of *unsigned* types	No explicit 'U suffix on constants of an unsigned type.	Warning when the type determined from the value and the base (octal, decimal or hexadecimal) is unsigned and there is no suffix `u` or `U`. For example, when the size of the `int` and `long int` data types is 32 bits, the coding rule checker will report a violation of rule 10.6 for the following line: `int a = 2147483648;` There is a difference between decimal and hexadecimal constants when `int` and `long int` are not the same size.

Pointer Type Conversion

N.	MISRA Definition	Messages in report file	Polyspace Specification
11.1	Conversion shall not be performed between a pointer to a function and any type other than an integral type	Conversion shall not be performed between a pointer to a function and any type other than an integral type.	Casts and implicit conversions involving a function pointer. Casts or implicit conversions from `NULL` or `(void*)0` do not give any warning.

N.	MISRA Definition	Messages in report file	Polyspace Specification
11.2	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void	Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void.	There is also a warning on qualifier loss
11.3	A cast should not be performed between a pointer type and an integral type	A cast should not be performed between a pointer type and an integral type.	Exception on zero constant. Extended to all conversions
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	A cast should not be performed between a pointer to object type and a different pointer to object type.	
11.5	A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer	A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer	Extended to all conversions

Expressions

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.1	Limited dependence should be placed on C's operator precedence rules in expressions	Limited dependence should be placed on C's operator precedence rules in expressions	
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.	• The value of 'sym' depends on the order of evaluation. • The value of volatile 'sym' depends on the order of evaluation because of multiple accesses.	The expression is a simple expression of symbols (Unlike i = i++; no detection on tab[2] = tab[2]++;). Rule 12.2 check assumes that no assignment in expressions that yield a Boolean values (rule 13.1) and the comma operator is not used (rule 12.10).

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.3	The `sizeof` operator should not be used on expressions that contain side effects.	The `sizeof` operator should not be used on expressions that contain side effects.	No warning on volatile accesses
12.4	The right hand operand of a logical && or `||` operator shall not contain side effects.	The right hand operand of a logical && or `||` operator shall not contain side effects.	No warning on volatile accesses
12.5	The operands of a logical && or `||` shall be primary-expressions.	• operand of logical && is not a primary expression • operand of logical `||` is not a primary expression • The operands of a logical && or `||` shall be primary-expressions.	During preprocessing, violations of this rule are detected on the expressions in #if directives. Allowed exception on associatively (a && b && c), (a `||` b `||` c).

N.	MISRA Definition	Messages in report file	Polyspace Specification

12.6	Operands of logical operators (&&, || and !) should be effectively Boolean. Expression that are effectively Boolean should not be used as operands to operators other than (&&, || or !).	• Operand of '!' logical operator should be effectively Boolean.
		• Left operand of '%s' logical operator should be effectively Boolean.
		• Right operand of '%s' logical operator should be effectively Boolean.
		• %s operand of '%s' is effectively Boolean. Boolean should not be used as operands to operators other than '&&', '||', '!', '=', '==', '!=' and '?:'.
			The operand of a logical operator should be a Boolean data type. Although the C standard does not explicitly define the Boolean data type, the standard implicitly assumes the use of the Boolean data type.
			
			Some operators may return Boolean-like expressions, for example, `(var == 0)`.
			
			Consider the following code:
			
			`unsigned char flag;`
			`if (!flag)`
			
			The rule checker reports a violation of rule 12.6:
			
			`Operand of '!' logical operator should be effectively Boolean.`
			The operand `flag` is not a Boolean but an `unsigned char`.
			
			To be compliant with rule 12.6, the code must be rewritten either as
			
			`if (!( flag != 0))`
			or
			
			`if (flag == 0)`
			
			The use of the option `-boolean-types` may increase or decrease the

N.	MISRA Definition	Messages in report file	Polyspace Specification
			number of warnings generated.
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed	• [~/Left Shift/Right shift/ &] operator applied on an expression whose underlying type is signed. • Bitwise ~ on operand of signed underlying type XX. • Bitwise [<<|>>] on left hand operand of signed underlying type XX. • Bitwise [& | ^] on two operands of s	The underlying type for an integer is signed when: • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
12.8	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	• shift amount is negative • shift amount is bigger than 64 • Bitwise [<< >>] count out of range [0 ..X] (width of the underlying type XX of the left hand operand - 1)..	The numbers that are manipulated in preprocessing directives are 64 bits wide so that valid shift range is between 0 and 63 Check is also extended onto bitfields with the field width or the width of the base type when it is within a complex expression
12.9	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	• Unary - on operand of unsigned underlying type XX. • Minus operator applied to an expression whose underlying type is unsigned	The underlying type for an integer is signed when: • it does not have a u or U suffix • it is small enough to fit into a 64 bits signed number
12.10	The comma operator shall not be used.	The comma operator shall not be used.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
12.11	Evaluation of constant unsigned expression should not lead to wraparound.	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

12.12	The underlying bit representations of floating-point values shall not be used.	The underlying bit representations of floating-point values shall not be used.	Warning when:
			• A float pointer is cast as a pointer to another data type. Casting a float pointer as a pointer to `void` does not generate a warning.
			• A float is packed with another data type. For example:
			`union {`
			` float f;`
			` int i;`
			`} …`

12.13	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression	Warning when ++ or -- operators are not used alone.

Control Statement Expressions

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.1	Assignment operators shall not be used in expressions that yield Boolean values.	Assignment operators shall not be used in expressions that yield Boolean values.	
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean	No warning is given on integer constants. Example: if (2) The use of the option `-boolean-types` may

N.	MISRA Definition	Messages in report file	Polyspace Specification
			increase or decrease the number of warnings generated.
13.3	Floating-point expressions shall not be tested for equality or inequality.	Floating-point expressions shall not be tested for equality or inequality.	Warning on directs tests only.
13.4	The controlling expression of a *for* statement shall not contain any objects of floating type	The controlling expression of a for statement shall not contain any objects of floating type	If *for* index is a variable symbol, checked that it is not a float.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.5	The three expressions of a *for* statement shall be concerned only with loop control	• 1st expression should be an assignment. • Bad type for loop counter (XX). • 2nd expression should be a comparison. • 2nd expression should be a comparison with loop counter (XX). • 3rd expression should be an assignment of loop counter (XX). • 3rd expression: assigned variable should be the loop counter (XX). • The following kinds of for loops are allowed: (a) all three expressions shall be present; (b) the 2nd and 3rd expressions shall be present with prior initialization of the loop counter; (c) all three expressions shall be empty for a deliberate infinite loop.	Checked if the for loop index (V) is a variable symbol; checked if V is the last assigned variable in the first expression (if present). Checked if, in first expression, if present, is assignment of V; checked if in 2nd expression, if present, must be a comparison of V; Checked if in 3rd expression, if present, must be an assignment of V.
13.6	Numeric variables being used within a *for* loop for iteration counting should not be modified in the body of the loop.	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop.	Detect only direct assignments if the for loop index is known and if it is a variable symbol.

N.	MISRA Definition	Messages in report file	Polyspace Specification
13.7	Boolean operations whose results are invariant shall not be permitted	• Boolean operations whose results are invariant shall not be permitted. Expression is always true. • Boolean operations whose results are invariant shall not be permitted. Expression is always false. • Boolean operations whose results are invariant shall not be permitted.	During compilation, check comparisons with at least one constant operand.

Control Flow

N.	MISRA Definition	Messages in report file	Polyspace Specification
14.1	There shall be no unreachable code.	There shall be no unreachable code.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
14.2	All non-null statements shall either have at lest one side effect however executed, or cause control flow to change	• All non-null statements shall either: • have at lest one side effect however executed, or • cause control flow to change	
14.3	All non-null statements shall either • have at lest one side effect however executed, or • cause control flow to change	A null statement shall appear on a line by itself	We assume that a ';' is a null statement when it is the first character on a line (excluding comments). The rule is violated when:

N.	MISRA Definition	Messages in report file	Polyspace Specification
			• there are some comments before it on the same line. • there is a comment immediately after it • there is something else than a comment after the ';' on the same line.
14.4	The *goto* statement shall not be used.	The goto statement shall not be used.	
14.5	The *continue* statement shall not be used.	The continue statement shall not be used.	
14.6	For any iteration statement there shall be at most one *break* statement used for loop termination	For any iteration statement there shall be at most one break statement used for loop termination	
14.7	A function shall have a single point of exit at the end of the function	A function shall have a single point of exit at the end of the function	
14.8	The statement forming the body of a *switch, while, do while* or *for* statement shall be a compound statement	• The body of a do while statement shall be a compound statement. • The body of a for statement shall be a compound statement. • The body of a switch statement shall be a compound statement	

N.	MISRA Definition	Messages in report file	Polyspace Specification
14.9	An *if (expression)* construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement	• An if (expression) construct shall be followed by a compound statement. • The else keyword shall be followed by either a compound statement, or another if statement	
14.10	All *if else if* constructs should contain a final *else* clause.	All if else if constructs should contain a final else clause.	

Switch Statements

N.	MISRA Definition	Messages in report file	Polyspace Specification

15.0	Unreachable code is detected between switch statement and first case.
	
	Note: This is not a MISRA C2004 rule.
		switch statements syntax normative restrictions.	Warning on declarations or any statements before the first switch case.
			
			Warning on label or jump statements in the body of switch cases.
			
			On the following example, the rule is displayed in the log file at line 3:
			
			`1 ...`
			`2 switch(index) {`
			`3  var = var + 1;`
			`// RULE 15.0`
			`// violated`
			`4case 1: ...`
			
			The code between switch statement and first case is checked as dead code by Polyspace. It follows ANSI standard behavior.

N.	MISRA Definition	Messages in report file	Polyspace Specification
15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	
15.2	An unconditional *break* statement shall terminate every non-empty switch clause	An unconditional break statement shall terminate every non-empty switch clause	Warning for each non-compliant case clause.
15.3	The final clause of a *switch* statement shall be the *default* clause	The final clause of a switch statement shall be the default clause	
15.4	A *switch* expression should not represent a value that is effectively Boolean	A switch expression should not represent a value that is effectively Boolean	The use of the option `-boolean-types` may increase the number of warnings generated.
15.5	Every *switch* statement shall have at least one *case* clause	Every switch statement shall have at least one case clause	

Functions

N.	MISRA Definition	Messages in report file	Polyspace Specification
16.1	Functions shall not be defined with variable numbers of arguments.	Function XX should not be defined as varargs.	
16.2	Functions shall not call themselves, either directly or indirectly.	Function %s should not call itself.	Done by Polyspace software (Use the call graph in Polyspace Code Prover). Polyspace also partially checks this rule during the compilation phase.
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.	Identifiers shall be given for all of the parameters in a function prototype declaration.	Assumes Rule **8.6** is not violated.

N.	MISRA Definition	Messages in report file	Polyspace Specification
16.4	The identifiers used in the declaration and definition of a function shall be identical.	The identifiers used in the declaration and definition of a function shall be identical.	Assumes that rules **8.8**, **8.1** and **16.3** are not violated. All occurrences are detected.
16.5	Functions with no parameters shall be declared with parameter type *void*.	Functions with no parameters shall be declared with parameter type void.	Definitions are also checked.
16.6	The number of arguments passed to a function shall match the number of parameters.	• Too many arguments to XX. • Insufficient number of arguments to XX.	Assumes that rule **8.1** is not violated.
16.7	A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.	Pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.	Warning if a non-`const` pointer parameter is either not used to modify the addressed object or is passed to a call of a function that is declared with a `const` pointer parameter.
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Missing return value for non-void function XX.	Warning when a non-void function is not terminated with an unconditional return with an expression.
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.	Function identifier XX should be preceded by a & or followed by a parameter list.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
16.10	If a function returns error information, then that error information shall be tested.	If a function returns error information, then that error information shall be tested.	Warning if a non-`void` function is called and the returned value is ignored. No warning if the result of the call is cast to `void`. No check performed for calls of `memcpy`, `memmove`, `memset`, `strcpy`, `strncpy`, `strcat`, or `strncat`.

Pointers and Arrays

N.	MISRA Definition	Messages in report file	Polyspace Specification
17.1	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Pointer arithmetic shall only be applied to pointers that address an array or array element.	
17.2	Pointer subtraction shall only be applied to pointers that address elements of the same array	Pointer subtraction shall only be applied to pointers that address elements of the same array.	
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	
17.4	Array indexing shall be the only allowed form of pointer arithmetic.	Array indexing shall be the only allowed form of pointer arithmetic.	Warning on operations on pointers. (`p+I`, `I+p` and `p-I`, where `p` is a pointer and `I` an integer).
17.5	A type should not contain more than 2 levels of pointer indirection	A type should not contain more than 2 levels of pointer indirection	
17.6	The address of an object with automatic storage shall not	Pointer to a parameter is an illegal return value. Pointer	Warning when assigning address to a global variable,

N.	MISRA Definition	Messages in report file	Polyspace Specification
	be assigned to an object that may persist after the object has ceased to exist.	to a local is an illegal return value.	returning a local variable address, or returning a parameter address.

Structures and Unions

N.	MISRA Definition	Messages in report file	Polyspace Specification
18.1	All structure or union types shall be complete at the end of a translation unit.	All structure or union types shall be complete at the end of a translation unit.	Warning for all incomplete declarations of structs or unions.
18.2	An object shall not be assigned to an overlapping object.	• An object shall not be assigned to an overlapping object. • Destination and source of XX overlap, the behavior is undefined.	
18.4	Unions shall not be used	Unions shall not be used.	

Preprocessing Directives

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.1	#include statements in a file shall only be preceded by other preprocessors directives or comments	#include statements in a file shall only be preceded by other preprocessors directives or comments	A message is displayed when a #include directive is preceded by other things than preprocessor directives, comments, spaces or “new lines”.
19.2	Nonstandard characters should not occur in header file names in #include directives	• A message is displayed on characters ', \, " or /* between < and > in #include <filename> • A message is displayed on characters ', \or /* between " and " in #include "filename"	

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.3	The *#include* directive shall be followed by either a <filename> or "filename" sequence.	• '#include' expects "FILENAME" or <FILENAME> • '#include_next' expects "FILENAME" or <FILENAME>	
19.4	C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Macro '<name>' does not expand to a compliant construct.	We assume that a macro definition does not violate this rule when it expands to: • a braced construct (not necessarily an initializer) • a parenthesized construct (not necessarily an expression) • a number • a character constant • a string constant (can be the result of the concatenation of string field arguments and literal strings) • the following keywords: typedef, extern, static, auto, register, const, volatile, __asm__ and __inline__ • a do-while-zero construct
19.5	Macros shall not be #defined and #undefd within a block.	• Macros shall not be `#define`'d within a block. • Macros shall not be `#undef`'d within a block.	
19.6	#undef shall not be used.	`#undef` shall not be used.	

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.7	A function should be used in preference to a function like-macro.	A function should be used in preference to a function like-macro	Message on all function-like macro definitions.
19.8	A function-like macro shall not be invoked without all of its arguments	• arguments given to macro '<name>' • macro '<name>' used without args. • macro '<name>' used with just one arg. • macro '<name>' used with too many (<number>) args.	
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Macro argument shall not look like a preprocessing directive.	This rule is detected as violated when the '#' character appears in a macro argument (outside a string or character constant)
19.10	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Parameter instance shall be enclosed in parentheses.	If `x` is a macro parameter, the following instances of x as an operand of the # and ## operators do not generate a warning: `#x`, `##x`, and `x##`. Otherwise, parentheses are required around `x`. The software does not generate a warning if a parameter is reused as an argument of a function or function-like macro. For example, consider a parameter x. The software does not generate a warning if `x` appears as `(x)` or `(x,` or `,x)` or `,x,`.

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	'<name>' is not defined.	
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	More than one occurrence of the # or ## preprocessor operators.	
19.13	The # and ## preprocessor operators should not be used	Message on definitions of macros using # or ## operators	
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	'defined' without an identifier.	

19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Precautions shall be taken in order to prevent multiple inclusions.	When a header file is formatted as,
			`#ifndef <control macro>`
			`#define <control macro>`
			`<contents> #endif`
			or,
			`#ifndef <control macro>`
			`#error ...`
			`#else`
			`#define <control macro>`
			`<contents> #endif`
			it is assumed that precautions have been taken to prevent multiple inclusions. Otherwise, a violation of this MISRA rule is detected.

N.	MISRA Definition	Messages in report file	Polyspace Specification
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	directive is not syntactically meaningful.	
19.17	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	• '#elif' not within a conditional. • '#else' not within a conditional. • '#elif' not within a conditional. • '#endif' not within a conditional. • unbalanced '#endif'. • unterminated '#if' conditional. • unterminated '#ifdef' conditional. • unterminated '#ifndef' conditional.	

Standard Libraries

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.1	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	• The macro '<name> shall not be redefined. • The macro '<name> shall not be undefined.	
20.2	The names of standard library macros, objects and functions shall not be reused.	Identifier XX should not be used.	In case a macro whose name corresponds to a standard library macro, object or function is defined, the rule that is detected as violated is **20.1**. Tentative of definitions are considered as definitions.

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.3	The validity of values passed to library functions shall be checked.	Validity of values passed to library functions shall be checked	Warning for argument in library function call if the following are all true: • Argument is a local variable • Local variable is not tested between last assignment and call to the library function • Library function is a common mathematical function • Corresponding parameter of the library function has a restricted input domain. The library function can be one of the following : `sqrt`, `tan`, `pow`, `log`, `log10`, `fmod`, `acos`, `asin`, `acosh`, `atanh`, or `atan2`.
20.4	Dynamic heap memory allocation shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the dynamic heap memory allocation functions are actually macros and the macro is expanded in the code, this rule is detected as violated. Assumes rule **20.2** is not violated.
20.5	The error indicator errno shall not be used	The error indicator errno shall not be used	Assumes that rule **20.2** is not violated
20.6	The macro *offsetof*, in library <stddef.h>, shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	Assumes that rule **20.2** is not violated

N.	MISRA Definition	Messages in report file	Polyspace Specification
20.7	The *setjmp* macro and the *longjmp* function shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the longjmp function is actually a macro and the macro is expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.8	The signal handling facilities of <signal.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case some of the signal functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.9	The input/output library <stdio.h> shall not be used in production code.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the input/output library functions are actually macros and are expanded in the code, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.10	The library functions atof, atoi and toll from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the atof, atoi and atoll functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.11	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the abort, exit, getenv and system functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated
20.12	The time handling functions of library <time.h> shall not be used.	• The macro '<name> shall not be used. • Identifier XX should not be used.	In case the time handling functions are actually macros and are expanded, this rule is detected as violated. Assumes that rule **20.2** is not violated

Runtime Failures

N.	MISRA Definition	Messages in report file	Polyspace Specification
21.1	Minimization of runtime failures shall be ensured by the use of at least one of: • static verification tools/ techniques; • dynamic verification tools/ techniques; • explicit coding of checks to handle runtime faults.		Done by Polyspace. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Unsupported MISRA C:2004 Rules

The Polyspace coding rules checker does not check the following MISRA C:2004 coding rules. These rules cannot be enforced because they are outside the scope of Polyspace software. They may concern documentation, dynamic aspects, or functional aspects of MISRA rules. The "**Polyspace Specification**" column describes the reason each rule is not checked.

Environment

Rule	Description	Polyspace Specification
1.2 (Required)	No reliance shall be placed on undefined or unspecified behavior	Not statically checkable unless the data dynamic properties is taken into account
1.3 (Required)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/ assemblers conform.	It is a process rule method.
1.4 (Required)	The compiler/linker/Identifiers (internal and external) shall not rely on significance of more than 31 characters. Furthermore the compiler/linker shall be checked to ensure that 31 character	The documentation of compiler must be checked.

Rule	Description	Polyspace Specification
	significance and case sensitivity are supported for external identifiers.	
1.5 (Advisory)	Floating point implementations should comply with a defined floating point standard.	The documentation of compiler must be checked as this implementation is done by the compiler

Language Extensions

Rule	Description	Polyspace Specification
2.4 (Advisory)	Sections of code should not be “commented out”	It might be some pseudo code or code that does not compile inside a comment.

Documentation

Rule	Description	Polyspace Specification
3.1 (Required)	All usage of implementation-defined behavior shall be documented.	The documentation of compiler must be checked. Error detection is based on undefined behavior, according to choices made for implementation- defined constructions. Documentation can not be checked.
3.2 (Required)	The character set and the corresponding encoding shall be documented.	The documentation of compiler must be checked.
3.3 (Advisory)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	The documentation of compiler must be checked.
3.5 (Required)	The implementation-defined behavior and packing of bitfields shall be documented if being relied upon.	The documentation of compiler must be checked.
3.6 (Required)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	The documentation of compiler must be checked.

Structures and Unions

Rule	Description	Polyspace Specification
18.3 (Required)	An area of memory shall not be reused for unrelated purposes.	"purpose" is functional design issue.

Polyspace MISRA C:2012 Checker

The Polyspace MISRA C:2012 checker helps you to comply with the MISRA C 2012 coding standard.[3]

When MISRA C:2012 guidelines are violated, the Polyspace MISRA C:2012 checker provides messages with information about the violated rule or directive. Most violations are found during the compile phase of an analysis.

The checker can check **138** of the **159** MISRA C:2012 guidelines.

Each guideline is categorized into one of these three categories: mandatory, required, or advisory. When you set up rule checking, you can select subsets of these categories to check. For automatically generated code, some rules change categories, including to one additional category: readability. The "Use generated code requirements (C)" option activates the categorization for automatically generated code.

There are additional subsets of MISRA C:2012 guidelines defined by Polyspace called Software Quality Objectives (SQO) that can have a direct or indirect impact on the precision of your results. When you set up checking, you can select these subsets. These subsets are defined in "Software Quality Objective Subsets (C:2012)" on page 2-54.

See Also

"Check MISRA C:2012" | "Use generated code requirements (C)"

Related Examples

- "Activate Coding Rules Checker"
- "Set Up Coding Rules Checking"

More About

- "MISRA C:2012 Directives and Rules"
- "Software Quality Objective Subsets (C:2012)" on page 2-54

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

Software Quality Objective Subsets (C:2012)

In this section...

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

Guidelines in `SQO-Subset1`

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

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

Guidelines in `SQO-Subset2`

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

Rule	Description
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage
8.11	When an array with external linkage is declared, its size should be explicitly specified
8.13	A pointer should point to a const-qualified type whenever possible
11.1	Conversions shall not be performed between a pointer to a function and any other type

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

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

See Also

"Check MISRA C:2012" | "Use generated code requirements (C)"

Related Examples

- "Activate Coding Rules Checker"

- “Set Up Coding Rules Checking”

More About

- “MISRA C:2012 Directives and Rules”

Unsupported MISRA C:2012 Guidelines

The Polyspace coding rules checker does not check the following MISRA C:2012 coding rules. These rules cannot be enforced because they are outside the scope of Polyspace software. These guidelines concern documentation, dynamic aspects, or functional aspects of MISRA rules.

Number	Category	AGC Category	Definition
Directive 1.1	Required	Required	Any implementation-defined behavior on which the output of the program depends shall be documented and understood
Directive 2.1	Required	Required	All source files shall compile without any compilation errors
Directive 3.1	Required	Required	All code shall be traceable to documented requirements
Directive 4.2	Advisory	Advisory	All usage of assembly language should be documented
Directive 4.4	Advisory	Advisory	Sections of code should not be “commented out”
Directive 4.5	Advisory	Readability	Identifiers in the same name space with overlapping visibility should be typographically unambiguous
Directive 4.7	Required	Required	If a function returns error information, then that error information shall be tested
Directive 4.8	Advisory	Advisory	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden
Directive 4.12	Required	Required	Dynamic memory allocation shall not be used
Directive 4.13	Advisory	Advisory	Functions which are designed to provide operations on a resource should be called in an appropriate sequence
Rule 2.6	Advisory	Readability	A function should not contain unused label declarations
Rule 2.7	Advisory	Readability	There should be no unused parameters in functions

Number	Category	AGC Category	Definition
Rule 17.5	Advisory	Readability	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements
Rule 17.8	Advisory	Readability	A function parameter should not be modified
Rule 21.12	Advisory	Advisory	The exception handling features of <fenv.h> should not be used.
Rule 22.1	Required	Required	All resources obtained dynamically by means of Standard Library functions shall be explicitly released
Rule 22.2	Mandatory	Mandatory	A block of memory shall only be freed if it was allocated by means of a Standard Library function
Rule 22.3	Required	Required	The same file shall not be open for read and write access at the same time on different streams
Rule 22.4	Mandatory	Mandatory	There shall be no attempt to write to a stream which has been opened as read only
Rule 22.5	Mandatory	Mandatory	A pointer to a FILE object shall not be dereferenced
Rule 22.6	Mandatory	Mandatory	The value of a pointer to a FILE shall not be used after the associated stream has been closed

Polyspace MISRA C++ Checker

The Polyspace MISRA C++ checker helps you comply with theMISRA C++:2008 coding standard.[4]

When MISRA C++ rules are violated, the Polyspace MISRA C++ checker enables Polyspace software to provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis. The MISRA C++ checker can check 185 of the 228 MISRA C++ coding rules.

There are subsets of MISRA C++ coding rules that can have a direct or indirect impact on the selectivity (reliability percentage) of your results. When you set up rule checking, you can select these subsets directly. These subsets are defined in “Software Quality Objective Subsets (C++)” on page 2-62.

Note: The Polyspace MISRA C++ checker is based on MISRA C++:2008 – “Guidelines for the use of the C++ language in critical systems." For more information on these coding standards, see http://www.misra-cpp.com.

---

4. MISRA is a registered trademark of MISRA Ltd., held on behalf of the MISRA Consortium.

Software Quality Objective Subsets (C++)

In this section...
“SQO Subset 1 – Direct Impact on Selectivity” on page 2-62
“SQO Subset 2 – Indirect Impact on Selectivity” on page 2-64

SQO Subset 1 – Direct Impact on Selectivity

The following set of coding rules will typically improve the selectivity of your results.

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	The One Definition Rule shall not be violated.
3-9-3	The underlying bit representations of floating-point values shall not be used.
5-0-15	Array indexing shall be the only form of pointer arithmetic.
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.
5-2-9	A cast should not convert a pointer type to an integral type.
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.
6-5-3	The loop-counter shall not be modified within condition or statement.
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.

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

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

SQO Subset 2 – Indirect Impact on Selectivity

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

MISRA C++ Rule	Description
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.
3-9-3	The underlying bit representations of floating-point values shall not be used.
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, `||`, !, the

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

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

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

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

MISRA C++ Coding Rules

In this section...
"Supported MISRA C++ Coding Rules" on page 2-69
"Unsupported MISRA C++ Rules" on page 2-88

Supported MISRA C++ Coding Rules

Language Independent Issues

N.	MISRA Definition	Polyspace Specification
0-1-1	A project shall not contain unreachable code.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
0-1-2	A project shall not contain infeasible paths.	
0-1-7	The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
0-1-10	Every defined function shall be called at least once.	Detects if static functions are not called in their translation unit. Other cases are detected by the software.

General

N.	MISRA Definition	Polyspace Specification
1-0-1	All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Lexical Conventions

N.	MISRA Definition	Polyspace Specification
2-3-1	Trigraphs shall not be used.	
2-5-1	Digraphs should not be used.	
2-7-1	The character sequence /* shall not be used within a C-style comment.	This rule cannot be annotated in the source code.
2-10-1	Different identifiers shall be typographically unambiguous.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-2	Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.	No detection for logical scopes: fields or member functions hiding outer scopes identifiers or hiding ancestors members.

N.	MISRA Definition	Polyspace Specification
		Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-3	A typedef name (including qualification, if any) shall be a unique identifier.	No detection across namespaces. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-4	A class, union or enum name (including qualification, if any) shall be a unique identifier.	No detection across namespaces. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-5	The identifier name of a non-member object or function with static storage duration should not be reused.	For functions the detection is only on the definition where there is a declaration. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-10-6	If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.	If the identifier is a function and the function is both declared and defined then the violation is reported only once. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
2-13-1	Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.	
2-13-2	Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.	
2-13-3	A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	
2-13-4	Literal suffixes shall be upper case.	

N.	MISRA Definition	Polyspace Specification
2-13-5	Narrow and wide string literals shall not be concatenated.	

Basic Concepts

N.	MISRA Definition	Polyspace Specification
3-1-1	It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	
3-1-2	Functions shall not be declared at block scope.	
3-1-3	When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.	
3-2-1	All declarations of an object or function shall have compatible types.	
3-2-2	The One Definition Rule shall not be violated.	Report type, template, and inline function defined in source file
3-2-3	A type, object or function that is used in multiple translation units shall be declared in one and only one file.	
3-2-4	An identifier with external linkage shall have exactly one definition.	
3-3-1	Objects or functions with external linkage shall be declared in a header file.	
3-3-2	If a function has internal linkage then all re-declarations shall include the static storage class specifier.	
3-4-1	An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	
3-9-1	The types used for an object, a function return type, or a function parameter	Comparison is done between current declaration and last seen declaration.

N.	MISRA Definition	Polyspace Specification
	shall be token-for-token identical in all declarations and re-declarations.	
3-9-2	typedefs that indicate size and signedness should be used in place of the basic numerical types.	No detection in non-instantiated templates.
3-9-3	The underlying bit representations of floating-point values shall not be used.	

Standard Conversions

N.	MISRA Definition	Polyspace Specification
4-5-1	Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.	
4-5-2	Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.	
4-5-3	Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N	

Expressions

N.	MISRA Definition	Polyspace Specification
5-0-1	The value of an expression shall be the same under any order of evaluation that the standard permits.	

N.	MISRA Definition	Polyspace Specification
5-0-2	Limited dependence should be placed on C++ operator precedence rules in expressions.	
5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type.	Assumes that `ptrdiff_t` is signed integer
5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.	Assumes that `ptrdiff_t` is signed integer If the conversion is to a narrower integer with a different sign then MISRA C++ 5-0-4 takes precedence over MISRA C++ 5-0-6.
5-0-5	There shall be no implicit floating-integral conversions.	This rule takes precedence over 5-0-4 and 5-0-6 if they apply at the same time.
5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	If the conversion is to a narrower integer with a different sign then MISRA C++ 5-0-4 takes precedence over MISRA C++ 5-0-6.
5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.	
5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	
5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	
5-0-10	If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	
5-0-11	The plain char type shall only be used for the storage and use of character values.	For numeric data, use a type which has explicit signedness.
5-0-12	Signed char and unsigned char type shall only be used for the storage and use of numeric values.	
5-0-14	The first operand of a conditional-operator shall have type bool.	

N.	MISRA Definition	Polyspace Specification
5-0-15	Array indexing shall be the only form of pointer arithmetic.	Warning on operations on pointers. (p+I, I+p and p-I, where p is a pointer and I an integer, p[i] accepted).
5-0-18	>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	Report when relational operator are used on pointers types (casts ignored).
5-0-19	The declaration of objects shall contain no more than two levels of pointer indirection.	
5-0-20	Non-constant operands to a binary bitwise operator shall have the same underlying type.	
5-0-21	Bitwise operators shall only be applied to operands of unsigned underlying type.	
5-2-1	Each operand of a logical && or || shall be a postfix - expression.	During preprocessing, violations of this rule are detected on the expressions in #if directives. Allowed exception on associativity (a && b && c), (a || b || c).
5-2-2	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.	
5-2-3	Casts from a base class to a derived class should not be performed on polymorphic types.	
5-2-4	C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.	
5-2-5	A cast shall not remove any const or volatile qualification from the type of a pointer or reference.	
5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.	No violation if pointer types of operand and target are identical.

N.	MISRA Definition	Polyspace Specification
5-2-7	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.	"Extended to all pointer conversions including between pointer to struct object and pointer to type of the first member of the struct type. Indirect conversions through non-pointer type (e.g. int) are not detected."
5-2-8	An object with integer type or pointer to void type shall not be converted to an object with pointer type.	Exception on zero constants. Objects with pointer type include objects with pointer to function type.
5-2-9	A cast should not convert a pointer type to an integral type.	
5-2-10	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	
5-2-11	The comma operator, && operator and the || operator shall not be overloaded.	
5-2-12	An identifier with array type passed as a function argument shall not decay to a pointer.	
5-3-1	Each operand of the ! operator, the logical && or the logical || operators shall have type bool.	
5-3-2	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	
5-3-3	The unary & operator shall not be overloaded.	
5-3-4	Evaluation of the operand to the sizeof operator shall not contain side effects.	No warning on volatile accesses and function calls
5-8-1	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	

N.	MISRA Definition	Polyspace Specification
5-14-1	The right hand operand of a logical && or || operator shall not contain side effects.	No warning on volatile accesses and function calls.
5-18-1	The comma operator shall not be used.	
5-19-1	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	

Statements

N.	MISRA Definition	Polyspace Specification
6-2-1	Assignment operators shall not be used in sub-expressions.	
6-2-2	Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	
6-2-3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.	
6-3-1	The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	
6-4-1	An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	
6-4-2	All if ... else if constructs shall be terminated with an else clause.	Also detects cases where the last `if` is in the block of the last `else` (same behavior as JSF, stricter than MISRA C). Example: "`if` ... `else` { `if` ...{}}" raises the rule

N.	MISRA Definition	Polyspace Specification
6-4-3	A switch statement shall be a well-formed switch statement.	Return statements are considered as jump statements.
6-4-4	A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	
6-4-5	An unconditional throw or break statement shall terminate every non - empty switch-clause.	
6-4-6	The final clause of a switch statement shall be the default-clause.	
6-4-7	The condition of a switch statement shall not have bool type.	
6-4-8	Every switch statement shall have at least one case-clause.	
6-5-1	A for loop shall contain a single loop-counter which shall not have floating type.	
6-5-2	If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.	
6-5-3	The loop-counter shall not be modified within condition or statement.	Detect only direct assignments if for_index is known (see 6-5-1).
6-5-4	The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.	
6-5-5	A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	
6-5-6	A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	

N.	MISRA Definition	Polyspace Specification
6-6-1	Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.	
6-6-2	The goto statement shall jump to a label declared later in the same function body.	
6-6-3	The continue statement shall only be used within a well-formed for loop.	Assumes 6.5.1 to 6.5.6: so it is implemented only for supported 6_5_x rules.
6-6-4	For any iteration statement there shall be no more than one break or goto statement used for loop termination.	
6-6-5	A function shall have a single point of exit at the end of the function.	At most one return not necessarily as last statement for void functions.

Declarations

N.	MISRA Definition	Polyspace Specification
7-3-1	The global namespace shall only contain main, namespace declarations and extern "C" declarations.	
7-3-2	The identifier main shall not be used for a function other than the global function main.	
7-3-3	There shall be no unnamed namespaces in header files.	
7-3-4	using-directives shall not be used.	
7-3-5	Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.	
7-3-6	using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	

N.	MISRA Definition	Polyspace Specification
7-4-2	Assembler instructions shall only be introduced using the asm declaration.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
7-4-3	Assembly language shall be encapsulated and isolated.	
7-5-1	A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	
7-5-2	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	
7-5-3	A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.	
7-5-4	Functions should not call themselves, either directly or indirectly.	

Declarators

N.	MISRA Definition	Polyspace Specification
8-0-1	An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.	
8-3-1	Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.	
8-4-1	Functions shall not be defined using the ellipsis notation.	
8-4-2	The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	

N.	MISRA Definition	Polyspace Specification
8-4-3	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
8-4-4	A function identifier shall either be used to call the function or it shall be preceded by &.	
8-5-1	All variables shall have a defined value before they are used.	Non-initialized variable in results and error messages for obvious cases
8-5-2	Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.	
8-5-3	In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	

Classes

N.	MISRA Definition	Polyspace Specification
9-3-1	const member functions shall not return non-const pointers or references to class-data.	Class-data for a class is restricted to all non-static member data.
9-3-2	Member functions shall not return non-const handles to class-data.	Class-data for a class is restricted to all non-static member data.
9-5-1	Unions shall not be used.	
9-6-2	Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.	
9-6-3	Bit-fields shall not have enum type.	
9-6-4	Named bit-fields with signed integer type shall have a length of more than one bit.	

Derived Classes

N.	MISRA Definition	Polyspace Specification
10-1-1	Classes should not be derived from virtual bases.	
10-1-2	A base class shall only be declared virtual if it is used in a diamond hierarchy.	Assumes 10.1.1 not required
10-1-3	An accessible base class shall not be both virtual and nonvirtual in the same hierarchy.	
10-2-1	All accessible entity names within a multiple inheritance hierarchy should be unique.	No detection between entities of different kinds (member functions against data members, ...).
10-3-1	There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.	Member functions that are virtual by inheritance are also detected.
10-3-2	Each overriding virtual function shall be declared with the virtual keyword.	
10-3-3	A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.	

Member Access Control

N.	MISRA Definition	Polyspace Specification
11-0-1	Member data in non- POD class types shall be private.	

Special Member Functions

N.	MISRA Definition	Polyspace Specification
12-1-1	An object's dynamic type shall not be used from the body of its constructor or destructor.	
12-1-2	All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.	

N.	MISRA Definition	Polyspace Specification
12-1-3	All constructors that are callable with a single argument of fundamental type shall be declared explicit.	
12-8-1	A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.	
12-8-2	The copy assignment operator shall be declared protected or private in an abstract class.	

Templates

N.	MISRA Definition	Polyspace Specification
14-5-2	A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.	
14-5-3	A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	
14-6-1	In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->	
14-6-2	The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	
14-7-3	All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.	
14-8-1	Overloaded function templates shall not be explicitly specialized.	All specializations of overloaded templates are rejected even if overloading occurs after the call.

N.	MISRA Definition	Polyspace Specification
		Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
14-8-2	The viable function set for a function call should either contain no function specializations, or only contain function specializations.	

Exception Handling

N.	MISRA Definition	Polyspace Specification
15-0-2	An exception object should not have pointer type.	NULL not detected (see 15-1-2).
15-0-3	Control shall not be transferred into a try or catch block using a goto or a switch statement.	
15-1-2	NULL shall not be thrown explicitly.	
15-1-3	An empty throw (throw;) shall only be used in the compound- statement of a catch handler.	
15-3-2	There should be at least one exception handler to catch all otherwise unhandled exceptions.	Detect that there is no try/catch in the main and that the catch does not handle all exceptions. Not detected if no "main". Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
15-3-3	Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.	
15-3-5	A class type exception shall always be caught by reference.	
15-3-6	Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all	

N.	MISRA Definition	Polyspace Specification
	of its bases, the handlers shall be ordered most-derived to base class.	
15-3-7	Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.	
15-4-1	If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.	
15-5-1	A class destructor shall not exit with an exception.	Limit detection to throw and catch that are internals to the destructor; rethrows are partially processed; no detections in nested handlers.
15-5-2	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).	Limit detection to throw that are internals to the function; rethrows are partially processed; no detections in nested handlers.

Preprocessing Directives

N.	MISRA Definition	Polyspace Specification
16-0-1	#include directives in a file shall only be preceded by other preprocessor directives or comments.	
16-0-2	Macros shall only be #define 'd or #undef 'd in the global namespace.	
16-0-3	#undef shall not be used.	
16-0-4	Function-like macros shall not be defined.	
16-0-5	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	
16-0-6	In the definition of a function-like macro, each instance of a parameter shall be	

N.	MISRA Definition	Polyspace Specification
	enclosed in parentheses, unless it is used as the operand of # or ##.	
16-0-7	Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.	
16-0-8	If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.	
16-1-1	The defined preprocessor operator shall only be used in one of the two standard forms.	
16-1-2	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	
16-2-1	The preprocessor shall only be used for file inclusion and include guards.	The rule is raised for #ifdef/#define if the file is not an include file.
16-2-2	C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.	
16-2-3	Include guards shall be provided.	
16-2-4	The ', ", /* or // characters shall not occur in a header file name.	
16-2-5	The \ character should not occur in a header file name.	
16-2-6	The #include directive shall be followed by either a <filename> or "filename" sequence.	
16-3-1	There shall be at most one occurrence of the # or ## operators in a single macro definition.	
16-3-2	The # and ## operators should not be used.	

Library Introduction

N.	MISRA Definition	Polyspace Specification
17-0-1	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
17-0-2	The names of standard library macros and objects shall not be reused.	
17-0-5	The setjmp macro and the longjmp function shall not be used.	

Language Support Library

N.	MISRA Definition	Polyspace Specification
18-0-1	The C library shall not be used.	
18-0-2	The library functions atof, atoi and atol from library <cstdlib> shall not be used.	
18-0-3	The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	The option `-dialect iso` must be used to detect violations, for example, `exit`.
18-0-4	The time handling functions of library <ctime> shall not be used.	
18-0-5	The unbounded functions of library <cstring> shall not be used.	
18-2-1	The macro offsetof shall not be used.	
18-4-1	Dynamic heap memory allocation shall not be used.	
18-7-1	The signal handling facilities of <csignal> shall not be used.	

Diagnostic Library

N.	MISRA Definition	Polyspace Specification
19-3-1	The error indicator errno shall not be used.	

Input/output Library

N.	MISRA Definition	Polyspace Specification
27-0-1	The stream input/output library <cstdio> shall not be used.	

Unsupported MISRA C++ Rules

Language Independent Issues

N.	MISRA Definition	Polyspace Specification
0–1–3	A project shall not contain unused variables.	
0-1-4	A project shall not contain non-volatile POD variables having only one use.	
0-1-5	A project shall not contain unused type declarations.	
0-1-6	A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	

N.	MISRA Definition	Polyspace Specification
0-1-8	All functions with void return type shall have external side effects.	
0-1-9	There shall be no dead code.	Not checked by the coding rules checker. Can be enforced through detection of dead code during analysis.
0-1-11	There shall be no unused parameters (named or unnamed) in nonvirtual functions.	
0-1-12	There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.	
0-2-1	An object shall not be assigned to an overlapping object.	
0-3-1	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	
0-3-2	If a function generates error information, then that error information shall be tested.	
0-4-1	Use of scaled-integer or fixed-point arithmetic shall be documented.	
0-4-2	Use of floating-point arithmetic shall be documented.	
0-4-3	Floating-point implementations shall comply with a defined floating-point standard.	

General

N.	MISRA Definition	Polyspace Specification
1-0-2	Multiple compilers shall only be used if they have a common, defined interface.	

N.	MISRA Definition	Polyspace Specification
1-0-3	The implementation of integer division in the chosen compiler shall be determined and documented.	

Lexical Conventions

N.	MISRA Definition	Polyspace Specification
2-2-1	The character set and the corresponding encoding shall be documented.	
2-7-2	Sections of code shall not be "commented out" using C-style comments.	
2-7-3	Sections of code should not be "commented out" using C++ comments.	

Standard Conversions

N.	MISRA Definition	Polyspace Specification
4-10-1	ULL shall not be used as an integer value.	
4-10-2	Literal zero (0) shall not be used as the null-pointer-constant.	

Expressions

N.	MISRA Definition	Polyspace Specification
5-0-13	The condition of an if-statement and the condition of an iteration- statement shall have type bool.	
5-0-16	A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	
5-0-17	Subtraction between pointers shall only be applied to pointers that address elements of the same array.	

N.	MISRA Definition	Polyspace Specification
5-17-1	The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	

Declarations

N.	MISRA Definition	Polyspace Specification
7-1-1	A variable which is not modified shall be const qualified.	
7-1-2	A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	
7-2-1	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	
7-4-1	All usage of assembler shall be documented.	

Classes

N.	MISRA Definition	Polyspace Specification
9-3-3	If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	
9-6-1	When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.	

Templates

N.	MISRA Definition	Polyspace Specification
14-5-1	A non-member generic function shall only be declared in a namespace that is not an associated namespace.	

N.	MISRA Definition	Polyspace Specification
14-7-1	All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	
14-7-2	For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.	

Exception Handling

N.	MISRA Definition	Polyspace Specification
15-0-1	Exceptions shall only be used for error handling.	
15-1-1	The assignment-expression of a throw statement shall not itself cause an exception to be thrown.	
15-3-1	Exceptions shall be raised only after start-up and before termination of the program.	
15-3-4	Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	
15-5-3	The terminate() function shall not be called implicitly.	

Preprocessing Directives

N.	MISRA Definition	Polyspace Specification
16-6-1	All uses of the #pragma directive shall be documented.	

Library Introduction

N.	MISRA Definition	Polyspace Specification
17-0-3	The names of standard library functions shall not be overridden.	
17-0-4	All library code shall conform to MISRA C++.	

Polyspace JSF C++ Checker

The Polyspace JSF C++ checker helps you comply with the Joint Strike Fighter® Air Vehicle C++ coding standards (JSF++). These coding standards were developed by Lockheed Martin® for the Joint Strike Fighter program. They are designed to improve the robustness of C++ code, and improve maintainability.

[5]

When JSF++ rules are violated, the Polyspace JSF C++ checker enables Polyspace software to provide messages with information about the rule violations. Most messages are reported during the compile phase of an analysis.

Note: The Polyspace JSF C++ checker is based on JSF++:2005. For more information on these coding standards, see Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program.

5. JSF and Joint Strike Fighter are registered trademarks of Lockheed Martin.

JSF C++ Coding Rules

In this section...
"Supported JSF C++ Coding Rules" on page 2-95
"Unsupported JSF++ Rules" on page 2-118

Supported JSF C++ Coding Rules

- "Fault Handling" on page 2-117
- "Portable Code" on page 2-118

Code Size and Complexity

N.	JSF++ Definition	Polyspace Specification
1	Any one function (or method) **will** contain no more than 200 logical source lines of code (L-SLOCs).	Message in report file: `<function name>` has `<num>` logical source lines of code.
3	All functions **shall** have a cyclomatic complexity number of 20 or less.	Message in report file: `<function name>` has cyclomatic complexity number equal to `<num>`.

Environment

N.	JSF++ Definition	Polyspace Specification
8	All code **shall** conform to ISO/IEC 14882:2002(E) standard C++.	Reports the compilation error message
9	Only those characters specified in the C++ basic source character set **will** be used.	
11	Trigraphs **will not** be used.	
12	The following digraphs **will not** be used: `<%`, `%>`, `<:`, `:>`, `%:`, `%:%:`.	Message in report file: The following digraph will not be used: `<digraph>`. Reports the digraph. If the rule level is set to warning, the digraph will be allowed even if it is not supported in `-dialect iso`.
13	Multi-byte characters and wide string literals **will not** be used.	Report `L'c'`, `L"string"`, and use of `wchar_t`.
14	Literal suffixes **shall** use uppercase rather than lowercase letters.	
15	Provision **shall** be made for run-time checking (defensive programming).	Done with checks in the software.

Libraries

N.	JSF++ Definition	Polyspace Specification
17	The error indicator `errno` **shall not** be used.	`errno` should not be used as a macro or a global with external "C" linkage.
18	The macro `offsetof`, in library `<stddef.h>`, **shall not** be used.	`offsetof` should not be used as a macro or a global with external "C" linkage.
19	`<locale.h>` and the `setlocale` function **shall not** be used.	`setlocale` and `localeconv` should not be used as a macro or a global with external "C" linkage.
20	The `setjmp` macro and the `longjmp` function **shall not** be used.	`setjmp` and `longjmp` should not be used as a macro or a global with external "C" linkage.
21	The signal handling facilities of `<signal.h>` **shall not** be used.	`signal` and `raise` should not be used as a macro or a global with external "C" linkage.
22	The input/output library `<stdio.h>` **shall not** be used.	all standard functions of `<stdio.h>` should not be used as a macro or a global with external "C" linkage.
23	The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` **shall not** be used.	`atof`, `atoi` and `atol` should not be used as a macro or a global with external "C" linkage.
24	The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` **shall not** be used.	`abort`, `exit`, `getenv` and `system` should not be used as a macro or a global with external "C" linkage.
25	The time handling functions of library `<time.h>` **shall not** be used.	`clock`, `difftime`, `mktime`, `asctime`, `ctime`, `gmtime`, `localtime` and `strftime` should not be used as a macro or a global with external "C" linkage.

Pre-Processing Directives

N.	JSF++ Definition	Polyspace Specification
26	Only the following preprocessor directives **shall** be used: `#ifndef`, `#define`, `#endif`, `#include`.	

N.	JSF++ Definition	Polyspace Specification
27	`#ifndef`, `#define` and `#endif` **will** be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files **will not** be used.	Detects the patterns `#if !defined`, `#pragma once`, `#ifdef`, and missing `#define`.
28	The `#ifndef` and `#endif` preprocessor directives **will** only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	Detects any use that does not comply with AV Rule 27. Assuming 35/27 is not violated, reports only `#ifndef`.
29	The `#define` preprocessor directive **shall not** be used to create inline macros. Inline functions shall be used instead.	Rule is split into two parts: the definition of a macro function (29.def) and the call of a macrofunction (29.use). Messages in report file: • 29.1 : The `#define` preprocessor directive shall not be used to create inline macros. • 29.2 : Inline functions shall be used instead of inline macros.
30	The `#define` preprocessor directive **shall not** be used to define constant values. Instead, the `const` qualifier **shall** be applied to variable declarations to specify constant values.	Reports `#define` of simple constants.
31	The `#define` preprocessor directive **will** only be used as part of the technique to prevent multiple inclusions of the same header file.	Detects use of `#define` that are not used to guard for multiple inclusion, assuming that rules 35 and 27 are not violated.
32	The `#include` preprocessor directive **will** only be used to include header (*.h) files.	

Header Files

N.	JSF++ Definition	Polyspace Specification
33	The `#include` directive **shall** use the `<filename.h>` notation to include header files.	
35	A header file **will** contain a mechanism that prevents multiple inclusions of itself.	
39	Header files (`*.h`) **will not** contain non-const variable definitions or function definitions.	Reports definitions of global variables / function in header.

Style

N.	JSF++ Definition	Polyspace Specification
40	Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.	Reports when type, template, or inline function is defined in source file. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
41	Source lines **will** be kept to a length of 120 characters or less.	
42	Each expression-statement **will** be on a separate line.	Reports when two consecutive expression statements are on the same line.
43	Tabs **should** be avoided.	
44	All indentations will be at least two spaces and be consistent within the same source file.	Reports when a statement indentation is not at least two spaces more than the statement containing it. Does not report bad indentation between opening braces following if/else, do/while, for, and while statements. NB: in final release it will accept any indentation
46	User-specified identifiers (internal and external) **will not** rely on significance of more than 64 characters.	

N.	JSF++ Definition	Polyspace Specification
47	Identifiers **will not** begin with the underscore character '_'.	
48	Identifiers **will not** differ by: • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O'; with the number '0' or the letter 'D' • The interchange of the letter 'I'; with the number '1' or the letter 'l' • The interchange of the letter 'S' with the number '5' • The interchange of the letter 'Z' with the number 2 • The interchange of the letter 'n' with the letter 'h'	Checked regardless of scope. Not checked between macros and other identifiers. Messages in report file: • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by the presence/absence of the underscore character. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by a mixture of case. • Identifier `Idf1` (*`file1.cpp line l1 column c1`*) and `Idf2` (*`file2.cpp line l2 column c2`*) only differ by letter `O`, with the number `0`.
50	The first word of the name of a class, structure, namespace, enumeration, or type created with `typedef` **will** begin with an uppercase letter. All others letters **will** be lowercase.	Messages in report file: • The first word of the name of a class will begin with an uppercase letter. • The first word of the namespace of a class will begin with an uppercase letter. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	JSF++ Definition	Polyspace Specification
51	All letters contained in function and variables names **will** be composed entirely of lowercase letters.	Messages in report file: • All letters contained in variable names will be composed entirely of lowercase letters. • All letters contained in function names will be composed entirely of lowercase letters.
52	Identifiers for constant and enumerator values **shall** be lowercase.	Messages in report file: • Identifier for enumerator value shall be lowercase. • Identifier for template constant parameter shall be lowercase.
53	Header files **will** always have file name extension of "`.h`".	`.H` is allowed if you set the option `-dos`.
53.1	The following character sequences **shall** not appear in header file names: `'`, `\`, `/*`, `//`, or `"`.	
54	Implementation files **will** always have a file name extension of ".cpp".	Not case sensitive if you set the option `-dos`.
57	The public, protected, and private sections of a class **will** be declared in that order.	
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument **will** be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	Detects that two parameters are not on the same line, The first parameter should be on the same line as function name. Does not check for the closing parenthesis.

N.	JSF++ Definition	Polyspace Specification
59	The statements forming the body of an if, else if, else, while, do ... while or for statement **shall** always be enclosed in braces, even if the braces form an empty block.	Messages in report file: • The statements forming the body of an if statement shall always be enclosed in braces. • The statements forming the body of an else statement shall always be enclosed in braces. • The statements forming the body of a while statement shall always be enclosed in braces. • The statements forming the body of a do ... while statement shall always be enclosed in braces. • The statements forming the body of a for statement shall always be enclosed in braces.
60	Braces ("{}") which enclose a block **will** be placed in the same column, on separate lines directly before and after the block.	Detects that statement-block braces should be in the same columns.
61	Braces ("{}") which enclose a block **will** have nothing else on the line except comments.	
62	The dereference operator '*' and the address-of operator '&' will be directly connected with the type-specifier.	Reports when there is a space between type and "*" "&" for variables, parameters and fields declaration.

N.	JSF++ Definition	Polyspace Specification
63	Spaces will not be used around '.' or '->', nor between unary operators and operands.	Reports when the following characters are not directly connected to a white space: • . • -> • ! • ~ • - • ++ • — **Note:** A violation will be reported for "." used in float/double definition.

Classes

N.	JSF++ Definition	Polyspace Specification
67	Public and protected data **should** only be used in structs - not classes.	
68	Unneeded implicitly generated member functions shall be explicitly disallowed.	Reports when default constructor, assignment operator, copy constructor or destructor is not declared.
71.1	A class's virtual functions shall not be invoked from its destructor or any of its constructors.	Reports when a constructor or destructor directly calls a virtual function.
74	Initialization of nonstatic class members **will** be performed through the member initialization list rather than through assignment in the body of a constructor.	All data should be initialized in the initialization list except for array. Does not report that an assignment exists in `ctor` body. Message in report file: Initialization of nonstatic class members "`<field>`" will be performed through the member initialization list.

N.	JSF++ Definition	Polyspace Specification
75	Members of the initialization list **shall** be listed in the order in which they are declared in the class.	
76	A copy constructor and an assignment operator **shall** be declared for classes that contain pointers to data items or nontrivial destructors.	Messages in report file: • `no copy constructor and no copy assign` • `no copy constructor` • `no copy assign` Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
77.1	The definition of a member function **shall not** contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	Does not report when an explicit copy constructor exists.
78	All base classes with a virtual function **shall** define a virtual destructor.	
79	All resources acquired by a class shall be released by the class's destructor.	Reports when the number of "new" called in a constructor is greater than the number of "delete" called in its destructor. **Note:** A violation is raised even if "new" is done in a "if/else".

N.	JSF++ Definition	Polyspace Specification
81	The assignment operator shall handle self-assignment correctly	Reports when copy assignment body does not begin with "`if (this != arg)`" A violation is not raised if an empty `else` statement follows the `if`, or the body contains only a return statement. A violation is raised when the `if` statement is followed by a statement other than the return statement.

82	An assignment operator **shall** return a reference to `*this`.	The following operators should return `*this` on method, and `*first_arg` on plain function.
		
		`operator=`
		`operator+=`
		`operator-=`
		`operator*=`
		`operator >>=`
		`operator <<=`
		`operator /=`
		`operator %=`
		`operator |=`
		`operator &=`
		`operator ^=`
		`prefix operator++`
		`prefix operator--`
		
		Does not report when no return exists.
		
		No special message if type does not match.
		
		Messages in report file:
		
		• An assignment operator shall return a reference to `*this`.
		• An assignment operator shall return a reference to its first arg.

N.	JSF++ Definition	Polyspace Specification
83	An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	Reports when a copy assignment does not assign all data members. In a derived class, it also reports when a copy assignment does not call inherited copy assignments.
88	Multiple inheritance **shall** only be allowed in the following restricted form: `n` interfaces plus `m` private implementations, plus at most one protected implementation.	Messages in report file: • Multiple inheritance on public implementation shall not be allowed: *`<public_base_class>`* is not an interface. • Multiple inheritance on protected implementation shall not be allowed : *`<protected_base_class_1>`*. • *`<protected_base_class_2>`* are not interfaces.
88.1	A stateful virtual base **shall** be explicitly declared in each derived class that accesses it.	
89	A base class **shall not** be both virtual and nonvirtual in the same hierarchy.	
94	An inherited nonvirtual function **shall not** be redefined in a derived class.	Does not report for destructor. Message in report file: Inherited nonvirtual function `%s` shall not be redefined in a derived class.
95	An inherited default parameter **shall never** be redefined.	
96	Arrays **shall not** be treated polymorphically.	Reports pointer arithmetic and array like access on expressions whose pointed type is used as a base class.
97	Arrays **shall not** be used in interface.	Only to prevent array-to-pointer-decay. Not checked on private methods

N.	JSF++ Definition	Polyspace Specification
97.1	Neither operand of an equality operator (== or `!=`) **shall** be a pointer to a virtual member function.	Reports == and `!=` on pointer to member function of polymorphic classes (cannot determine statically if it is virtual or not), except when one argument is the null constant.

Namespaces

N.	JSF++ Definition	Polyspace Specification
98	Every nonlocal name, except `main()`, **should** be placed in some namespace.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
99	Namespaces **will not** be nested more than two levels deep.	

Templates

N.	JSF++ Definition	Polyspace Specification
104	A template specialization **shall** be declared before its use.	Reports the actual compilation error message.

Functions

N.	JSF++ Definition	Polyspace Specification
107	Functions **shall** always be declared at file scope.	
108	Functions with variable numbers of arguments **shall not** be used.	
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	Reports when "inline" is not in the definition of a member function inside the class definition.
110	Functions with more than 7 arguments **will not** be used.	
111	A function **shall not** return a pointer or reference to a non-static local object.	Simple cases without alias effect detected.

N.	JSF++ Definition	Polyspace Specification
113	Functions **will** have a single exit point.	Reports first return, or once per function.
114	All exit points of value-returning functions **shall** be through return statements.	
116	Small, concrete-type arguments (two or three words in size) **should** be passed by value if changes made to formal parameters should not be reflected in the calling function.	Report constant parameters references with `sizeof <= 2 * sizeof(int)`. Does not report for copy-constructor.
119	Functions **shall** not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	Direct recursion is reported statically. Indirect recursion reported through the software. Message in report file: Function <*F*> shall not call directly itself.
121	Only functions with 1 or 2 statements **should** be considered candidates for inline functions.	Reports inline functions with more than 2 statements.

Comments

N.	JSF++ Definition	Polyspace Specification
126	Only valid C++ style comments (//) **shall** be used.	
133	Every source file will be documented with an introductory comment that provides information on the file name, its contents, and any program-required information (e.g. legal statements, copyright information, etc).	Reports when a file does not begin with two comment lines. **Note**: This rule cannot be annotated in the source code.

Declarations and Definitions

N.	JSF++ Definition	Polyspace Specification
135	Identifiers in an inner scope **shall not** use the same name as an identifier in an outer scope, and therefore hide that identifier.	Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

N.	JSF++ Definition	Polyspace Specification
136	Declarations should be at the smallest feasible scope.	Reports when: • A global variable is used in only one function. • A local variable is not used in a statement (`expr`, `return`, `init` ...) of the same level of its declaration (in the same block) or is not used in two sub-statements of its declaration. **Note:** • Non-used variables are reported. • Initializations at definition are ignored (not considered an access)
137	All declarations at file scope should be static where possible.	
138	Identifiers **shall not** simultaneously have both internal and external linkage in the same translation unit.	
139	External objects will not be declared in more than one file.	Reports all duplicate declarations inside a translation unit. Reports when the declaration localization is not the same in all translation units.
140	The register storage class specifier **shall not** be used.	
141	A class, structure, or enumeration **will not** be declared in the definition of its type.	

Initialization

N.	JSF++ Definition	Polyspace Specification
142	All variables **shall** be initialized before use.	Done with Non-initialized variable checks in the software.

N.	JSF++ Definition	Polyspace Specification
144	Braces **shall** be used to indicate and match the structure in the non-zero initialization of arrays and structures.	This covers partial initialization.
145	In an enumerator list, the '=' construct **shall not** be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Generates one report for an enumerator list.

Types

N.	JSF++ Definition	Polyspace Specification
147	The underlying bit representations of floating point numbers **shall not** be used in any way by the programmer.	Reports on casts with float pointers (except with `void*`).
148	Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	Reports when non enumeration types are used in switches.

Constants

N.	JSF++ Definition	Polyspace Specification
149	Octal constants (other than zero) **shall not** be used.	
150	Hexadecimal constants **will** be represented using all uppercase letters.	
151	Numeric values in code **will not** be used; symbolic values will be used instead.	Reports direct numeric constants (except integer/float value `1, 0`) in expressions, non `-const` initializations. and switch cases. char constants are allowed. Does not report on templates non-type parameter. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
151.1	A string literal shall not be modified.	Report when a `char*`, `char[]`, or `string` type is used not as `const`.

N.	JSF++ Definition	Polyspace Specification
		A violation is raised if a string literal (for example, " ") is cast as a non `const`.

Variables

N.	JSF++ Definition	Polyspace Specification
152	Multiple variable declarations **shall not** be allowed on the same line.	

Unions and Bit Fields

N.	JSF++ Definition	Polyspace Specification
153	Unions **shall not** be used.	
154	Bit-fields **shall** have explicitly unsigned integral or enumeration types only.	
156	All the members of a structure (or class) **shall** be named and shall only be accessed via their names.	Reports unnamed bit-fields (unnamed fields are not allowed).

Operators

N.	JSF++ Definition	Polyspace Specification
157	The right hand operand of a && or || operator shall not contain side effects.	Assumes rule 159 is not violated. Messages in report file: • The right hand operand of a && operator shall not contain side effects. • The right hand operand of a || operator shall not contain side effects.
158	The operands of a logical && or || **shall** be parenthesized if the operands contain binary operators.	Messages in report file: • The operands of a logical && shall be parenthesized if the operands contain binary operators. • The operands of a logical || shall be parenthesized if the operands contain binary operators.

N.	JSF++ Definition	Polyspace Specification
		Exception for: `X || Y || Z , Z&&Y &&Z`
159	Operators ||, &&, and unary & **shall not** be overloaded.	Messages in report file: • Unary operator & shall not be overloaded. • Operator || shall not be overloaded. • Operator && shall not be overloaded.
160	An assignment expression **shall** be used only as the expression in an expression statement.	Only simple assignment, not +=, ++, etc.
162	Signed and unsigned values **shall not** be mixed in arithmetic or comparison operations.	
163	Unsigned arithmetic **shall not** be used.	
164	The right hand operand of a shift operator **shall** lie between zero and one less than the width in bits of the left-hand operand (inclusive).	
164.1	The left-hand operand of a right-shift operator **shall not** have a negative value.	Detects constant case +. Found by the software for dynamic cases.
165	The unary minus operator **shall not** be applied to an unsigned expression.	
166	The `sizeof` operator **will not** be used on expressions that contain side effects.	
168	The comma operator **shall not** be used.	

Pointers and References

N.	JSF++ Definition	Polyspace Specification
169	Pointers to pointers should be avoided when possible.	Reports second-level pointers, except for arguments of main.
170	More than 2 levels of pointer indirection **shall not** be used.	Only reports on variables/parameters.

N.	JSF++ Definition	Polyspace Specification
171	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to: • the same object, • the same function, • members of the same object, or • elements of the same array (including one past the end of the same array).	Reports when relational operator are used on pointer types (casts ignored).
173	The address of an object with automatic storage **shall not** be assigned to an object which persists after the object has ceased to exist.	
174	The null pointer **shall not** be de-referenced.	Done with checks in software.
175	A pointer **shall not** be compared to `NULL` or be assigned `NULL`; use plain `0` instead.	Reports usage of `NULL` macro in pointer contexts.
176	A `typedef` **will** be used to simplify program syntax when declaring function pointers.	Reports non-`typedef` function pointers, or pointers to member functions for types of variables, fields, parameters. Returns type of function, cast, and exception specification.

Type Conversions

N.	JSF++ Definition	Polyspace Specification
177	User-defined conversion functions **should** be avoided.	Reports user defined conversion function, non-explicit constructor with one parameter or default value for others (even undefined ones). Does not report copy-constructor. Additional message for constructor case: This constructor should be flagged as "explicit".

N.	JSF++ Definition	Polyspace Specification
178	Down casting (casting from base to derived class) **shall** only be allowed through one of the following mechanism: • Virtual functions that act like dynamic casts (most likely useful in relatively simple cases). • Use of the visitor (or similar) pattern (most likely useful in complicated cases).	Reports explicit down casting, dynamic_cast included. (Visitor patter does not have a special case.)
179	A pointer to a virtual base class **shall not** be converted to a pointer to a derived class.	Reports this specific down cast. Allows dynamic_cast.

180	Implicit conversions that may result in a loss of information **shall not** be used.	Reports the following implicit casts :
		`integer => smaller integer`
		`unsigned => smaller or eq signed`
		`signed => smaller or eq un-signed`
		`integer => float`
		`float => integer`
		Does not report for cast to `bool` reports for implicit cast on constant done with the options `-scalar-overflows-checks signed-and-unsigned` or `-ignore-constant-overflows`
		Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

181	Redundant explicit casts **will not** be used.	Reports useless cast: `cast T to T`. Casts to equivalent `typedefs` are also reported.
182	Type casting from any type to or from pointers **shall not** be used.	Does not report when Rule 181 applies.

N.	JSF++ Definition	Polyspace Specification
184	Floating point numbers **shall not** be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	Reports `float->int` conversions. Does not report implicit ones.
185	C++ style casts (`const_cast`, `reinterpret_cast`, and `static_cast`) **shall** be used instead of the traditional C-style casts.	

Flow Control Standards

N.	JSF++ Definition	Polyspace Specification
186	There **shall** be no unreachable code.	Done with gray checks in the software. Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.
187	All non-null statements **shall** potentially have a side-effect.	
188	Labels **will not** be used, except in switch statements.	
189	The `goto` statement **shall** not be used.	
190	The `continue` statement **shall not** be used.	
191	The `break` statement **shall not** be used (except to terminate the cases of a switch statement).	
192	All `if`, `else if` constructs will contain either a final `else` clause or a comment indicating why a final `else` clause is not necessary.	`else if` should contain an `else` clause.
193	Every non-empty `case` clause in a switch statement **shall** be terminated with a `break` statement.	

N.	JSF++ Definition	Polyspace Specification
194	All `switch` statements that do not intend to test for every enumeration value **shall** contain a final `default` clause.	Reports only for missing `default`.
195	A `switch` expression **will** not represent a Boolean value.	
196	Every `switch` statement **will** have at least two cases and a potential `default`.	
197	Floating point variables **shall not** be used as loop counters.	Assumes 1 loop parameter.
198	The initialization expression in a `for` loop **will** perform no actions other than to initialize the value of a single `for` loop parameter.	Reports if `loop` parameter cannot be determined. Assumes Rule 200 is not violated. The `loop variable` parameter is assumed to be a variable.
199	The increment expression in a `for` loop **will** perform no action other than to change a single loop parameter to the next value for the loop.	Assumes 1 loop parameter (Rule 198), with non class type. Rule 200 must not be violated for this rule to be reported.
200	Null initialize or increment expressions in `for` loops **will not** be used; a `while` loop will be used instead.	
201	Numeric variables being used within a *for* loop for iteration counting shall not be modified in the body of the loop.	Assumes 1 loop parameter (AV rule 198), and no alias writes.

Expressions

N.	JSF++ Definition	Polyspace Specification
202	Floating point variables **shall not** be tested for exact equality or inequality.	Reports only direct equality/inequality. Check done for all expressions.
203	Evaluation of expressions **shall not** lead to overflow/underflow.	Done with overflow checks in the software.
204	A single operation with side-effects shall only be used in the following contexts: • by itself	Reports when: • A side effect is found in a return statement

N.	JSF++ Definition	Polyspace Specification
	• the right-hand side of an assignment • a condition • the only argument expression with a side-effect in a function call • condition of a loop • switch condition • single part of a chained operation	• A side effect exists on a single value, and only one operand of the function call has a side effect.
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	Reports when: • Variable is written more than once in an expression • Variable is read and write in sub-expressions • Volatile variable is accessed more than once **Note:** Read-write operations such as ++, are only considered as a write.
205	The volatile keyword **shall not** be used unless directly interfacing with hardware.	Reports if volatile keyword is used.

Memory Allocation

N.	JSF++ Definition	Polyspace Specification
206	Allocation/deallocation from/to the free store (heap) **shall not** occur after initialization.	Reports calls to C library functions: `malloc` / `calloc` / `realloc` / `free` and all `new`/ `delete` operators in functions or methods.

Fault Handling

N.	JSF++ Definition	Polyspace Specification
208	C++ exceptions **shall not** be used.	Reports `try`, `catch`, `throw spec`, and `throw`.

Portable Code

N.	JSF++ Definition	Polyspace Specification
209	The basic types of `int`, `short`, `long`, `float` and `double` **shall not** be used, but specific-length equivalents should be `typedef`'d accordingly for each compiler, and these type names used in the code.	Only allows use of basic types through direct `typedefs`.
213	No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	Reports when a binary operation has one operand that is not parenthesized and is an operation with inferior precedence level. Reports bitwise and shifts operators that are used without parenthesis and binary operation arguments.

215	Pointer arithmetic **will not** be used.	Reports:
		`p + I`
		`p - I`
		`p++`
		`p--`
		`p+=`
		`p-=`
		
		Allows `p[i]`.

Unsupported JSF++ Rules

Code Size and Complexity

N.	JSF++ Definition
2	There shall not be any self-modifying code.

Rules

N.	JSF++ Definition
4	To break a “should” rule, the following approval must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)
5	To break a “will” or a “shall” rule, the following approvals must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) • approval from the software product manager (obtained by the unit approval in the developmental CM tool)
6	Each deviation from a “shall” rule shall be documented in the file that contains the deviation. Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.
7	Approval will not be required for a deviation from a “shall” or “will” rule that complies with an exception specified by that rule.

Environment

N.	JSF++ Definition
10	Values of character types will be restricted to a defined and documented subset of ISO 10646 1.

Libraries

N.	JSF++ Definition
16	Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.

Header Files

N.	JSF++ Definition
34	Header files should contain logically related declarations only.
36	Compilation dependencies should be minimized when possible.
37	Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file — not the .h file.
38	Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.

Style

N.	JSF++ Definition
45	All words in an identifier will be separated by the '_' character.
49	All acronyms in an identifier will be composed of uppercase letters.
55	The name of a header file should reflect the logical entity for which it provides declarations.
56	The name of an implementation file should reflect the logical entity for which it provides definitions and have a ".cpp" extension (this name will normally be identical to the header file that provides the corresponding declarations.) At times, more than one .cpp file for a given logical entity will be required. In these cases, a suffix should be appended to reflect a logical differentiation.

Classes

N.	JSF++ Definition
64	A class interface should be complete and minimal.
65	A structure should be used to model an entity that does not require an invariant.
66	A class should be used to model an entity that maintains an invariant.
69	A member function that does not affect the state of an object (its instance variables) will be declared const. Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.
70	A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.
70.1	An object shall not be improperly used before its lifetime begins or after its lifetime ends.
71	Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.
72	The invariant for a class should be: • A part of the postcondition of every class constructor, • A part of the precondition of the class destructor (if any), • A part of the precondition and postcondition of every other publicly accessible operation.
73	Unnecessary default constructors shall not be defined.
77	A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).
80	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.
84	Operator overloading will be used sparingly and in a conventional manner.
85	When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.
86	Concrete types should be used to represent simple independent concepts.
87	Hierarchies should be based on abstract classes.
90	Heavily used interfaces should be minimal, general and abstract.

N.	JSF++ Definition
91	Public inheritance will be used to implement "is-a" relationships.
92	A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system: • Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. • Postconditions of derived methods must be at least as strong as the postconditions of the methods they override. In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.
93	"has-a" or "is-implemented-in-terms-of" relationships will be modeled through membership or non-public inheritance.

Namespaces

N.	JSF++ Definition
100	Elements from a namespace should be selected as follows: • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.

Templates

N.	JSF++ Definition
101	Templates shall be reviewed as follows: **1** with respect to the template in isolation considering assumptions or requirements placed on its arguments. **2** with respect to all functions instantiated by actual arguments.
102	Template tests shall be created to cover all actual template instantiations.
103	Constraint checks should be applied to template arguments.
105	A template definition's dependence on its instantiation contexts should be minimized.

N.	JSF++ Definition
106	Specializations for pointer types should be made where appropriate.

Functions

N.	JSF++ Definition
112	Function return values should not obscure resource ownership.
115	If a function returns error information, then that error information will be tested.
117	Arguments should be passed by reference if NULL values are not possible: • **117.1** – An object should be passed as `const T&` if the function should not change the value of the object. • **117.2** – An object should be passed as `T&` if the function may change the value of the object.
118	Arguments should be passed via pointers if NULL values are possible: • **118.1** – An object should be passed as `const T*` if its value should not be modified. • **118.2** – An object should be passed as `T*` if its value may be modified.
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.
122	Trivial accessor and mutator functions should be inlined.
123	The number of accessor and mutator functions should be minimized.
124	Trivial forwarding functions should be inlined.
125	Unnecessary temporary objects should be avoided.

Comments

N.	JSF++ Definition
127	Code that is not used (commented out) shall be deleted. **Note**: This rule cannot be annotated in the source code.
128	Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.

N.	JSF++ Definition
129	Comments in header files should describe the externally visible behavior of the functions or classes being documented.
130	The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.
134	Assumptions (limitations) made by functions should be documented in the function's preamble.

Initialization

N.	JSF++ Definition
143	Variables will not be introduced until they can be initialized with meaningful values. (See also AV Rule 136, AV Rule 142, and AV Rule 73 concerning declaration scope, initialization before use, and default constructors respectively.)

Types

N.	JSF++ Definition
146	Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE® Std 754 [1].

Unions and Bit Fields

N.	JSF++ Definition
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Operators

N.	JSF++ Definition
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.

Type Conversions

N.	JSF++ Definition
183	Every possible measure should be taken to avoid type casting.

Expressions

N.	JSF++ Definition
204	A single operation with side-effects shall only be used in the following contexts: **1** by itself **2** the right-hand side of an assignment **3** a condition **4** the only argument expression with a side-effect in a function call **5** condition of a loop **6** switch condition **7** single part of a chained operation

Memory Allocation

N.	JSF++ Definition
207	Unencapsulated global data will be avoided.

Portable Code

N.	JSF++ Definition
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.).
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
211	Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
212	Underflow or overflow functioning shall not be depended on in any special way.

N.	JSF++ Definition
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.

Efficiency Considerations

N.	JSF++ Definition
216	Programmers should not attempt to prematurely optimize code.

Miscellaneous

N.	JSF++ Definition
217	Compile-time and link-time errors should be preferred over run-time errors.
218	Compiler warning levels will be set in compliance with project policies.

Testing

N.	JSF++ Definition
219	All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.
220	Structural coverage algorithms shall be applied against flattened classes.
221	Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.

Check Coding Rules from the Polyspace Environment

Activate Coding Rules Checker

This example shows how to activate the coding rules checker before you start an analysis. This activation enables Polyspace Bug Finder to search for coding rule violations. You can view the coding rule violations in your analysis results.

1 Open project configuration.
2 On the **Configuration** pane, select **Coding Rules**.
3 Select the check box for the type of coding rules that you want to check.

   For C code, you can check compliance with:

   - MISRA C:2004
   - MISRA AC AGC
   - MISRA C:2012

     If you have generated code, use the **Use generated code requirements** option to use the MISRA C:2012 categories for generated code.
   - Custom coding rules

   For C++ code, you can check compliance with:

   - MISRA C++: 2008
   - JSF C++
   - Custom coding rules
4 For each rule type that you select, from the drop-down list, select the subset of rules to check.

   MISRA C:2004

Option	Description
`required-rules`	All required MISRA C:2004 coding rules.
`all-rules`	AllMISRA C:2004 coding rules (required and advisory).
`SQO-subset1`	A small subset of MISRA C:2004 rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.

Option	Description
`SQO-subset2`	A second subset of rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA C:2004 coding rules that you specify.

MISRA AC AGC

Option	Description
`OBL-rules`	All required MISRA AC AGC coding rules.
`OBL-REC-rules`	All required and recommended MISRA AC AGC coding rules.
`all-rules`	All required, recommended, and readability coding rules.
`SQO-subset1`	A small subset of MISRA AC AGC rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of MISRA AC AGC rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA AC AGC coding rules that you specify.

MISRA C:2012

Option	Description
`mandatory`	All mandatory MISRA C:2012 coding rules. If you have generated code, also use the **Use generated code requirements** option categorization for generated code.
`mandatory-required`	All mandatory and required MISRA C:2012 coding rules. If you have generated code, also use the **Use generated code requirements** option categorization for generated code.
`all`	All MISRA C:2012 coding rules (mandatory, required, and advisory).

Option	Description
`SQO-subset1`	A small subset of MISRA C rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA C:2012 coding rules that you specify.

MISRA C++

Option	Description
`required-rules`	All required MISRA C++ coding rules.
`all-rules`	All required and advisory MISRA C++ coding rules.
`SQO-subset1`	A small subset of MISRA C++ rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules with indirect impact on the selectivity in addition to `SQO-subset1`. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A specified set of MISRA C++ coding rules.

JSF C++

Option	Description
`shall-rules`	**Shall** rules are mandatory requirements. These rules require verification.
`shall-will-rules`	All **Shall** and **Will** rules. **Will** rules are intended to be mandatory requirements. However, these rules do not require verification.
`all-rules`	All **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.
`custom`	A set of JSF C++ coding rules that you specify.

5 If you select **Check custom rules**, specify the path to your custom rules file or click **Edit** to create one.

When rules checking is complete, the software displays the coding rule violations in purple on the **Results Summary** pane.

Related Examples

- “Select Specific MISRA or JSF Coding Rules”
- “Create Custom Coding Rules”
- “Exclude Files From Analysis”

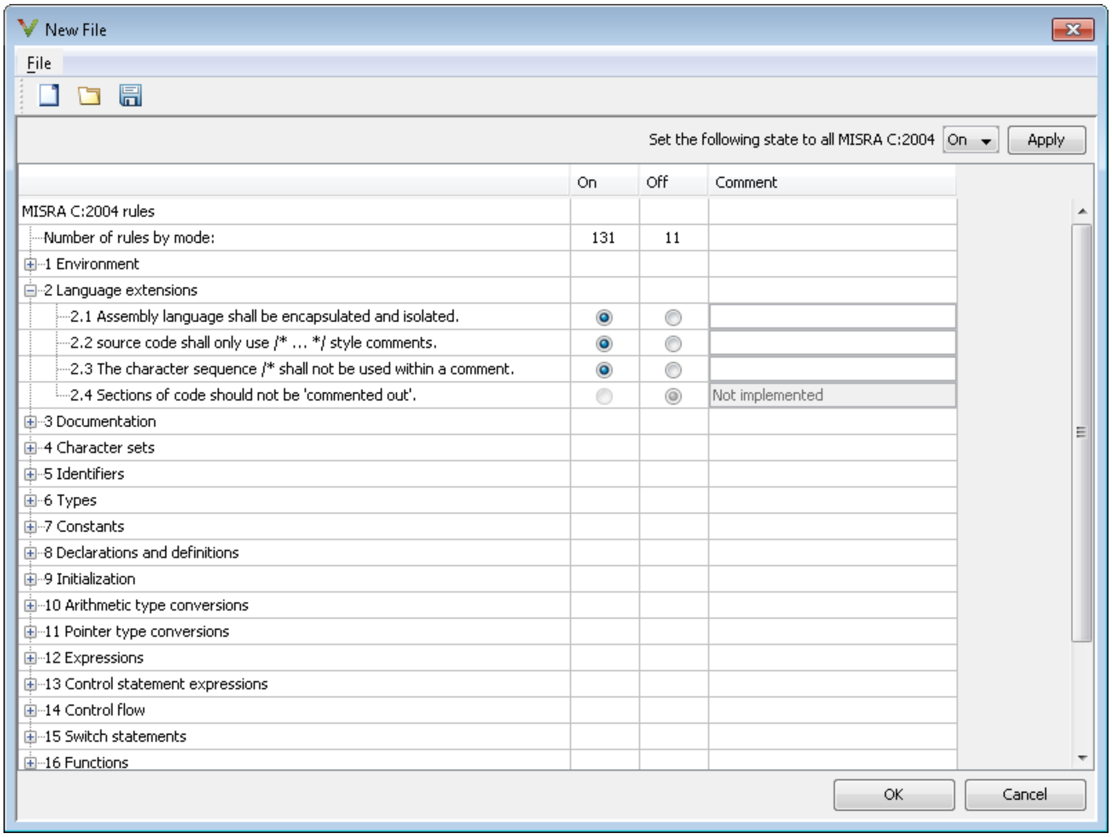
More About

- “Rule Checking”

Select Specific MISRA or JSF Coding Rules

This example shows how to specify a subset of MISRA or JSF rules for the coding rules checker. If you select `custom` from the MISRA or JSF drop-down list, you must provide a file that specifies the rules to check.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**.
3. Select the check box for the type of coding rules you wish to check
4. From the corresponding drop-down list, select `custom`. The software displays a new field for your custom file.
5. To the right of this field, click **Edit**. A New File window opens, displaying a table of rules.



	On	Off	Comment
MISRA C:2004 rules			
Number of rules by mode:	131	11	
1 Environment			
2 Language extensions			
2.1 Assembly language shall be encapsulated and isolated.	●	○	
2.2 source code shall only use /* … */ style comments.	●	○	
2.3 The character sequence /* shall not be used within a comment.	●	○	
2.4 Sections of code should not be 'commented out'.	○	●	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			

Select **On** for the rules you want to check.

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

The **Save as** dialog box opens.

7 In the **File** field, enter a name for the rules file.

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

The full path to the rules file appears. To reuse this rules file for other projects, type this path name or use the 🗀 icon in the New File window.

Related Examples

- "Activate Coding Rules Checker"
- "Create Custom Coding Rules"

More About

- "Rule Checking"

Create Custom Coding Rules

This example shows how to create a custom coding rules file. You can use this file to check names or text patterns in your source code against custom rules that you specify. For each rule, you specify a pattern in the form of a regular expression. The software compares the pattern against identifiers in the source code and determines whether the custom rule is violated.

1 Create Coding Rules File

1 Create a Polyspace project. Add `printInitialValue.c` to the project.

2 On the **Configuration** pane, select **Coding Rules**. Select the **Check custom rules** box.

3 Click Edit.

The New File window opens, displaying a table of rule groups.

4 From the drop-down list **Set the following state to all Custom C**, select `Off`. Click **Apply**.

5 Expand the **Structs** node. For the option **4.3 All struct fields must follow the specified pattern**:

Column Title	Action
On	Select ●.
Convention	Enter `All struct fields must begin with s_ and have capital letters`
Pattern	Enter `s_[A-Z0-9_]`
Comment	Leave blank. This column is for comments that appear in the coding rules file alone.

2 Review Coding Rule Violations

1 Save the file and run the verification. On the **Results Summary** pane, you see two violations of rule 4.3. Select the first violation.

a On the **Source** pane, the line `int a;` is marked.

  b. On the **Check Details** pane, you see the error message you had entered, `All struct fields must begin with s_ and have capital letters`

2. Right-click on the **Source** pane and select **Open Editor**. The file `printInitialValue.c` opens in the **Code Editor** pane or an external text editor depending on your **Preferences**.
3. In the file, replace all instances of `a` with `s_A` and `b` with `s_B`. Rerun the verification.

   The custom rule violations no longer appear on the **Results Summary** pane.

Related Examples

- "Activate Coding Rules Checker"
- "Select Specific MISRA or JSF Coding Rules"
- "Exclude Files From Analysis"

More About

- "Rule Checking"
- "Format of Custom Coding Rules File"

Format of Custom Coding Rules File

In a custom coding rules file, each rule appears in the following format:

```
N.n off|on
convention=violation_message
pattern=regular_expression
```

- `N.n` — Custom rule number, for example, 1.2.
- `off` — Rule is not considered.
- `on` — The software checks for violation of the rule. After verification, it displays the coding rule violation on the **Results Summary** pane.
- *`violation_message`* — Software displays this text in an XML file within the *`Results`*`/Polyspace-Doc` folder.
- *`regular_expression`* — Software compares this text pattern against a source code identifier that is specific to the rule. See “Custom Coding Rules”.

  The keywords `convention=` and `pattern=` are optional. If present, they apply to the rule whose number immediately precedes these keywords. If `convention=` is not given for a rule, then a standard message is used. If `pattern=` is not given for a rule, then the default regular expression is used, that is, `.*`.

  Use the symbol # to start a comment. Comments are not allowed on lines with the keywords `convention=` and `pattern=`.

  The following example contains three custom rules: 1.1, 8.1, and 9.1.

```
# Custom rules configuration file
1.1  off           # Disable custom rule number 1.1
8.1  on       # Violation of custom rule 8.1 produces a warning
convention=Global constants must begin by G_ and must be in capital letters.
pattern=G_[A-Z0-9_]*
9.1  on    # Non-adherence to custom rule 9.1 produces a warning
convention=Global variables should begin by g_.
pattern=g_.*
```

Related Examples

- “Create Custom Coding Rules”

Exclude Files From Analysis

This example shows how to exclude certain files from defect and coding rules checking.

1. Open the project configuration.
2. In the **Configuration** tree view, select **Inputs & Stubbing**.
3. Select the **Files and folders to ignore** check box.
4. From the corresponding drop-down list, select one of the following:

   - `all-headers` (default) — Excludes header files in the Include folders of your project. For example `.h` or `.hpp` files.
   - `all` — Excludes all include files in the Include folders of your project. For example, if you are checking a large code base with standard or Visual headers, excluding include folders can significantly improve the speed of code analysis.
   - `custom` — Excludes files or folders specified in the **File/Folder** view. To add files to the custom **File/Folder** list, select to choose the files and folders to exclude. To remove a file or folder from the list of excluded files and folders, select the row. Then click .

Related Examples

- "Activate Coding Rules Checker"

More About

- "Rule Checking"

Allow Custom Pragma Directives

This example shows how to exclude custom pragma directives from coding rules checking. MISRA C rule 3.4 requires checking that pragma directives are documented within the documentation of the compiler. However, you can allow undocumented pragma directives to be present in your code.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**.
3. To the right of **Allowed pragmas**, click ➕.

   In the **Pragma** view, the software displays an active text field.
4. In the text field, enter a pragma directive.
5. To remove a directive from the **Pragma** list, select the directive. Then click ✖.

Related Examples

- "Activate Coding Rules Checker"

More About

- "Rule Checking"

Specify Boolean Types

This example shows how to specify data types you want Polyspace to consider as Boolean during MISRA C rules checking. The software applies this redefinition only to data types defined by `typedef` statements. The use of this option may affect the checking of MISRA C:2004 rules 12.6, 13.2, 15.4, and MISRA C:2012 rules 14.4, 16.7.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**.
3. To the right of **Effective boolean types**, click ➕.

   In the **Type** view, the software displays an active text field.
4. In the text field, specify the data type that you want Polyspace to treat as Boolean.
5. To remove a data type from the **Type** list, select the data type. Then click ✖.

Related Examples

- "Activate Coding Rules Checker"

More About

- "Rule Checking"

Find Coding Rule Violations

This example shows how to check for coding rule violations alone.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**. Activate the desired coding rule checker.
3. In the **Configuration** tree view, select **Bug Finder Analysis**.
4. Clear the **Find defects** check box.
5. Click ▷ Run to run the coding rules checker without checking defects.

   You can view the results by selecting the `*RuleSet*-report.xml` file from the results folder.

Related Examples

- “Activate Coding Rules Checker”
- “Select Specific MISRA or JSF Coding Rules”
- “Review Coding Rule Violations”

More About

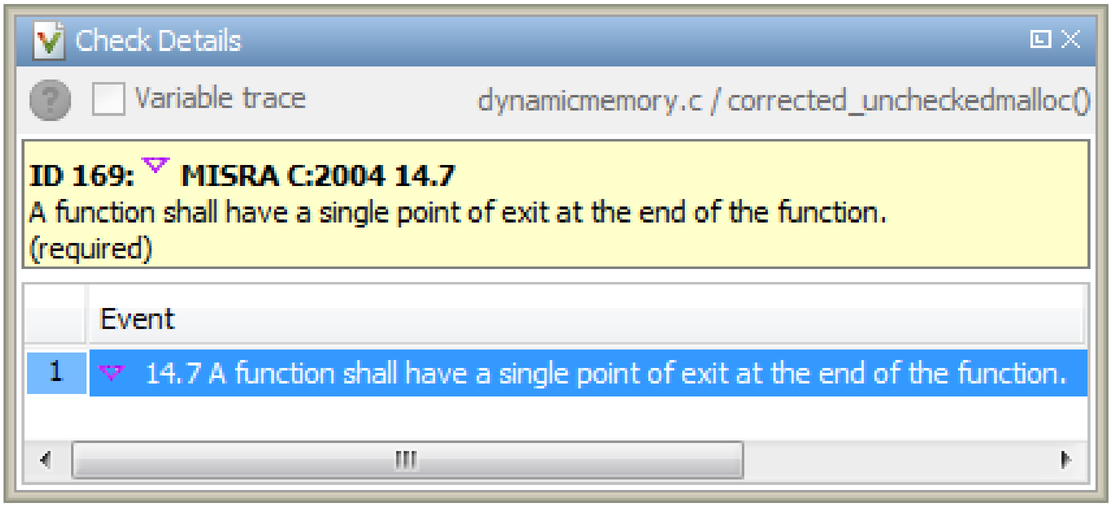
- “Rule Checking”

Review Coding Rule Violations

This example shows how to review coding rule violations once code analysis is complete. After analysis, the **Results Summary** pane displays the rule violations with a

- ▽ symbol for predefined coding rules, MISRA or JSF.
- ▼ symbol for custom coding rules.

1. Select a coding-rule violation on the **Results Summary** pane.
   - The predefined rules such as MISRA or JSF are indicated by ▽ .
   - The custom rules are indicated by ▼ .
2. On the **Check Details** pane, view the location and description of the violated rule. In the source code, the line containing the violation appears highlighted.



dynamicmemory.c / corrected_uncheckedmalloc()

ID 169: ▽ MISRA C:2004 14.7
A function shall have a single point of exit at the end of the function. (required)

1 ▼ 14.7 A function shall have a single point of exit at the end of the function.

3. Review the violation. On the **Results Summary** pane, select a **Classification** to describe the severity of the issue:
   - `High`
   - `Medium`
   - `Low`
   - `Not a defect`

4. Select a **Status** to describe how you intend to address the issue:

   - `Fix`
   - `Improve`
   - `Investigate`
   - `Justified`

     (This status also marks the result as justified.)
   - `No action planned`

     (This status also marks the result as justified.)
   - `Other`

   You can also define your own statuses.
5. In the comment box, enter additional information about the violation.
6. To open the source file that contains the coding rule violation, on the **Source** pane, right-click the code with the purple check. From the context menu, select **Open Editor**. The file opens in the **Code Editor** pane or an external text editor depending on your **Preferences**.
7. Fix the coding rule violation.
8. When you have corrected the coding rule violations, run the analysis again.

Related Examples

- "Activate Coding Rules Checker"
- "Find Coding Rule Violations"
- "Filter and Group Coding Rule Violations"

Filter and Group Coding Rule Violations

This example shows how to use filters in the **Results Summary** pane to focus on specific kinds of coding rule violations. By default, the software displays both coding rule violations and defects.

Group Violations

1. On the **Results Summary** pane, select **Group by** > **Family**.

   The rules are grouped by numbers. Each group corresponds to a certain code construct.
2. Expand the group nodes to select an individual coding rule violation.

Filter Violations

1. On the **Results Summary** pane, place your cursor on the **Check** column header. Click the filter icon that appears.
2. From the context menu, clear the **All** check box.
3. Select the violated rule numbers that you want to focus on.
4. Click **OK**.

Related Examples

- "Activate Coding Rules Checker"
- "Review Coding Rule Violations"

Find Bugs From the Polyspace Environment

Choose Specific Defects

There are two preset configurations for Bug Finder defects, but you can also customize which defects to check for during the analysis.

1 On the **Configuration** pane, select **Bug Finder Analysis**.
2 From the **Find defects** menu, select a set of defects. The options are:

   - `default` for the default list of defects. This list contains defects that are applicable to most coding projects. To see the defects in the default list, expand the nodes **Numerical**, **Static memory**, etc.
   - `all` for all defects.
   - `custom` to add defects to the default list or remove defects from it.

Run Local Analysis

Before running an analysis from the Polyspace interface, you must set up your project's source files and analysis options. For more information, see "Create New Project".

1. Select a project to analyze.
2. Select the ▷ Run button.
3. Monitor the analysis on the **Output Summary** tab. If the analysis fails, this tab also lists errors or warnings.

   Once the analysis is complete, on the **Project Browser**, you can see the word **Completed** next to your project result. The **Results Summary** tab opens automatically with your completed results.

Run Remote Batch Analysis

Before running a batch analysis, you must set up your project's source files, analysis options, and remote analysis settings. If you have not done so, see "Create New Project" and "Set Up Polyspace Metrics".

1 Select a project to analyze.
2 On the **Configuration** pane, select **Distributed Computing**.
3 Select **Batch**.
4 If you want to store your results in the Polyspace Metrics repository, select **Add to results repository**.

   Otherwise, clear this check box.
5 Select the ▷ Run button.
6 To monitor the analysis, select **Tools** > **Open Job Monitor**.

   Once the analysis is complete, you can open your results from the Results folder, or download them from Polyspace Metrics.

Related Examples

- "Open Results"
- "Download Results From Polyspace Metrics"

Monitor Analysis

To monitor the progress of a local analysis, use the following panes in the Polyspace Bug Finder interface. To open or close one of the tabs, select **Window** > **Show/Hide View**.

- **Output Summary** — Displays progress of verification, compile phase messages and errors. To search for a term, in the **Search** field, enter the required term. Click the up or down arrow to move sequentially through occurrences of the term.
- **Full Log** — This tab displays messages, errors, and statistics for the phases of the analysis. To search for a term, in the **Search** field, enter the required term. Click the up arrow or down arrow to move sequentially through occurrences of this term.

At the end of a local analysis, the **Dashboard** tab displays statistics, for example, code coverage and check distribution.

To monitor the progress of a remote analysis:

1. From the Polyspace interface, select **Tools** > **Open Job Monitor**.
2. In the Polyspace Job Monitor, follow your queued job to monitor progress.

Specify Results Folder

By default, Polyspace Bug Finder saves your results in the same directory as your project in a folder called `Results`. Each subsequent analysis overwrites the old results.

However, to specify a different location for results:

1. On the **Project Browser**, right-click the Results folder.
2. Select **Choose a Result Folder**.
3. In the Choose a Result Folder window, navigate to the new results folder and click **Select**.

   On the **Project Browser**, the new results folder appears.

   The previous results folder disappears from the **Project Browser**. However, the results have not been deleted, just removed from the Polyspace interface. To view the previous results, use **File** > **Open Result**.

View Results in the Polyspace Environment

Open Results

This example shows how to open Polyspace Bug Finder results. Before you open the results, you must run Polyspace Bug Finder analysis on your source files, which produces a results file with the extension `.psbf`.

Open Results from Active Project

Suppose you have a project called `Bug_Finder_Example` open in the **Project Browser**. After analysis, the results appear under the project as `Result_Bug_Finder_Example`. Results open automatically. To manually open results, double-click `Result_Bug_Finder_Example`.

Open Results File Using File Browser

If the results file `Bug_Finder_Example.psbf` is located on the path `'C:\Bug_Finder_Example\Results'`

1. Select **File** > **Open Result**. The Open Results browser opens.
2. Navigate to the result folder containing the file with extension `.psbf`. In this example, navigate to `'C:\Bug_Finder_Example\Results'`.
3. Select the file. Click **Open**.

More About

- “Results Folder Contents” on page 5-32
- “Windows Used to Review Results”

View Results Summary in Polyspace Metrics

This example shows how to view results summary in Polyspace Metrics. On the **Configuration** pane, under **Distributed Computing**, if you select **Add to results repository**, after remote analysis, you can view a summary of the results in Polyspace Metrics.

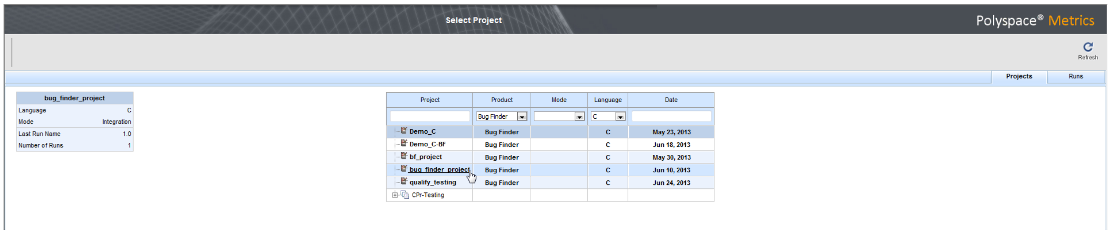
Open Polyspace Metrics

In the address bar of your Web browser, enter the following URL:

`protocol``://` *`ServerName`*`:` *`PortNumber`*

- *`protocol`* is either `http` (default) or `https`. To use HTTPS, you must set up the configuration file and the **Metrics and Remote Server Settings**.
- *`ServerName`* is the name or IP address of your Polyspace Metrics server.
- *`PortNumber`* is the Web server port number (default 8080)

On the webpage, you can view the projects saved to your Polyspace Metrics repository.



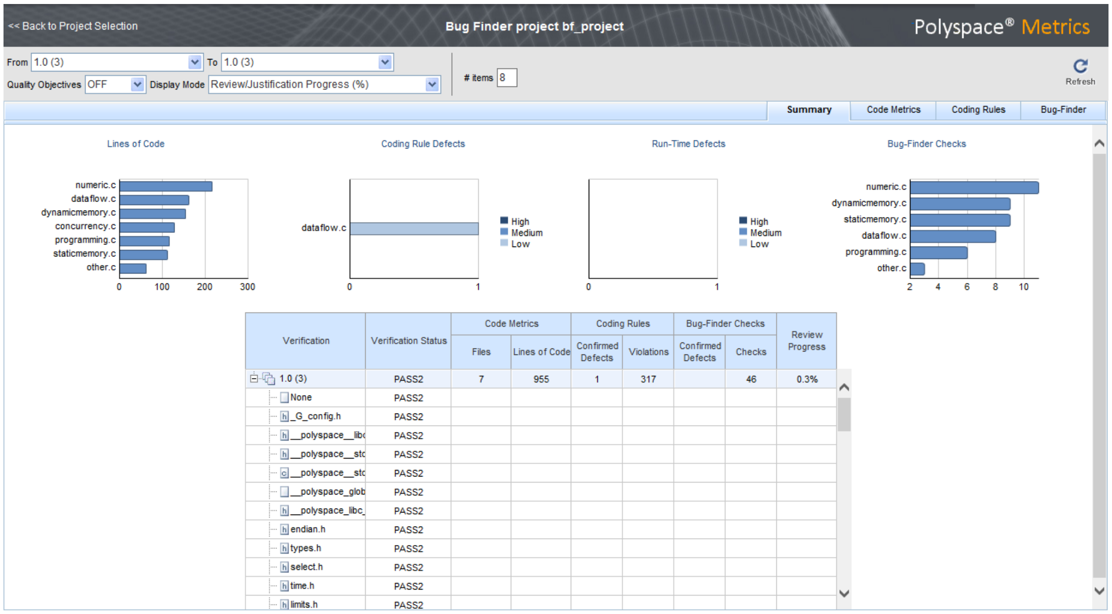
bug_finder_project	
Language	C
Mode	Integration
Last Run Name	1.0
Number of Runs	1

Project	Product	Mode	Language	Date
	Bug Finder		C	
Demo_C	Bug Finder		C	May 23, 2013
Demo_C-BF	Bug Finder		C	Jun 18, 2013
bf_project	Bug Finder		C	May 30, 2013
bug_finder_project	Bug Finder		C	Jun 10, 2013
qualify_testing	Bug Finder		C	Jun 24, 2013
CPr-Testing				

View Results Summary

1. Select the **Projects** tab.
2. To view the results summary for your project, on the **Projects** column, select the project name.

   The results summary for the project appears on the webpage under the **Summary** tab. The **Confirmed Defects** column lists the number of coding rule violations or checks that you have reviewed.



From 1.0 (3) To 1.0 (3)

Quality Objectives OFF Display Mode Review/Justification Progress (%)

items 8

Verification	Verification Status	Code Metrics		Coding Rules		Bug-Finder Checks		Review Progress
		Files	Lines of Code	Confirmed Defects	Violations	Confirmed Defects	Checks	
1.0 (3)	PASS2	7	955	1	317		46	0.3%
None	PASS2							
_G_config.h	PASS2							
__polyspace__libc	PASS2							
__polyspace__stc	PASS2							
__polyspace__stc	PASS2							
__polyspace_glob	PASS2							
__polyspace_libc_	PASS2							
endian.h	PASS2							
types.h	PASS2							
select.h	PASS2							
time.h	PASS2							
limits.h	PASS2							

3 To view the results in more detail, select the tabs:

- **Code Metrics**: Metrics such as number of lines, header files and function calls.
- **Coding Rules**: Description of coding rule violations
- **Bug-Finder**: Description of defects detected by Polyspace Bug Finder

Related Examples

- “Set Up Polyspace Metrics”
- “Download Results From Polyspace Metrics”
- “Review and Comment Results”

Download Results From Polyspace Metrics

This example shows how to download results from Polyspace Metrics. On the **Configuration** pane, under **Distributed Computing**, if you select **Add to results repository**, after remote analysis, you can view a summary of the results in Polyspace Metrics.

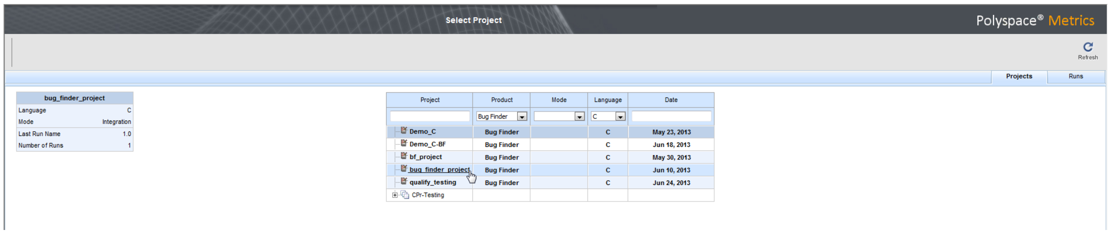
Open Polyspace Metrics

In the address bar of your Web browser, enter the following URL:

`protocol:// ServerName: PortNumber`

- *`protocol`* is either `http` (default) or `https`. To use HTTPS, you must set up the configuration file and the **Metrics and Remote Server Settings**.
- *`ServerName`* is the name or IP address of your Polyspace Metrics server.
- *`PortNumber`* is the Web server port number (default 8080)

On the webpage, you can view the projects saved to your Polyspace Metrics repository.



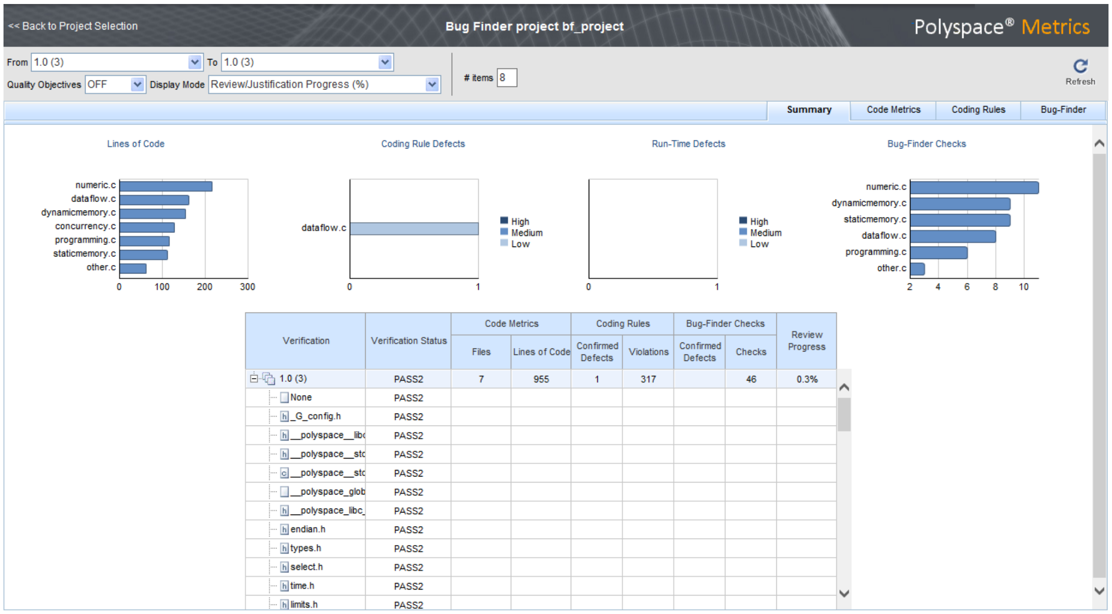
bug_finder_project	
Language	C
Mode	Integration
Last Run Name	1.0
Number of Runs	1

Project	Product	Mode	Language	Date
	Bug Finder		C	
Demo_C	Bug Finder		C	May 23, 2013
Demo_C-BF	Bug Finder		C	Jun 18, 2013
bf_project	Bug Finder		C	May 30, 2013
bug_finder_project	Bug Finder		C	Jun 10, 2013
qualify_testing	Bug Finder		C	Jun 24, 2013
CPr-Testing				

Download Results

1. Select the **Projects** tab.
2. To view the results summary for your project, on the **Projects** column, select the project name.

   The results summary for the project appears on the webpage under the **Summary** tab.



Bug Finder project bf_project

From 1.0 (3) To 1.0 (3)

Quality Objectives OFF Display Mode Review/Justification Progress (%)

items 8

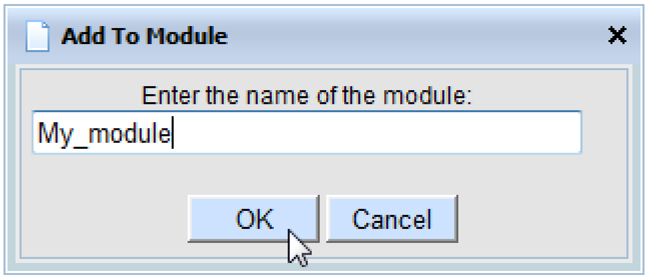
Verification	Verification Status	Code Metrics		Coding Rules		Bug-Finder Checks		Review Progress
		Files	Lines of Code	Confirmed Defects	Violations	Confirmed Defects	Checks	
1.0 (3)	PASS2	7	955	1	317		46	0.3%
None	PASS2							
_G_config.h	PASS2							
__polyspace__libc	PASS2							
__polyspace__stc	PASS2							
__polyspace__stc	PASS2							
__polyspace_glob	PASS2							
__polyspace_libc_	PASS2							
endian.h	PASS2							
types.h	PASS2							
select.h	PASS2							
time.h	PASS2							
limits.h	PASS2							

3 To download results:

- For an individual file, on the **Verification** column, select the name of the file.
- For a group of files:

  a Right-click on the row containing a file in the group. From the context menu, select **Add To Module**.

  b Enter the name of your module in the dialog box. Click **OK**.



Enter the name of the module:

My_module

The name of the module appears on the **Verification** column.

   c. Drag and drop the other files in the group to the module.
   d. Select the name of the module.
- For all files in the project, on the **Verification** column, select the version number of the project.

The results open in Polyspace Bug Finder.

Related Examples

- “Set Up Polyspace Metrics”
- “View Results Summary in Polyspace Metrics”
- “Review and Comment Results”

Filter and Group Results

This example shows how to filter and group defects on the **Results Summary** pane. To organize your review of results, use filters and groups when you want to:

- Review certain categories of defects in preference to others. For instance, you first want to address the defects resulting from **Missing or invalid return statement**.
- Review only new results found since the last analysis.
- Not address the full set of coding rule violations detected by the coding rules checker.
- Review only those defects that you have already assigned a certain status. For instance, you want to review only those defects to which you have assigned the status, `Investigate`.
- Review defects from a particular file or function. Because of continuity of code, reviewing these defects together can help you organize your review process.

  If you have written the code for a particular source file, you can review the defects only in that file.

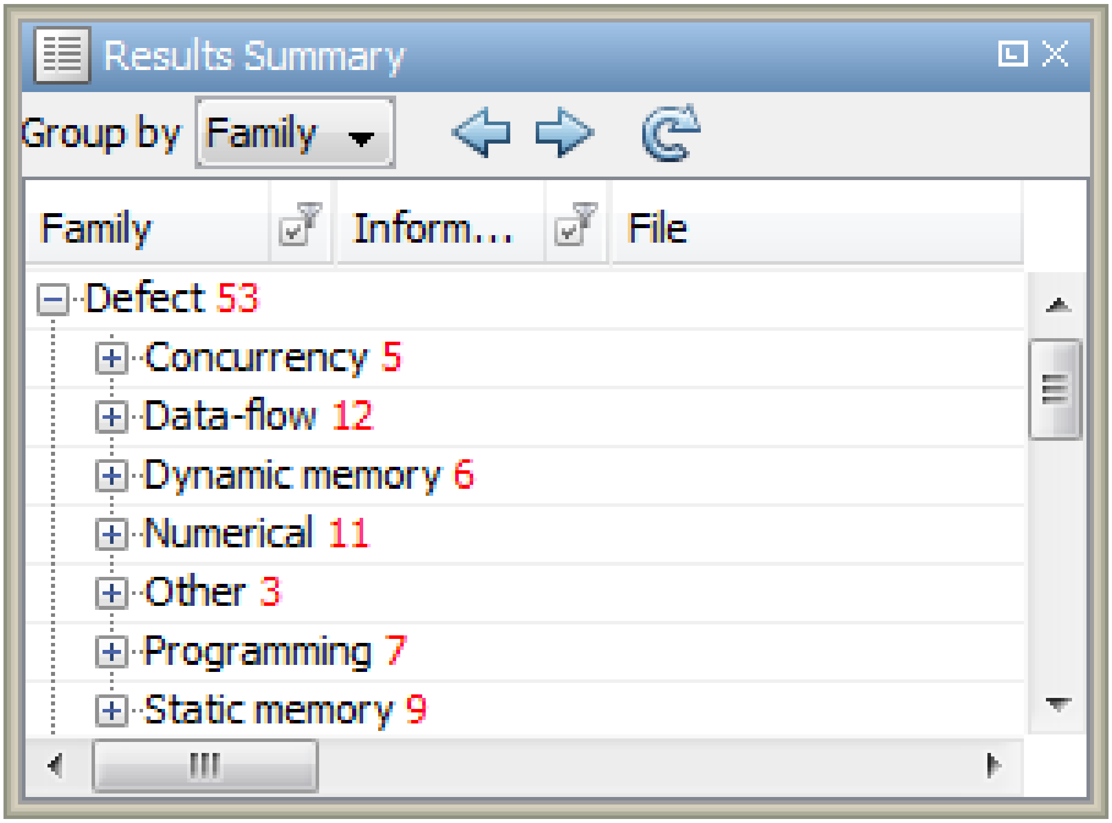
Review Defects in a Given Category

1 To review defects resulting from **Array access out of bounds**:

   a Open the results file, with extension, `.psbf`.

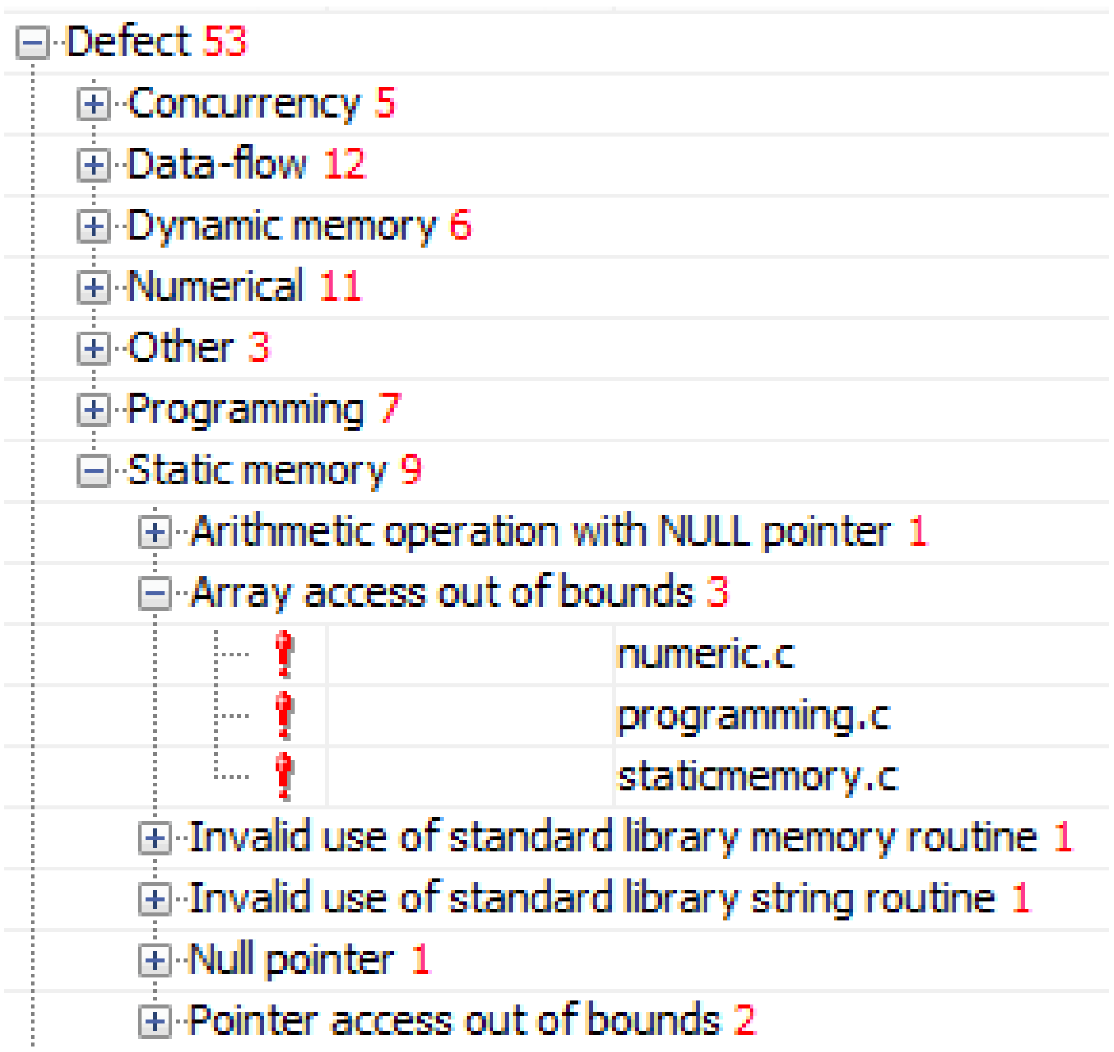
   b On the **Results Summary** pane, select **Group by** > **Family**.

      The defects are grouped by type.



Defect 53
- Concurrency 5
- Data-flow 12
- Dynamic memory 6
- Numerical 11
- Other 3
- Programming 7
- Static memory 9

c Under the category **Static memory**, expand the subcategory **Array access out of bounds**.


Defect 53
Concurrency 5
Data-flow 12
Dynamic memory 6
Numerical 11
Other 3
Programming 7
Static memory 9
Arithmetic operation with NULL pointer 1
Array access out of bounds 3
numeric.c
programming.c
staticmemory.c
Invalid use of standard library memory routine 1
Invalid use of standard library string routine 1
Null pointer 1
Pointer access out of bounds 2

Expand **Array access out of bounds** to view all instances of this defect type.

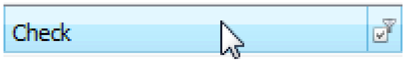
To see further information about an instance, select it. The information appears on the **Check Details** pane.

2 To view only the defects resulting from **Array access out of bounds**:

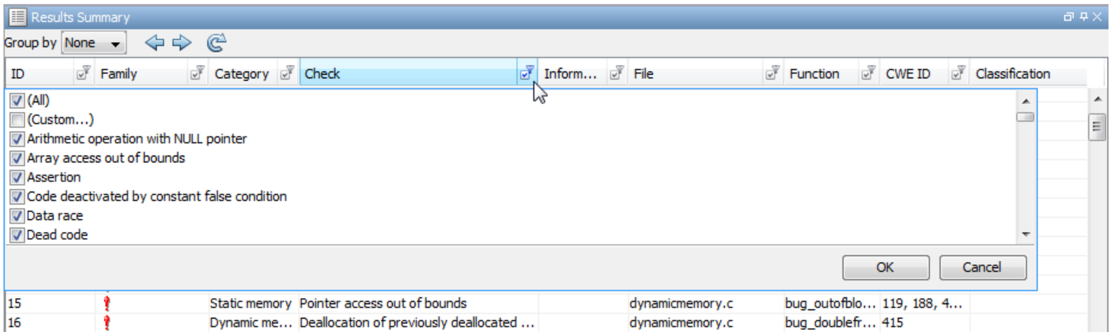
a On the **Results Summary** pane, select **Group by** > **None**.

The defects appear in a flat list.

b Click the filter icon on the **Check** column header.



A context menu lists the filter options available.



ID	Family	Category	Check	Inform...	File	Function	CWE ID	Classification
15	!	Static memory	Pointer access out of bounds		dynamicmemory.c	bug_outofblo...	119, 188, 4...	
16	!	Dynamic me...	Deallocation of previously deallocated ...		dynamicmemory.c	bug_doublefr...	415	

c Clear the **All** check box.

d Select the **Array access out of bounds** check box. Click **OK**.

The **Results Summary** pane displays only the defects resulting from the **Array access out of bounds** error.

Review New Results Only

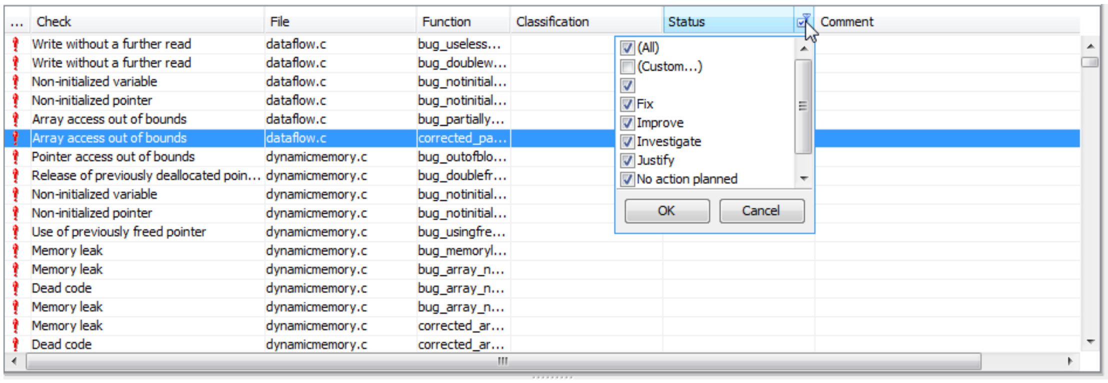
To review only new results found since the last analysis, on the **Results Summary** pane, select **New results**.

Review Defects with Given Status

To review only the defects with `Investigate` status:

1. Open the results file, with extension, `.psbf`.
2. On the **Results Summary** pane, place your cursor on the **Status** column head.
3. Click the filter icon.

A context menu lists the filter options available.



...	Check	File	Function	Classification	Status	Comment
	Write without a further read	dataflow.c	bug_useless...			
	Write without a further read	dataflow.c	bug_doublew...			
	Non-initialized variable	dataflow.c	bug_notinitial...			
	Non-initialized pointer	dataflow.c	bug_notinitial...			
	Array access out of bounds	dataflow.c	bug_partially...			
	Array access out of bounds	dataflow.c	corrected_pa...			
	Pointer access out of bounds	dynamicmemory.c	bug_outofblo...			
	Release of previously deallocated poin...	dynamicmemory.c	bug_doublefr...			
	Non-initialized variable	dynamicmemory.c	bug_notinitial...			
	Non-initialized pointer	dynamicmemory.c	bug_notinitial...			
	Use of previously freed pointer	dynamicmemory.c	bug_usingfre...			
	Memory leak	dynamicmemory.c	bug_memoryl...			
	Memory leak	dynamicmemory.c	bug_array_n...			
	Dead code	dynamicmemory.c	bug_array_n...			
	Memory leak	dynamicmemory.c	bug_array_n...			
	Memory leak	dynamicmemory.c	corrected_ar...			
	Dead code	dynamicmemory.c	corrected_ar...			

4 Clear the **All** check box.

5 Select the **Investigate** check box. Click **OK**.

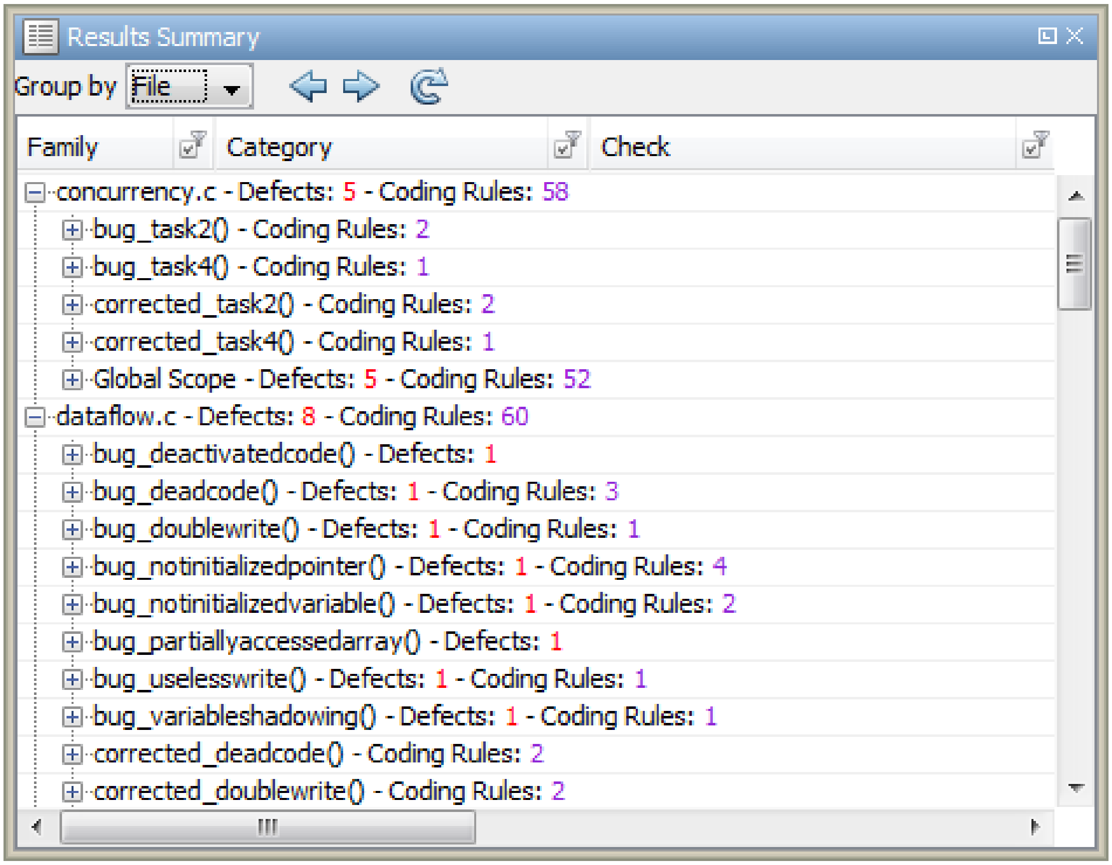
The **Results Summary** pane displays only the defects with the `Investigate` status.

Review Defects in a File

1 To review the defects in the file, `dataflow.c`:

a On the **Results Summary** pane, select **Group by** > **File**.

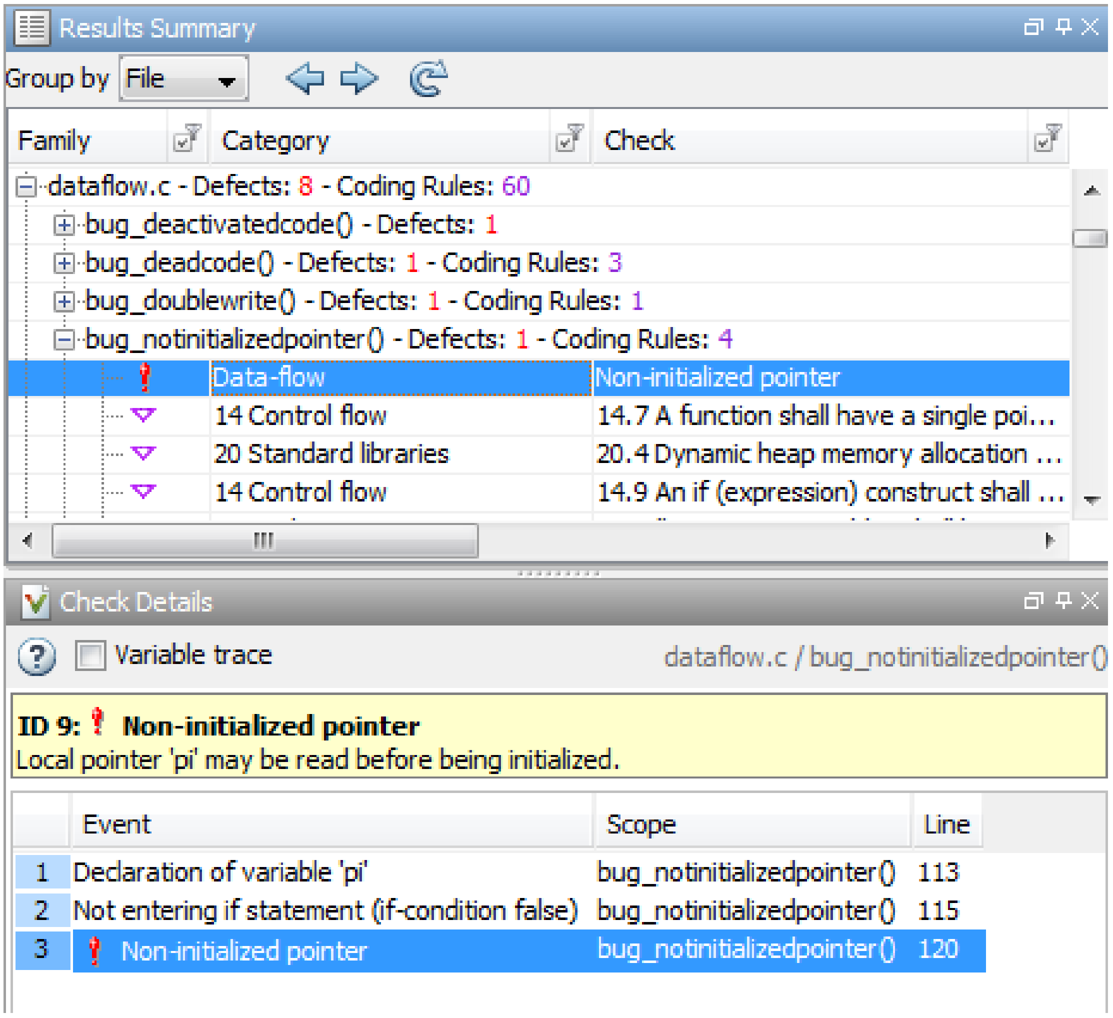
The defects displayed are grouped by files. The file names are sorted alphabetically. Within each file name, the defects are grouped by functions, sorted alphabetically.



concurrency.c - Defects: 5 - Coding Rules: 58
bug_task2() - Coding Rules: 2
bug_task4() - Coding Rules: 1
corrected_task2() - Coding Rules: 2
corrected_task4() - Coding Rules: 1
Global Scope - Defects: 5 - Coding Rules: 52
dataflow.c - Defects: 8 - Coding Rules: 60
bug_deactivatedcode() - Defects: 1
bug_deadcode() - Defects: 1 - Coding Rules: 3
bug_doublewrite() - Defects: 1 - Coding Rules: 1
bug_notinitializedpointer() - Defects: 1 - Coding Rules: 4
bug_notinitializedvariable() - Defects: 1 - Coding Rules: 2
bug_partiallyaccessedarray() - Defects: 1
bug_uselesswrite() - Defects: 1 - Coding Rules: 1
bug_variableshadowing() - Defects: 1 - Coding Rules: 1
corrected_deadcode() - Coding Rules: 2
corrected_doublewrite() - Coding Rules: 2

b To view the defects in `dataflow.c`, expand a function name under the category, **dataflow.c**.

To view further information on a bug, select the defect. Further information about the defect appears on the **Check Details** pane.


Results Summary

Group by File

Family	Category	Check
dataflow.c - Defects: 8 - Coding Rules: 60		
bug_deactivatedcode() - Defects: 1		
bug_deadcode() - Defects: 1 - Coding Rules: 3		
bug_doublewrite() - Defects: 1 - Coding Rules: 1		
bug_notinitializedpointer() - Defects: 1 - Coding Rules: 4		
!	Data-flow	Non-initialized pointer
▽	14 Control flow	14.7 A function shall have a single poi…
▽	20 Standard libraries	20.4 Dynamic heap memory allocation …
▽	14 Control flow	14.9 An if (expression) construct shall …

Check Details

Variable trace dataflow.c / bug_notinitializedpointer()

ID 9: ! Non-initialized pointer
Local pointer 'pi' may be read before being initialized.

	Event	Scope	Line
1	Declaration of variable 'pi'	bug_notinitializedpointer()	113
2	Not entering if statement (if-condition false)	bug_notinitializedpointer()	115
3	! Non-initialized pointer	bug_notinitializedpointer()	120

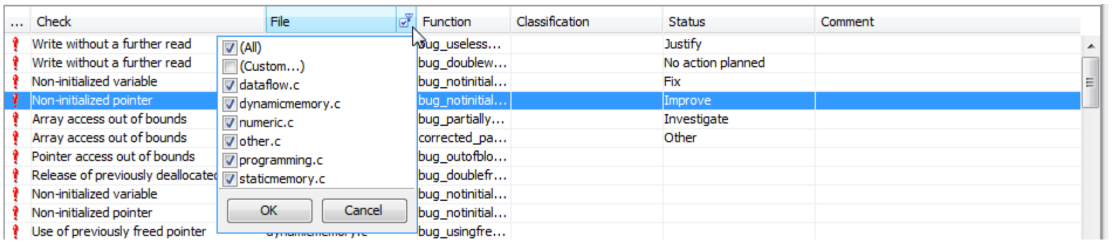
2 To view only the defects in `dataflow.c`:

a On the **Results Summary** pane, select **Group by** > **None**.

The **Results Summary** pane displays defects ungrouped.

b Click the filter icon on the **File** column head.

A context menu lists the filter options available.



...	Check	File	Function	Classification	Status	Comment
	Write without a further read		bug_useless...		Justify	
	Write without a further read		bug_doublew...		No action planned	
	Non-initialized variable		bug_notinitial...		Fix	
	Non-initialized pointer		bug_notinitial...		Improve	
	Array access out of bounds		bug_partially...		Investigate	
	Array access out of bounds		corrected_pa...		Other	
	Pointer access out of bounds		bug_outofblo...			
	Release of previously deallocated		bug_doublefr...			
	Non-initialized variable		bug_notinitial...			
	Non-initialized pointer		bug_notinitial...			
	Use of previously freed pointer	dynamicmemory.c	bug_usingfre...			

c Clear the **All** check box.

d Select the **dataflow.c** check box. Click **OK**.

The **Results Summary** pane displays only the defects in `dataflow.c`.

Tip If you apply a filter on a column on the **Results Summary** pane, the column header displays the number of rows suppressed.

Related Examples

- “Open Results”
- “Review and Comment Results”
- “Limit Display of Defects”

More About

- “Windows Used to Review Results”

Limit Display of Defects

This example shows how to control the number and type of defects displayed on the **Results Summary** pane. To reduce your review effort, you can limit the number of defects to display for certain checks or suppress them altogether.

To prevent the analysis from looking for some defects, see “Choose Specific Defects”.

If you want to change your analysis configuration, you can still change which defects are displayed in your results. There are two ways to filter defects from your results:

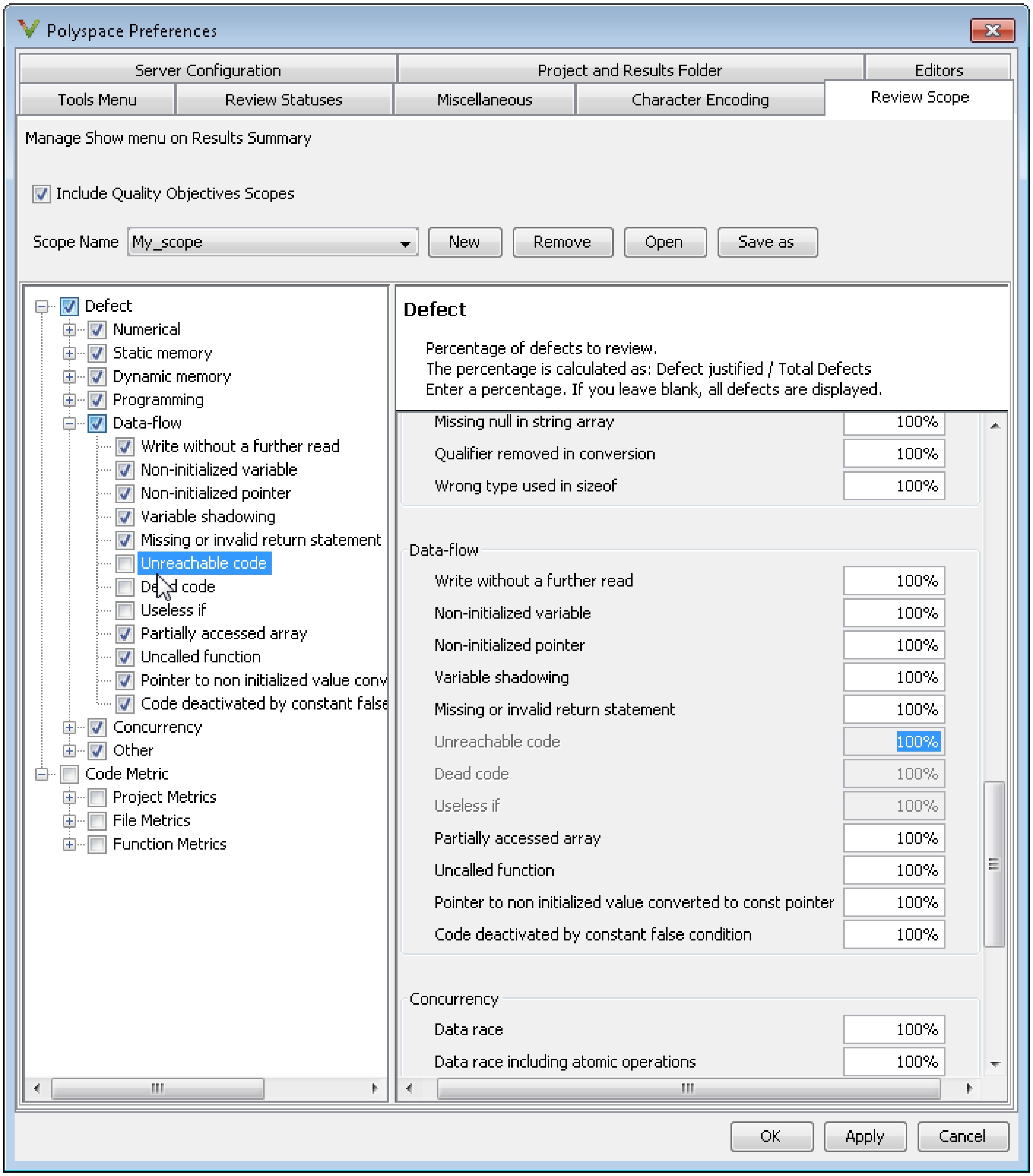
- Filter individual defects from display after each run.

  For more information, see “Filter and Group Results”.
- Create a set of filters that you can apply in one sweep.

This example shows the second approach.

1. Select **Tools** > **Preferences**.
2. On the **Review Scope** tab, create your filter file.
   - a. Select **New**. Save your filter file.
   - b. If you want a defect to be suppressed from **Results Summary**, on the left pane, under **Defect**, clear the box for the defect. Otherwise, on the right pane, specify a percentage of defects to display. The default is 100%.

     Instead of a percentage, you can specify a number or the string `ALL`. To specify a number, clear the box **Specify percentage of checks**.



Manage Show menu on Results Summary

[✓] Include Quality Objectives Scopes

Scope Name My_scope

- [✓] Defect
  - [✓] Numerical
  - [✓] Static memory
  - [✓] Dynamic memory
  - [✓] Programming
  - [✓] Data-flow
    - [✓] Write without a further read
    - [✓] Non-initialized variable
    - [✓] Non-initialized pointer
    - [✓] Variable shadowing
    - [✓] Missing or invalid return statement
    - [ ] Unreachable code
    - [ ] Dead code
    - [ ] Useless if
    - [✓] Partially accessed array
    - [✓] Uncalled function
    - [✓] Pointer to non initialized value conv
    - [✓] Code deactivated by constant false
  - [✓] Concurrency
  - [✓] Other
- [ ] Code Metric
  - [ ] Project Metrics
  - [ ] File Metrics
  - [ ] Function Metrics

Defect

Percentage of defects to review.
The percentage is calculated as: Defect justified / Total Defects
Enter a percentage. If you leave blank, all defects are displayed.

Missing null in string array	100%
Qualifier removed in conversion	100%
Wrong type used in sizeof	100%

Data-flow

Write without a further read	100%
Non-initialized variable	100%
Non-initialized pointer	100%
Variable shadowing	100%
Missing or invalid return statement	100%
Unreachable code	100%
Dead code	100%
Useless if	100%
Partially accessed array	100%
Uncalled function	100%
Pointer to non initialized value converted to const pointer	100%
Code deactivated by constant false condition	100%

Concurrency

Data race	100%
Data race including atomic operations	100%

3. Select **Apply** or **OK**.

   On the **Results Summary** pane, the **Show** menu displays additional options.

4. Select the option corresponding to the filters that you want. Only the number or percentage of defects that you specify remain on the **Results Summary** pane.

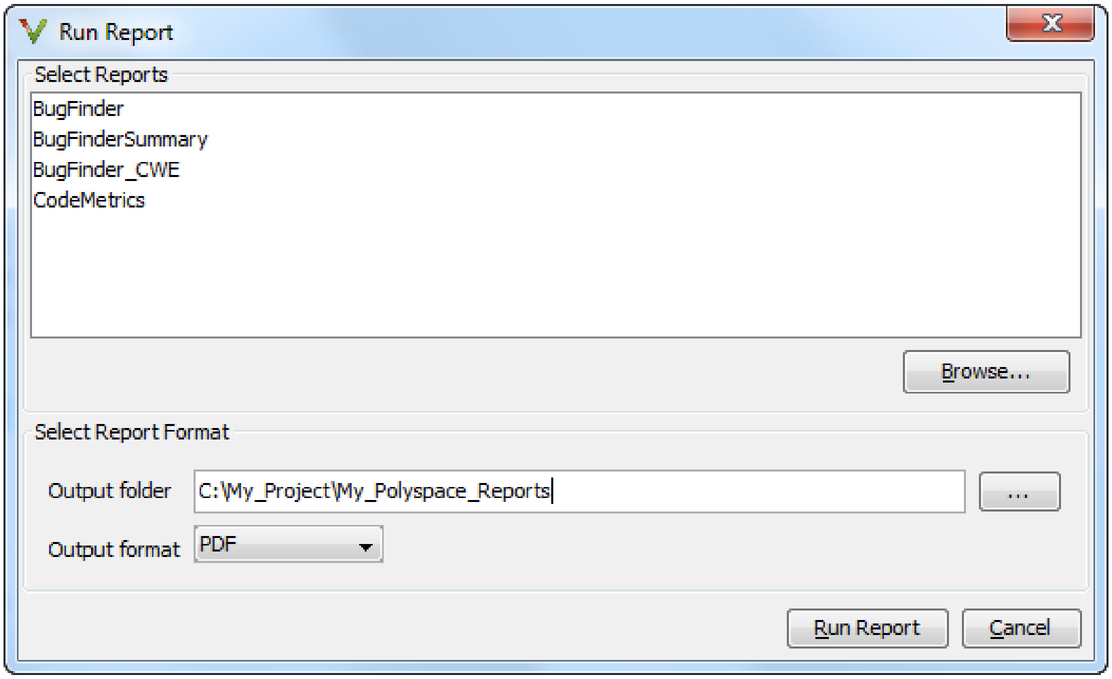
   - If you specify an absolute number, Polyspace displays that number of defects.
   - If you specify a percentage, Polyspace displays that percentage of the total number of defects.

Generate Reports

This example shows how to generate reports for a Polyspace Bug Finder analysis.

1 Open your results file.
2 Select **Reporting** > **Run Report**.

   The Run Report dialog box opens.



3 In the **Select Reports** section, select the types of reports that you want to generate. Press the **Ctrl** key to select multiple types. For example, you can select **BugFinder** and **CodeMetrics**.
4 Select the **Output folder** in which to save the report.
5 Select an **Output format** for the report.
6 Click **Run Report**.

The software creates the specified report and opens it.

See Also

“Generate report (C/C++)” | “Report template (C/C++)” | “Output format (C/C++)”

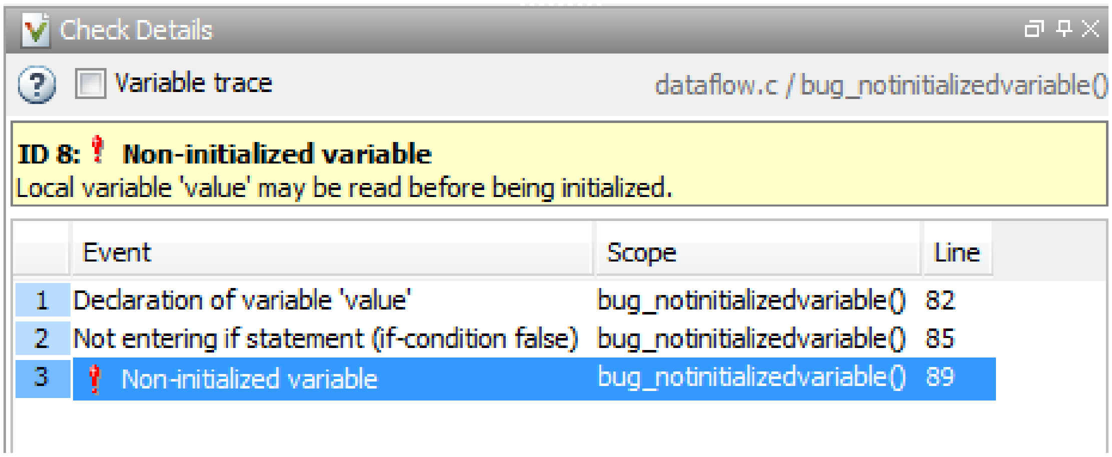
Review and Comment Results

This example shows how to review and comment your Bug Finder results. When reviewing results, you can assign a status to the defects and enter comments to describe the results of your review. These actions help you to track the progress of your review and avoid reviewing the same defect twice.

Review and Comment Individual Defect

1 On the **Results Summary** pane, select the defect that you want to review.

The **Check Details** pane displays information about the current defect.



ID 8: Non-initialized variable
Local variable 'value' may be read before being initialized.

	Event	Scope	Line
1	Declaration of variable 'value'	bug_notinitializedvariable()	82
2	Not entering if statement (if-condition false)	bug_notinitializedvariable()	85
3	Non-initialized variable	bug_notinitializedvariable()	89

2 On the **Results Summary** pane, enter a **Classification** for the defect to describe its severity:

- `High`
- `Medium`
- `Low`
- `Not a defect`

3 On the **Results Summary** pane, enter a **Status** to describe how you intend to address the defect:

- `Fix`
- `Improve`

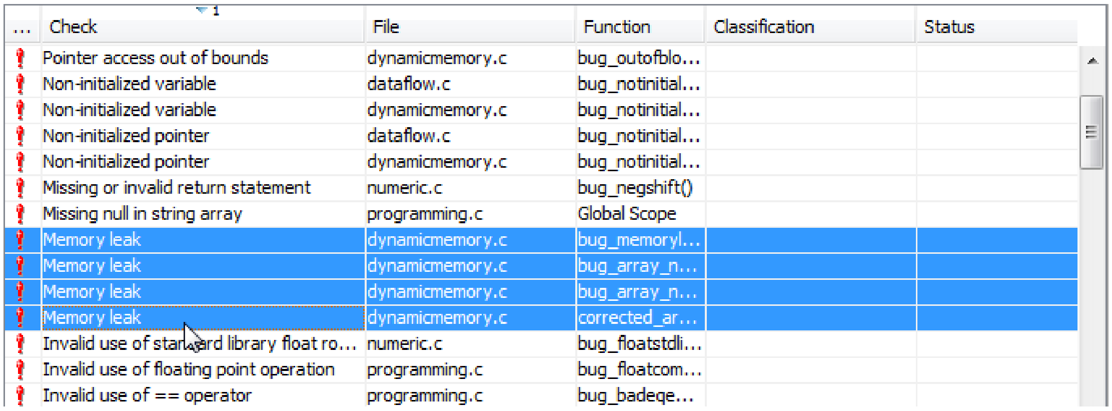
- `Investigate`
- `Justified`
- `No action planned`
- `Other`

4 On the **Results Summary** pane, enter remarks in the **Comment** field, for example, defect or justification information.

Review and Comment Group of Defects

1 On the **Results Summary** pane, select a group of defects using one of the following methods:

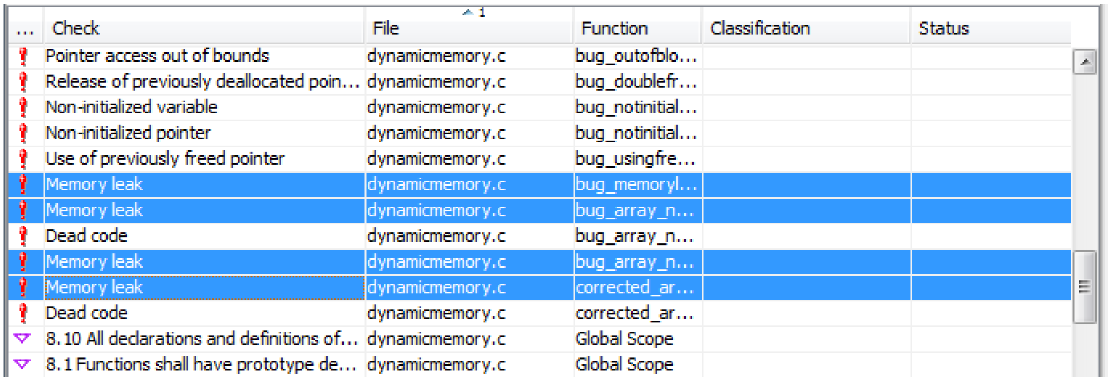
- For contiguous defects, select the first defect. Then **Shift**-select the last defect.



...	Check	File	Function	Classification	Status
❗	Pointer access out of bounds	dynamicmemory.c	bug_outofblo...		
❗	Non-initialized variable	dataflow.c	bug_notinitial...		
❗	Non-initialized variable	dynamicmemory.c	bug_notinitial...		
❗	Non-initialized pointer	dataflow.c	bug_notinitial...		
❗	Non-initialized pointer	dynamicmemory.c	bug_notinitial...		
❗	Missing or invalid return statement	numeric.c	bug_negshift()		
❗	Missing null in string array	programming.c	Global Scope		
❗	Memory leak	dynamicmemory.c	bug_memoryl...		
❗	Memory leak	dynamicmemory.c	bug_array_n...		
❗	Memory leak	dynamicmemory.c	bug_array_n...		
❗	Memory leak	dynamicmemory.c	corrected_ar...		
❗	Invalid use of standard library float ro...	numeric.c	bug_floatstdli...		
❗	Invalid use of floating point operation	programming.c	bug_floatcom...		
❗	Invalid use of == operator	programming.c	bug_badeqe...		

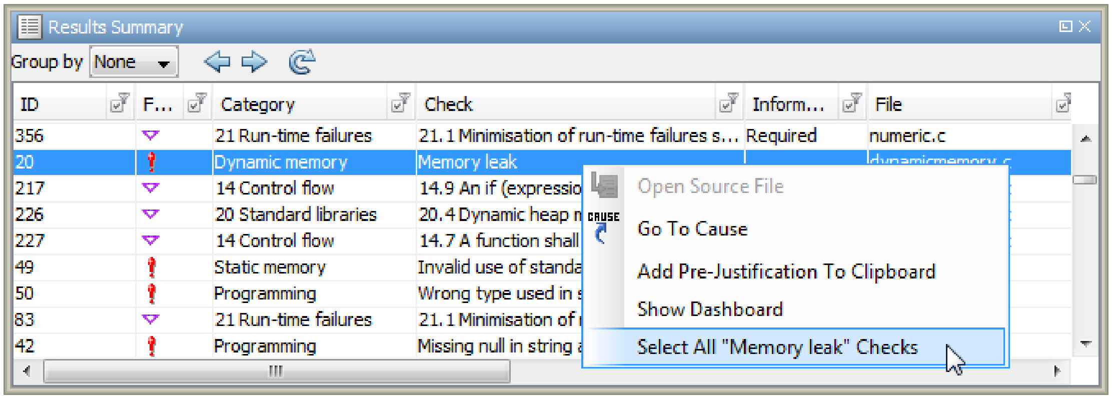
To group together the defects that belong to a certain category, click the **Check** column header on the **Results Summary** pane.

- For non-contiguous defects, **Ctrl**-select each defect.



...	Check	File	Function	Classification	Status
!	Pointer access out of bounds	dynamicmemory.c	bug_outofblo...		
!	Release of previously deallocated poin...	dynamicmemory.c	bug_doublefr...		
!	Non-initialized variable	dynamicmemory.c	bug_notinitial...		
!	Non-initialized pointer	dynamicmemory.c	bug_notinitial...		
!	Use of previously freed pointer	dynamicmemory.c	bug_usingfre...		
!	Memory leak	dynamicmemory.c	bug_memoryl...		
!	Memory leak	dynamicmemory.c	bug_array_n...		
!	Dead code	dynamicmemory.c	bug_array_n...		
!	Memory leak	dynamicmemory.c	bug_array_n...		
!	Memory leak	dynamicmemory.c	corrected_ar...		
!	Dead code	dynamicmemory.c	corrected_ar...		
▽	8.10 All declarations and definitions of...	dynamicmemory.c	Global Scope		
▽	8.1 Functions shall have prototype de...	dynamicmemory.c	Global Scope		

- For defects of a similar category, right-click one defect from that category. From the context menu, select **Select All *`Defect Category`* Checks**, for example, **Select All "Memory leak" Checks**.



Results Summary

Group by None

ID	F...	Category	Check	Inform...	File
356	▽	21 Run-time failures	21.1 Minimisation of run-time failures s...	Required	numeric.c
20	!	Dynamic memory	Memory leak		dynamicmemory.c
217	▽	14 Control flow	14.9 An if (expressio		
226	▽	20 Standard libraries	20.4 Dynamic heap n		
227	▽	14 Control flow	14.7 A function shall		
49	!	Static memory	Invalid use of standa		
50	!	Programming	Wrong type used in s		
83	▽	21 Run-time failures	21.1 Minimisation of r		
42	!	Programming	Missing null in string a		

Open Source File
CAUSE
Go To Cause
Add Pre-Justification To Clipboard
Show Dashboard
Select All "Memory leak" Checks

2 On the **Results Summary** pane, enter **Classification**, **Status** and **Comments**. The software applies this information to all the selected defects.

Save Review Comments

After you have reviewed your results, save your comments with the analysis results. Saving your comments makes them available the next time that you open the results file, allowing you to avoid reviewing the same check twice.

To save your review comments, select **File** > **Save**. Your comments are saved with the analysis results.

Related Examples

- "Open Results"
- "Filter and Group Results"
- "Copy and Paste Annotations"

More About

- "Windows Used to Review Results"

Review Code Metrics

Polyspace does not compute code complexity metrics by default. To compute them during analysis, do the following:

- **User interface**: On the **Configuration** pane, select **Advanced Settings**. Select **Calculate Code Metrics**.
- **Command line**: Use the option `-code-metrics` with the `polyspace-bug-finder-nodesktop` command.

After analysis, the software displays code complexity metrics on the **Results Summary** pane. You can:

- Specify limits for the metric values through **Tools** > **Preferences**.

  If you impose limits on metrics, the **Results Summary** pane displays only those metric values that violate the limits. Use predefined limits or assign your own limits. If you assign your own limits, you can share the limits file to enforce coding standards in your organization.
- Justify the value of a metric.

  If a metric value exceeds specified limits and appears red, you can add a comment with the rationale.

You can also suppress code metrics from the **Results Summary** display. Select **Show** > **Defects & Rules**.

For information on the metrics, see “Code Metrics”.

In this section...

Impose Limits on Metrics

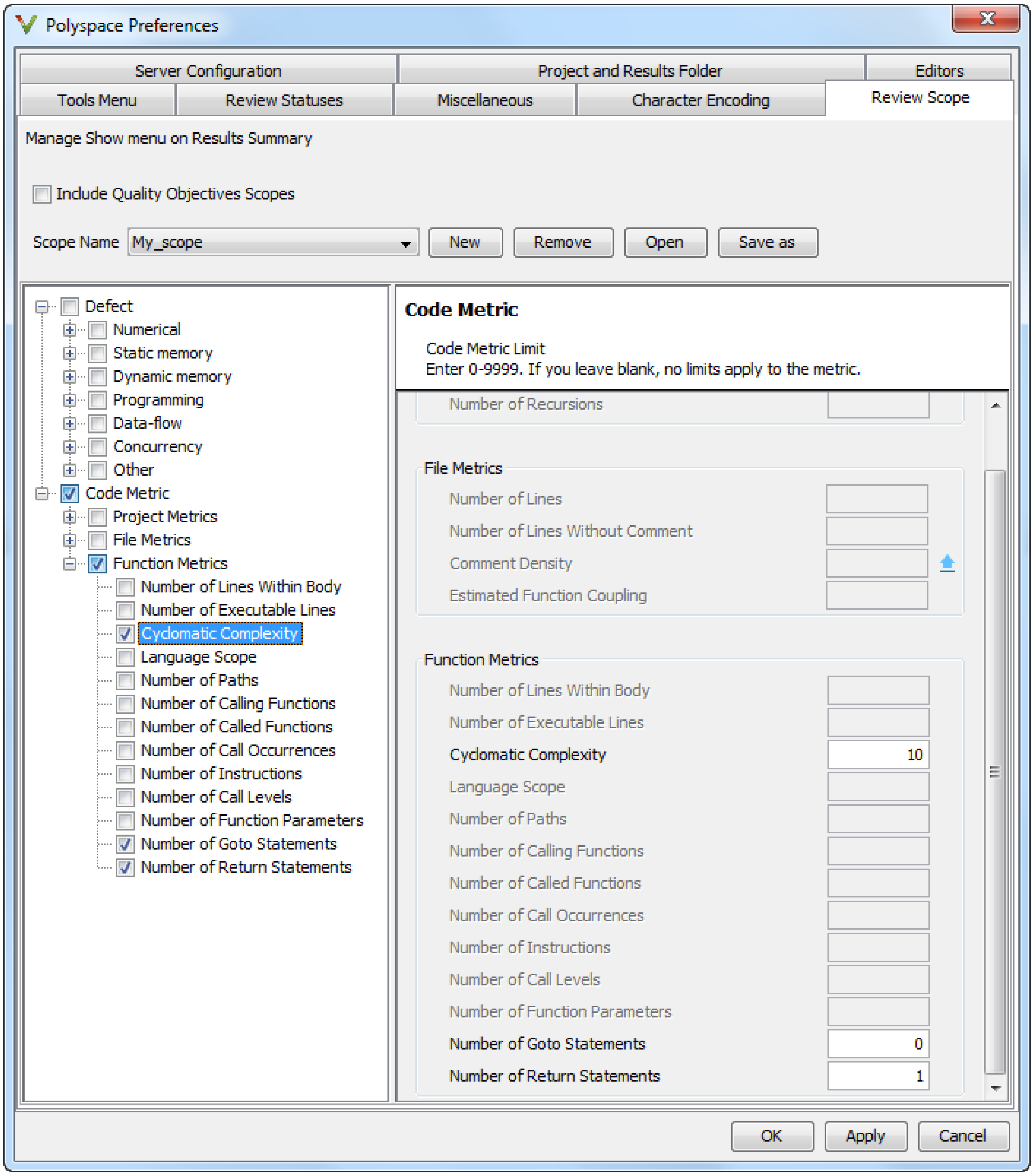
1 Select **Tools** > **Preferences**.
2 On the **Review Scope** tab, do one of the following:

   - To use a predefined limit, select **Include Quality Objectives Scopes**.

The **Scope Name** list shows the additional option `HIS`. Select the option to see the limit values.

- To define your own limits, select **New**. Save your limits file.

  On the left pane, under **Code Metric**, select the box for a metric. On the right pane, specify a limit value for the metric. Other than **Comment Density**, limit values are upper limits.



Manage Show menu on Results Summary

☐ Include Quality Objectives Scopes

Scope Name: My_scope

- ☐ Defect
  - ☐ Numerical
  - ☐ Static memory
  - ☐ Dynamic memory
  - ☐ Programming
  - ☐ Data-flow
  - ☐ Concurrency
  - ☐ Other
- ☑ Code Metric
  - ☐ Project Metrics
  - ☐ File Metrics
  - ☑ Function Metrics
    - ☐ Number of Lines Within Body
    - ☐ Number of Executable Lines
    - ☑ Cyclomatic Complexity
    - ☐ Language Scope
    - ☐ Number of Paths
    - ☐ Number of Calling Functions
    - ☐ Number of Called Functions
    - ☐ Number of Call Occurrences
    - ☐ Number of Instructions
    - ☐ Number of Call Levels
    - ☐ Number of Function Parameters
    - ☑ Number of Goto Statements
    - ☑ Number of Return Statements

Code Metric

Code Metric Limit
Enter 0-9999. If you leave blank, no limits apply to the metric.

Number of Recursions	

File Metrics

Number of Lines	
Number of Lines Without Comment	
Comment Density	
Estimated Function Coupling	

Function Metrics

Number of Lines Within Body	
Number of Executable Lines	
Cyclomatic Complexity	10
Language Scope	
Number of Paths	
Number of Calling Functions	
Number of Called Functions	
Number of Call Occurrences	
Number of Instructions	
Number of Call Levels	
Number of Function Parameters	
Number of Goto Statements	0
Number of Return Statements	1

3 Select **Apply** or **OK**.

   On the **Results Summary** pane, the **Show** menu displays additional options.

   - If you use predefined limits, the option `HIS` appears. This option displays code metrics only.
   - If you define your own limits, the option corresponding to your limits file name appears.

4 Select the option corresponding to the limits that you want. Only metric values that violate your limits appear on the **Results Summary** pane.

Note: To enforce coding standards across your organization, share your limits file that you saved in XML format.

People in your organization can use the **Open** button on the **Review Scope** tab and navigate to the location of the XML file.

Comment and Justify Limit Violations

Once you use the **Show** menu to display only metrics that violate limits, you can review each violation.

1 On the **Results Summary** pane, select **Group by** > **Family**.

   The code metrics appear together under one node.

2 Expand the node. Select each violation.

   - On the **Results Summary** pane, in the **Information** column, you can see the metric value.
   - On the **Check Details** pane, you can see the metric value and a brief description of the metric.

     For more detailed descriptions and examples, select the ? icon.

3 On the **Results Summary** pane, add a comment and justification describing why the violation occurs. For more information, see "Review and Comment Results".

Import Comments from Previous Analyses

This example shows how to import review comments from previous analyses. By default, Polyspace Bug Finder automatically imports comments from the previous analysis, allowing you to avoid reviewing the same defect twice. However, you can also manually import comments into the current review

Import Comments from Previous Analysis

1. Open your most recent results.
2. Select **Tools** > **Import Comments**.
3. Navigate to the folder containing your previous results.
4. Select the results file with extension `.psbf`, then click **Open**.

   The review comments from the previous results are imported into the current results, and the Import checks and comments report opens.

Change Preferences for Automatically Importing Comments

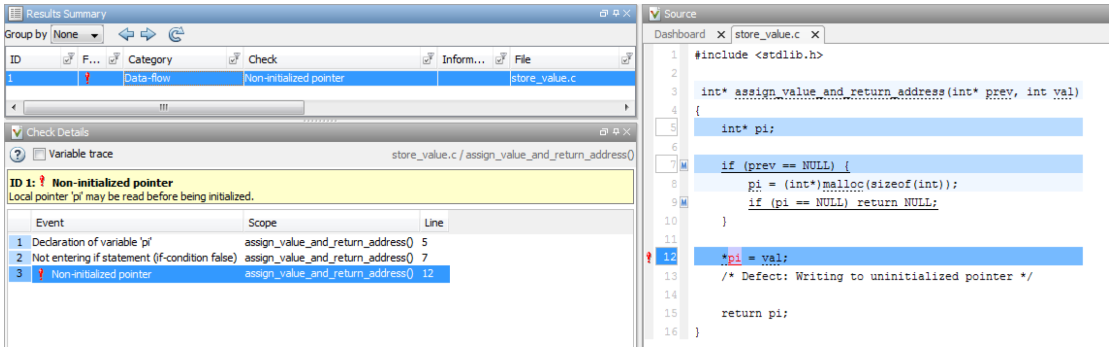
1. Select **Tools** > **Preferences**, which opens the Polyspace Preferences dialog box.
2. Select the **Project and Results Folder** tab.
3. Under **Import comments**, select or clear **Automatically import comments from last verification**.
4. Click **OK**.

View Code Sequence Causing Defect

This example shows how to view the code sequence that is probably causing a defect. The example uses the following code, which contains a `Non-initialized pointer` defect:

```
#include <stdlib.h>

 int* assign_value_and_return_address(int* prev, int val)
{
    int* pi;

    if (prev == NULL) {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
    }

    *pi = val;
    /* Defect: Writing to uninitialized pointer */

    return pi;
}
```
The code is stored in a source file `store_value.c`.

1 Run a Polyspace Bug Finder analysis on `store_value.c`.
2 Open the results file with extension `.psbf`.
3 On the **Results Summary** pane, select the **Non-initialized pointer** defect.



ID	F...	Category	Check	Inform...	File
1	❗	Data-flow	Non-initialized pointer		store_value.c

store_value.c / assign_value_and_return_address()

ID 1: ❗ Non-initialized pointer
Local pointer 'pi' may be read before being initialized.

	Event	Scope	Line
1	Declaration of variable 'pi'	assign_value_and_return_address()	5
2	Not entering if statement (if-condition false)	assign_value_and_return_address()	7
3	❗ Non-initialized pointer	assign_value_and_return_address()	12

```
1  #include <stdlib.h>
2
3   int* assign_value_and_return_address(int* prev, int val)
4  {
5      int* pi;
6
7      if (prev == NULL) {
8          pi = (int*)malloc(sizeof(int));
9          if (pi == NULL) return NULL;
10     }
11
12     *pi = val;
13     /* Defect: Writing to uninitialized pointer */
14
15     return pi;
16 }
```

- The code line containing the defect is highlighted in dark blue on the **Source** pane. More information on the defect is available on the **Check Details** pane.

- The following columns describe the sequence of code instructions causing the defect:

  a. **Event**: Instruction causing the defect
  b. **Scope**: Function containing instruction
  c. **Line**: Line number of instruction

  These instructions are also highlighted in medium blue on the **Source** pane. The corresponding line numbers are marked by squares. Place your cursor over a square to view a tooltip. The tooltip describes how the instruction is possibly related to the defect.
- Other instructions that can possibly impact the defect are highlighted in light blue on the **Source** pane. To see these instructions on the **Check Details** pane, select the **Variable trace** check box.

4. To navigate to an instruction from the probable code sequence in the source code, select the instruction on the **Event** column. The corresponding line is highlighted on the **Source** pane.

Related Examples

- "Run Local Analysis"
- "View Results Summary in Polyspace Metrics"
- "Review and Comment Results"

More About

- "Source"
- "Check Details"

Results Folder Contents

Every time you run an analysis, Polyspace generates files and folders that contain information about configuration options and analysis results. The contents of results folders depend on the configuration options and how the analysis was started.

By default, your results are saved in your project folder in a folder called `Result`. To use a different folder, see "Specify Results Folder".

Files in the Results Folder

Some of the files and folders in the results folder are described below:

- `Polyspace_`*`release_project_name_date-time`*`.log` — A log file associated with each analysis.
- `ps_results.psbf` — An encrypted file containing your Polyspace results. Open this file in the Polyspace environment to view your results.
- `ps_sources.db` — A non-encrypted database file listing source files and macros.
- `drs-template.xml` — A template generated when you use constraint specification.
- `ps_comments.db` — An encrypted database file containing your comments and justifications.
- `comments_bak` — A subfolder used to import comments between results.
- `.status` and `.settings` — Two folders used to store files needed to relaunch the analysis.
- `Polyspace-Doc` — When you generate a report, by default, your report is saved in this folder with the name *`ProjectName_ReportType`*. For example, a developer report in Rich Text Format would be, `myProject_Developer.rtf`.

See Also

-results-dir

Related Examples

- "Specify Results Folder"
- "Open Results"

Windows Used to Review Results

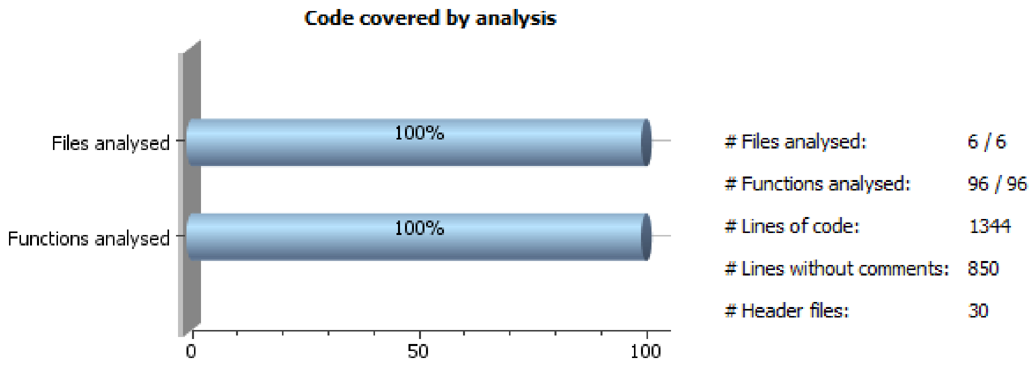
In this section...

Dashboard

On the **Source** pane, the **Dashboard** tab provides statistics on the analysis results in a graphical format.

When you open a results file in Polyspace, this tab is displayed by default. You can view the following graphs:

- **Code covered by analysis**



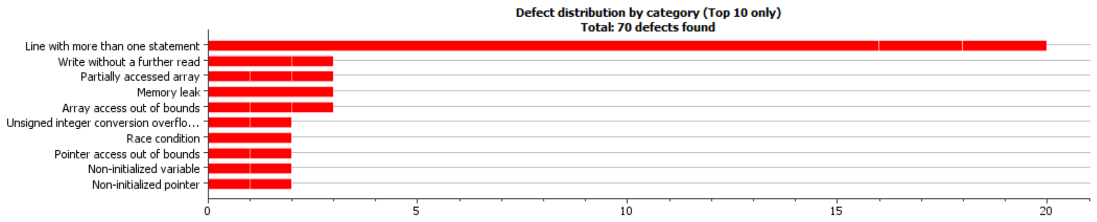
# Files analysed:	6 / 6
# Functions analysed:	96 / 96
# Lines of code:	1344
# Lines without comments:	850
# Header files:	30

From this graph you can obtain the following information:

- **# Files analyzed**: Ratio of analyzed files to total number of files. If a file contains a compilation error, Polyspace Bug Finder does not analyze the file.

- **# Functions analyzed**: Ratio of analyzed functions to total number of functions in the analyzed files. If the analysis of a function takes longer than a certain threshold value, Polyspace Bug Finder does not analyze the function.
- **# Lines of code**: Total number of code lines in source files.
- **# Lines without comments**: Total number of code lines in source files excluding lines that are only comments.
- **# Header files**: Total number of files included in your source files using `#include` directive.

- **Defect distribution by category or file**

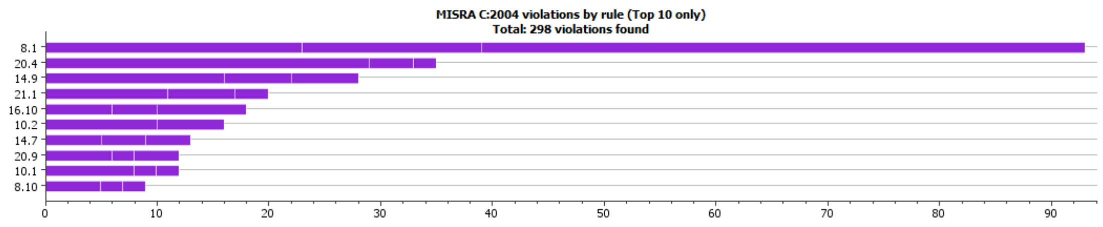


From this graph you can obtain the following information.

	Category	File
Top 10	The ten defect types with the highest number of individual defects. • Each column represents a defect type and is divided into the: • File with highest number of defects of this type. • File with second highest number of defects of this type. • All other files with defects of this type. Place your cursor on a column to see the file name and number of defects of this type in this file.	The ten source files with the highest number of defects. • Each column represents a file and is divided into the: • Defect type with highest number of defects in this file. • Defect type with second highest number of defects in this file. • All other defect types in this file. Place your cursor on a column to see the defect type name and number of defects of this type in this file.

	Category	File
	• The x-axis represents the number of defects. Use this view to organize your check review starting at defect types with more individual defects.	• The x-axis represents the number of defects. Use this view to organize your check review starting at files with more defects.
Bottom 10	The ten defect types with the lowest number of individual defects. Each column on the graph is divided the same way as the **Top 10** defect types. Use this view to organize your check review starting at defect types with fewer individual defects.	The ten source files with the lowest number of defects. Each column on the graph is divided the same way as the **Top 10** files. Use this view to organize your check review starting at files with fewer defects.

- **Coding rule violations by rule or file**



For every type of coding rule that you check (MISRA, JSF, or custom), the **Dashboard** contains a graph of the rule violations.

From this graph you can obtain the following information.

	Category	File
Top 10	The ten rules with the highest number of violations. • Each column represents a rule number and is divided into the:	The ten source files containing the highest number of violations. • Each column represents a file and is divided into the:

	Category	File
	• File with highest number of violations of this rule. • File with second highest number of violations of this rule. • All other files with violations of this rule. Place your cursor on a column to see the file name and number of violations of this rule in the file. • The x-axis represents the number of rule violations. Use this view to organize your review starting at rules with more violations.	• Rule with highest number of violations in this file. • Rule with second highest number of violations in this file. • All other rules violated in this file. Place your cursor on a column to see the rule number and number of violations of the rule in this file. • The x-axis represents the number of rule violations. Use this view to organize your review starting at files with more rule violations.
Bottom 10	The ten rules with the lowest number of violations. Each column on the graph is divided in the same way as the **Top 10** rules. Use this view to organize your review starting at rules with fewer violations.	The ten source files containing the lowest number of rule violations. Each column on the graph is divided in the same way as the **Top 10** files. Use this view to organize your review starting at files with fewer rule violations.

For a list of supported coding rules, see "Supported MISRA C:2004 Rules", "Supported MISRA C++ Coding Rules" and "Supported JSF C++ Coding Rules".

Results Summary

The **Results Summary** pane lists all defects along with their attributes. To organize your results review, from the **Group by** list on this pane, select one of the following options:

- **None**: Lists defects and coding rule violations in alphabetical order.
- **Family**: Lists results grouped by category. For more information on the defects covered by a category, see "Polyspace Bug Finder Results".

- **Class**: Lists results grouped by class. Within each class, the results are grouped by method. The first group, **Global Scope**, lists results not occurring in a class definition.

  This option is available for `C++` code only.
- **File**: Lists results grouped by file. Within each file, the results are grouped by function.

For each defect, the **Results Summary** pane contains the defect attributes, listed in columns:

Attribute	Description
Family	Group to which the defect belongs.
ID	Unique identification number of the defect. In the default view on the **Results Summary** pane, the defects appear sorted by this number.
Type	Defect or coding rule violation.
Category	Category of the defect. For more information on the defects covered by a category, see “Polyspace Bug Finder Results”.
Check	Description of the defect
File	File containing the instruction where the defect occurs
Class	Class containing the instruction where the defect occurs. If the defect is not inside a class definition, then this column contains the entry, **Global Scope**.
Function	Function containing the instruction where the defect occurs. If the function is a method of a class, it appears in the format `*class_name*::*function_name*`.
Classification	Level of severity you have assigned to the defect. The possible levels are: • `High`

Attribute	Description
	• `Medium` • `Low` • `Not a defect`
Status	Review status you have assigned to the check. The possible statuses are: • `Fix` • `Improve` • `Investigate` • `Justified` • `No action planned` • `Other`
Comments	Comments you have entered about the check

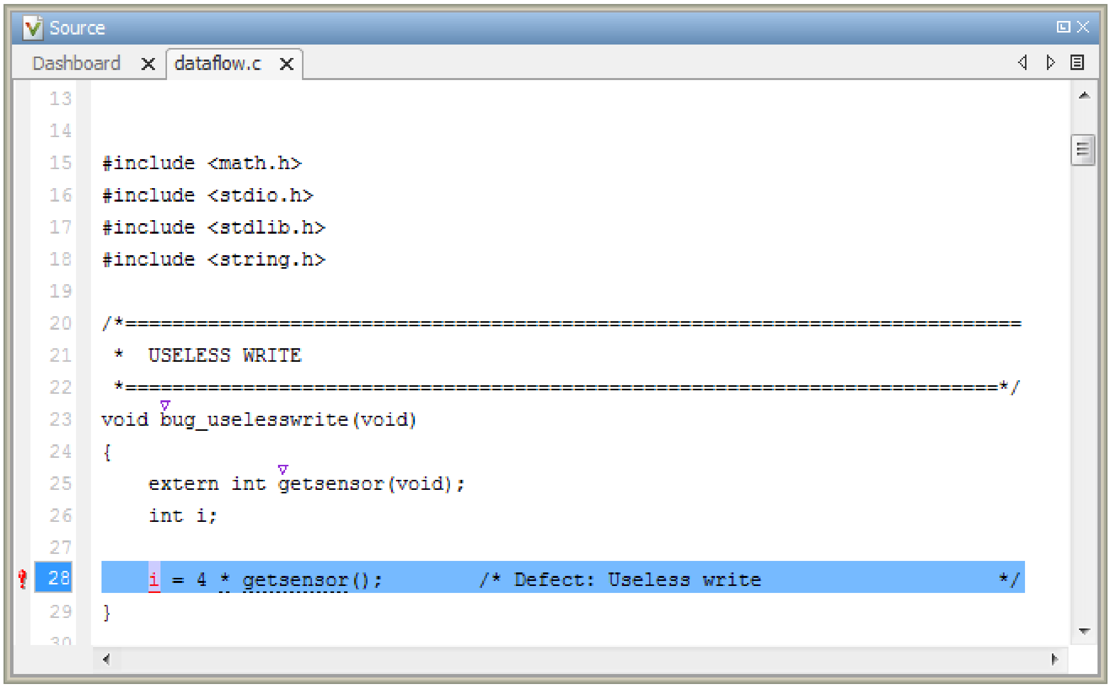
To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the checks. For more information, see “Review and Comment Results”.
- Organize your check review using filters on the columns. For more information, see “Filter and Group Results”.

Source

The **Source** pane shows the source code with the defects colored in red and the corresponding line number marked by !.



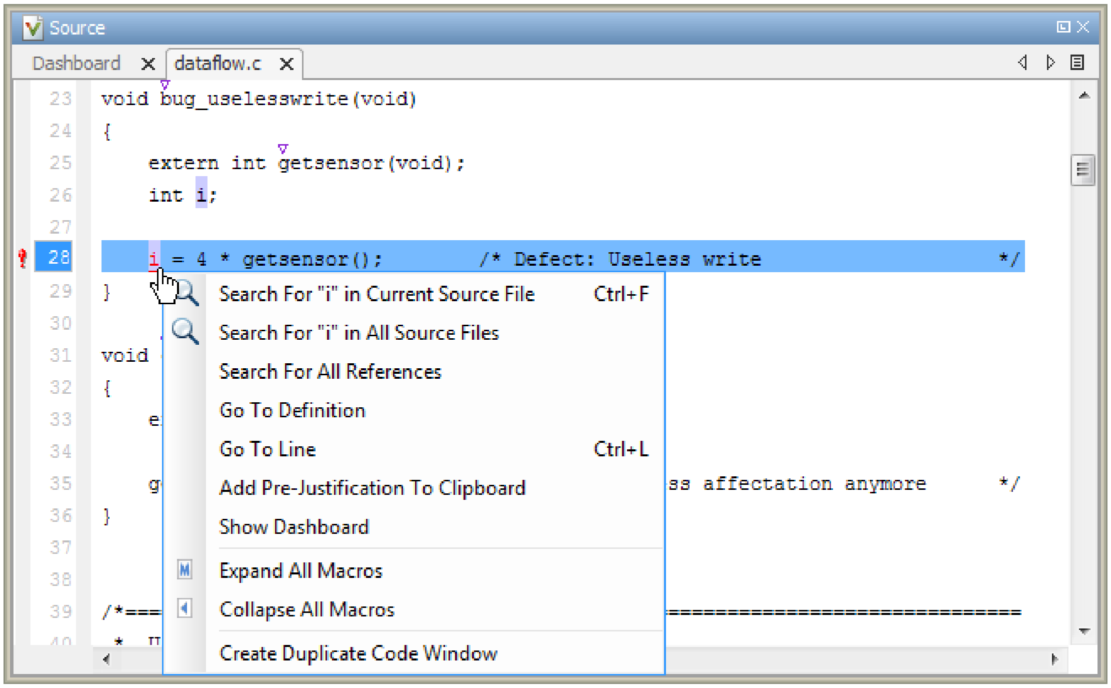
```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*==========================================================================
 *  USELESS WRITE
 *========================================================================*/
void bug_uselesswrite(void)
{
    extern int getsensor(void);
    int i;

    i = 4 * getsensor();         /* Defect: Useless write                */
}
```

Tooltips

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

Examine Source Code

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


Source
Dashboard
dataflow.c
Search For "i" in Current Source File Ctrl+F
Search For "i" in All Source Files
Search For All References
Go To Definition
Go To Line Ctrl+L
Add Pre-Justification To Clipboard
Show Dashboard
Expand All Macros
Collapse All Macros
Create Duplicate Code Window

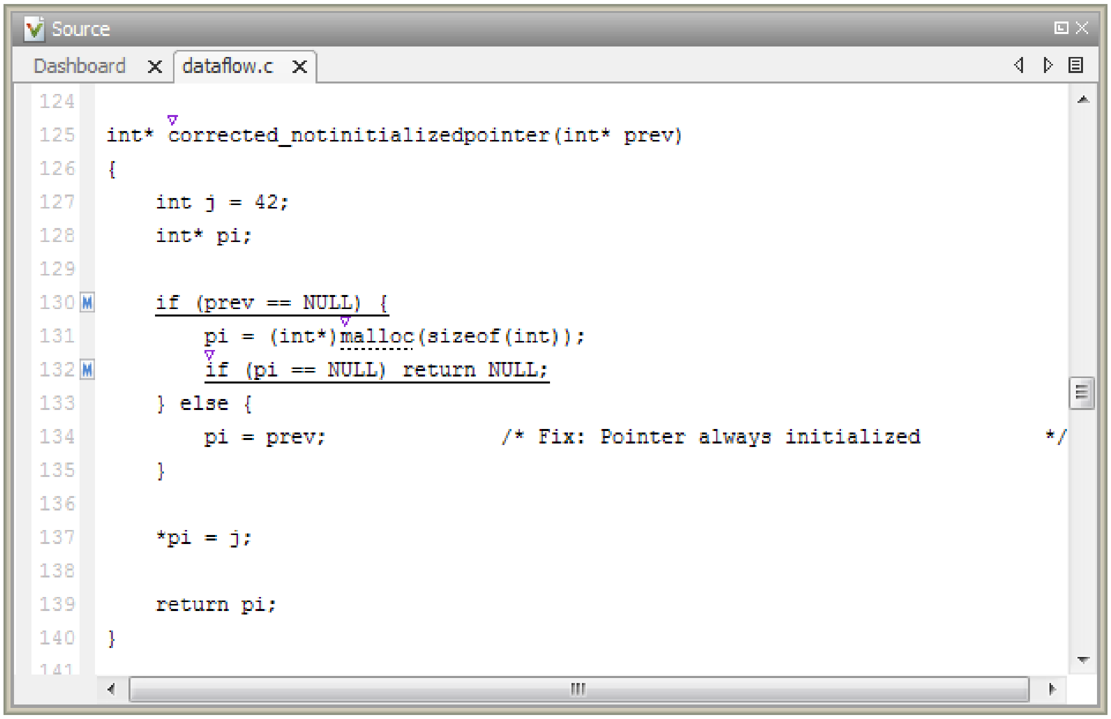
```
void bug_uselesswrite(void)
{
    extern int getsensor(void);
    int i;

    i = 4 * getsensor();         /* Defect: Useless write                */
}

void
{
    e

    g                                ss affectation anymore      */
}


/*===                             ==============================
```

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

- **Search "i" in Current Source** — List occurrences of the string within the current source file on the **Search** pane.
- **Search "i" in All Source Files** — List occurrences of the string within the source files on the **Search** pane.
- **Search For All References** — List all references in the **Search** pane. The software supports this feature for global and local variables, functions, types, and classes.
- **Go To Definition** — Go to the line of code that contains the definition of `i`. The software supports this feature for global and local variables, functions, types, and classes.
- **Go To Line** — Open the Go to line dialog box. If you specify a line number and click **Enter**, the software displays the specified line of code.
- **Expand All Macros** or **Collapse All Macros** — Display or hide the content of macros in current source file.

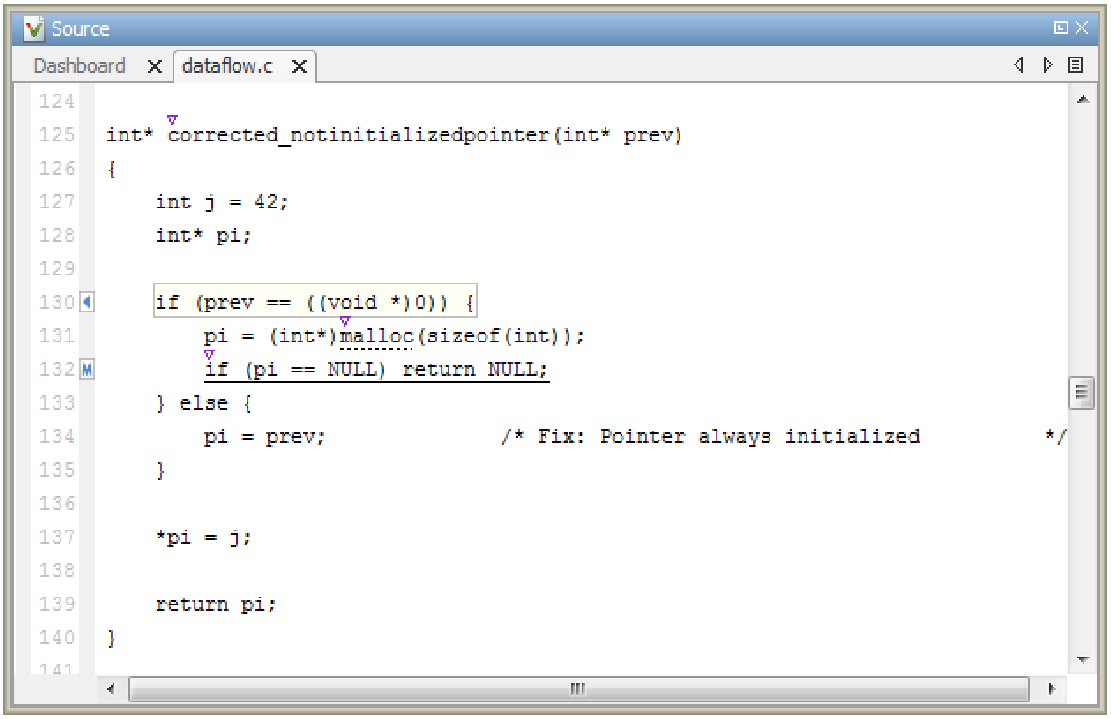
Expand Macros

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



```
int* corrected_notinitializedpointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == NULL) {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
    } else {
        pi = prev;              /* Fix: Pointer always initialized         */
    }

    *pi = j;

    return pi;
}
```

When you click a line with this icon, the software displays the contents of macros on that line in a box.



```
int* corrected_notinitializedpointer(int* prev)
{
    int j = 42;
    int* pi;

    if (prev == ((void *)0)) {
        pi = (int*)malloc(sizeof(int));
        if (pi == NULL) return NULL;
    } else {
        pi = prev;              /* Fix: Pointer always initialized         */
    }

    *pi = j;

    return pi;
}
```

To display the normal source code again, click the line away from the box, for example, on the ◂ icon.

To display or hide the content of *all* macros:

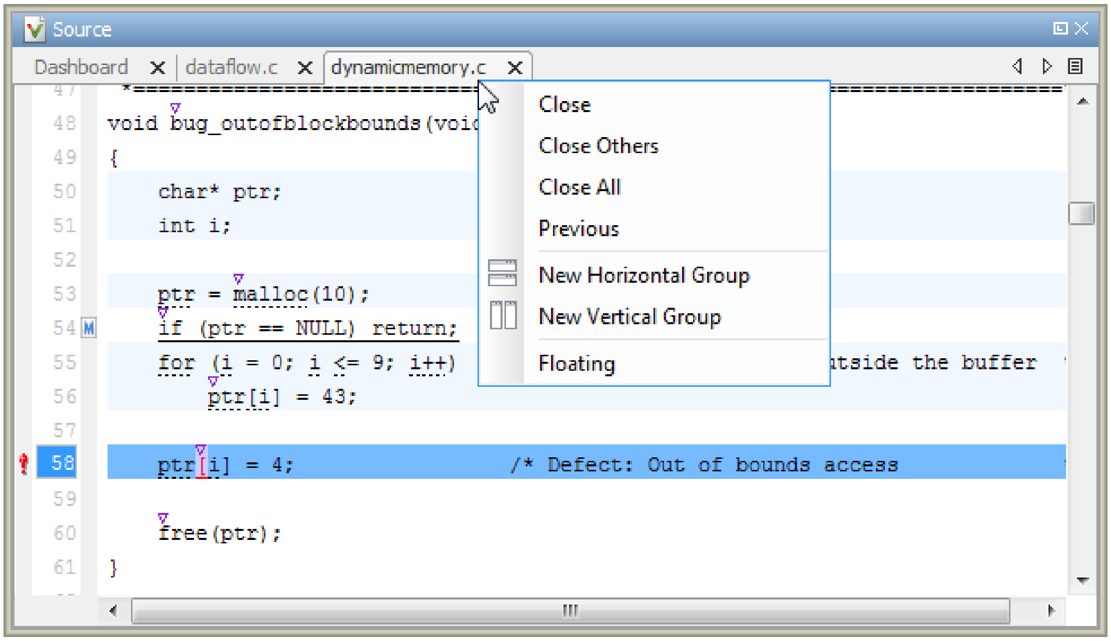
1 Right-click anywhere on the source.
2 From the context menu, select either **Expand All Macros** or **Collapse All Macros**.

Note: The **Check Details** pane also allows you to view the contents of a macro if the check you select lies within a macro.

Manage Multiple Files in Source Pane

You can view multiple source files in the **Source** pane.

Right-click on the **Source** pane toolbar.



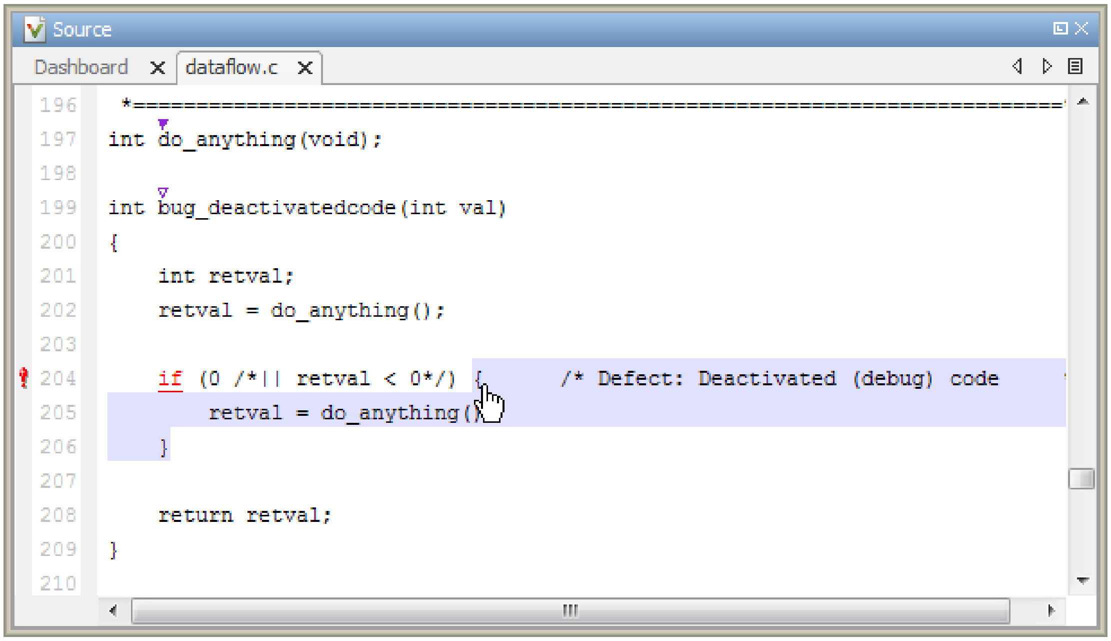
```
 *============================                    =====================
void bug_outofblockbounds(voic
{
    char* ptr;
    int i;

    ptr = malloc(10);
    if (ptr == NULL) return;
    for (i = 0; i <= 9; i++)                         ıtside the buffer
        ptr[i] = 43;

    ptr[i] = 4;                 /* Defect: Out of bounds access

    free(ptr);
}
```

From the **Source** pane context menu, you can:

- **Close** – Close the currently selected source file. You can also use the χ button to close tabs.
- **Close Others** – Close all source files except the currently selected file.
- **Close All** – Close all source files.
- **Next** – Display the next view.
- **Previous** – Display the previous view.
- **New Horizontal Group** – Split the Source window horizontally to display the selected source file below another file.
- **New Vertical Group** – Split the Source window vertically to display the selected source file side-by-side with another file.
- **Floating** – Display the current source file in a new window, outside the **Source** pane.

View Code Block

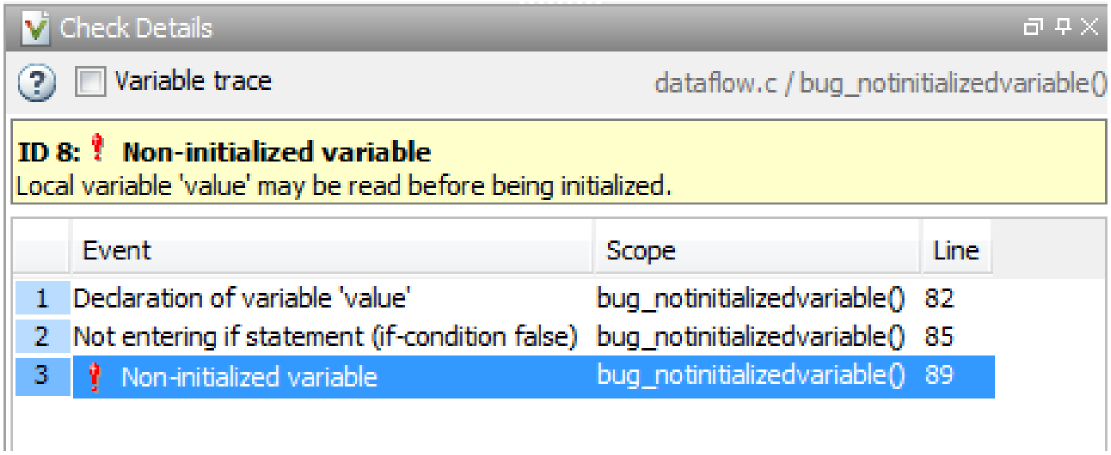
On the **Source** pane, to highlight a block of code, click either its opening or closing brace. If the brace itself is highlighted, click the brace twice.



```
 *=========================================================================
int do_anything(void);

int bug_deactivatedcode(int val)
{
    int retval;
    retval = do_anything();

    if (0 /*|| retval < 0*/) {      /* Defect: Deactivated (debug) code
        retval = do_anything();
    }

    return retval;
}

```

Check Details

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



ID 8: Non-initialized variable
Local variable 'value' may be read before being initialized.

	Event	Scope	Line
1	Declaration of variable 'value'	bug_notinitializedvariable()	82
2	Not entering if statement (if-condition false)	bug_notinitializedvariable()	85
3	Non-initialized variable	bug_notinitializedvariable()	89

- The top right corner shows the file and function containing the defect, in the format *`file_name/function_name`*.
- The yellow box contains the name of the defect with an explanation of why the defect occurs.
- The **Event** column lists the sequence of code instructions causing the defect. The **Scope** column lists the name of the function containing the instructions. The **Line** column lists the line number of the instructions.
- The **Variable trace** check box allows you to see an additional set of instructions that are related to the defect.
- The ? button allows you to access documentation for the defect.

For more information, see “View Code Sequence Causing Defect”.

Bug Finder Defect Categories

In this section...

Numerical

These defects are errors relating to variables in your code; their values, data types, and usage. The defects include:

- Mathematical operations
- Conversion overflow
- Operational overflow

For specific defects, see "Numerical Defects".

Static Memory

These defects are errors relating to memory usage when the memory is statically allocated. The defects include:

- Accessing arrays outside their bounds
- Null pointers
- Casting of pointers

For specific defects, see "Static Memory Defects".

Dynamic Memory

These defects are errors relating to memory usage when the memory is dynamically allocated. The defects include:

- Freeing dynamically allocated memory
- Unprotected memory allocations

For specific defects, see “Dynamic Memory Defects”.

Programming

These defects are errors relating to programming syntax. These defects include:

- Assignment versus equality operators
- Mismatches between variable qualifiers or declarations
- Badly formatted strings

For specific defects, see “Programming Defects”

Data-flow

These defects are errors relating to how information moves throughout your code. The defects include:

- Dead or unreachable code
- Unused code
- Non-initialized information

For the specific defects, see “Data-flow Defects”.

Concurrency

These defects are related to multitasking code. To find these defects, you must specify the multitasking options before analysis. To specify these options, on the **Configuration** pane, select **Multitasking**.

Data Race Defects

The data race defects occur when multiple tasks operate on a shared variable without protection. For the defect to occur:

- One of the operations must be a write operation.
- The operations must not be protected by the same mechanism.

For the specific defects, see “Concurrency Defects”.

Locking Defects

The locking defects occur when the critical sections are not set up appropriately. For example:

- The critical sections are involved in a deadlock.
- A lock function does not have the corresponding unlock function.
- A lock function is called twice without an intermediate call to an unlock function.

Critical sections protect shared variables from concurrent access. Polyspace expects critical sections to follow a certain format. The critical section must lie between a call to a lock function and a call to an unlock function.

For the specific defects, see “Concurrency Defects”.

Other

These defects are those that do not fit into the other categories. They can be thing from race conditions to pass-by-value errors.

For specific defects, see “Other Defects”.

Common Weakness Enumeration from Bug Finder Defects

In this section...
"Common Weakness Enumeration" on page 5-49
"Polyspace Bug Finder and CWE Compatibility" on page 5-49

Common Weakness Enumeration

Common Weakness Enumeration (CWE™) is a dictionary of common software weaknesses that can occur in software architecture, design, code, or implementation. These weaknesses can lead to security vulnerabilities.

The dictionary assigns a unique identifier to each software weakness. Therefore, this dictionary serves as a common language for describing software security weaknesses, and a standard for software security tools targeting these weaknesses.

For more information, see Common Weakness Enumeration.

Polyspace Bug Finder and CWE Compatibility

With Polyspace Bug Finder, you can check and document whether your software contains weaknesses listed in the CWE dictionary. Polyspace Bug Finder supports some aspects of the CWE Compatibility and Effectiveness Program:

CWE Compatibility Requirement	Polyspace Bug Finder Support
CWE Searchable	You can list instances of a software weakness corresponding to a certain CWE identifier. For more information, see "Filter CWE Identifiers".
CWE Output	• You can view CWE identifiers corresponding to certain Polyspace Bug Finder defects. For more information, see "View CWE Identifiers".

CWE Compatibility Requirement	Polyspace Bug Finder Support
	• You can include CWE identifiers corresponding to Polyspace Bug Finder defects in your report. For more information, see “Generate Report with CWE Identifiers”.

For more information on the CWE Compatibility and Effectiveness Program, see CWE Compatibility.

Related Examples

- “Find CWE Identifiers from Defects”

More About

- “Mapping Between CWE Identifiers and Defects”

Find CWE Identifiers from Defects

This example shows how to check whether your software has weaknesses listed by the Common Weakness Enumeration or CWE dictionary. The dictionary assigns a unique identifier to each software weakness. When a Polyspace Bug Finder result can be associated with CWE identifiers, the software displays those identifiers for the result. Using the identifiers, you can evaluate your code against CWE standards.

In this section...

View CWE Identifiers

To view the CWE identifiers for defects on the **Results Summary** pane:

1. Right-click any column header.
2. Select **CWE ID**.

Filter CWE Identifiers

To filter a particular CWE identifier:

1. On the **CWE ID** column, click the icon.
2. From the drop-down list, select **Custom**.
3. From the **Condition** drop-down list, select `contains`.
4. In the **Value** field, enter the CWE ID that you want to filter. Click **OK**.

Generate Report with CWE Identifiers

To generate a report containing CWE identifiers, do the following.

- To enable report generation before analysis:

  1. On the **Configuration** pane, select **Reporting**.

2. Select **Generate report**.
3. From the **Report template** list, select `BugFinder_CWE`.

- To generate a report after analysis:

  1. Open your results.
  2. Select **Reporting** > **Run Report**.
  3. From the **Select Reports** list, select `BugFinder_CWE`.

More About

- "Common Weakness Enumeration from Bug Finder Defects"
- "Mapping Between CWE Identifiers and Defects"

Mapping Between CWE Identifiers and Defects

The following table lists the CWE IDs addressed by Polyspace Bug Finder and the corresponding defects.

CWE ID	Polyspace Bug Finder Defect
119	Array access out of bounds Pointer access out of bounds
120	Invalid use of standard library memory routine Invalid use of standard library string routine
134	Format string specifiers and arguments mismatch
170	Missing null in string array
188	Array access out of bounds Pointer access out of bounds Unreliable cast of pointer
190	Integer conversion overflow Integer overflow Shift operation overflow Unsigned integer overflow
191	Integer conversion overflow Integer overflow Unsigned integer overflow
194	Sign change integer conversion overflow
195	Integer conversion overflow Sign change integer conversion overflow
196	Integer conversion overflow Sign change integer conversion overflow

CWE ID	Polyspace Bug Finder Defect
197	Integer conversion overflow
244	Memory leak
252	Missing or invalid return statement
253	Missing or invalid return statement
366	Data race including atomic operations Data race
369	Float division by zero Integer division by zero
393	Missing or invalid return statement
394	Missing or invalid return statement
398	Write without further read
401	Memory leak
404	Invalid deletion of pointer Invalid free of pointer Memory leak
415	Deallocation of previously deallocated pointer
416	Use of previously freed pointer
456	Non-initialized pointer Non-initialized variable
457	Non-initialized pointer Non-initialized variable
466	Array access out of bounds Pointer access out of bounds
467	Wrong type used in sizeof

CWE ID	Polyspace Bug Finder Defect
468	Array access out of bounds Pointer access out of bounds Unreliable cast of pointer
476	Null pointer
481	Invalid use of = (assignment) operator
482	Invalid use of == (equality) operator
489	Code deactivated by constant false condition
561	Dead code Uncalled function Unreachable code Useless if
563	Write without further read
588	Array access out of bounds Pointer access out of bounds
590	Invalid free of pointer
617	Assertion
628	Declaration mismatch
667	Missing unlock
681	Float conversion overflow
685	Declaration mismatch
686	Declaration mismatch
761	Invalid free of pointer
762	Invalid free of pointer
764	Double lock
765	Double unlock
789	Unprotected dynamic memory allocation

CWE ID	Polyspace Bug Finder Defect
823	Array access out of bounds Pointer access out of bounds
824	Non-initialized pointer
832	Missing lock
833	Deadlock
873	Invalid use of floating point operation Float overflow
908	Non-initialized pointer Non-initialized variable

Command-Line Analysis

Create Project Automatically at Command Line

If you use build automation scripts to build your source code, you can automatically setup a Polyspace project from your scripts. The automatic project setup runs your automation scripts to determine:

- Source files.
- Includes.
- Target & compiler options. For more information on these options, see:
  - C Code: "Target & Compiler"
  - C++ Code: "Target & Compiler"

Use the `polyspace-configure` command to trace your build automation scripts. You can use the trace information to:

- Create a Polyspace project. You can then open the project in the user interface.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following command to create a Polyspace project `myProject.psprj` from your makefile:

  ```
  polyspace-configure -prog myProject make targetName buildOptions
  ```

  For the list of options allowed with the GNU `make`, see make options.
- Create an options file. You can then use the options file to run verification on your source code from the command-line.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following commands to create an options file `myOptions` from your makefile:

  ```
  polyspace-configure -no-project -output-options-file myOptions ...
            make targetName buildOptions
  ```
  Use the options file to run verification:

  ```
  polyspace-bug-finder-nodesktop -options-file myOptions
  ```

You can also use advanced options to modify the default behavior of `polyspace-configure`. For more information, see the `-options value` argument for `polyspaceConfigure`.

Run Local Analysis from Command Line

To run an analysis from a DOS or UNIX command window, use the command `polyspace-bug-finder-nodesktop` followed by other options you wish to use.

Note: To run Bug Finder from the MATLAB Command Window, use the command `polyspaceBugFinder` *`[options]`*

In this section...

Specify Sources and Analysis Options Directly

At the Windows, Linux or Mac OS X command-line, append sources and analysis options to the `polyspace-bug-finder-nodesktop` command.

For instance:

- To specify the target processor, use the `-target` option. For instance, to specify the `m68k` processor for your source file `file.c`, use the command:

  ```
  polyspace-bug-finder-nodesktop -sources "file.c" -lang c -target m68k
  ```
- To check for violation of MISRA C rules, use the `-misra2` option. For instance, to check for only the required MISRA C rules on your source file `file.c`, use the command:

  ```
  polyspace-bug-finder-nodesktop -sources "file.c" -misra2 required-rules
  ```

For the full list of analysis options, see "Analysis Options for C" or "Analysis Options for C++".

You can also enter the following at the command line:

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

Specify Sources and Analysis Options in Text File

1 Create an options file called `listofoptions.txt` with your options. For example:

```
#These are the options for MyBugFinderProject
-lang c
-prog MyBugFinderProject
-author jsmith
-sources "mymain.c,funAlgebra.c,funGeometry.c"
-OS-target no-predefined-OS
-target x86_64
-dialect none
-dos
-misra2 required-rules
-includes-to-ignore all-headers
-checkers default
-disable-checkers concurrency
-results-dir C:\Polyspace\MyBugFinderProject
```

2 Run Polyspace using options in the file `listofoptions.txt`.

```
polyspace-bug-finder-nodesktop -options-file listofoptions.txt
```

Create Options File from Build System

1 Create a list of Polyspace options using the configuration tool.

```
polyspace-configure -c -no-project -output-options-file \
        myOptions make -B myCode
```

2 Run Polyspace Bug Finder using the options read from your build.

```
polyspace-bug-finder-nodesktop -options-file myOptions \
        -results-dir myResults
```

3 Open the results in the Bug Finder interface.

```
polyspace-bug-finder myResults
```

Run Remote Analysis at Command Line

Before you run a remote analysis, you must set up a server for this purpose. For more information, see “Set Up Server for Remote Verification and Analysis”.

In this section...

Run Remote Analysis

Use the following command to run a remote verification:

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

where:

- *`MATLAB_Install`* is your MATLAB installation folder.
- *`NodeHost`* is the name of the computer that hosts the head node of your MDCS cluster.
- *`MJSName`* is the name of the MATLAB Job Scheduler (MJS) on the head node host.
- *`options`* are the analysis options. These options are the same as that of a local analysis. For more information, see “Run Local Analysis from Command Line”.

After compilation, the software submits the verification job to the cluster and provides you a job ID. Use the `polyspace-jobs-manager` command with the job ID to monitor your verification and download results after verification is complete. For more information, see:

- “Manage Remote Analysis” on page 6-6
- “Download Results” on page 6-8

Tip In Windows, to avoid typing the commands each time, you can save the commands in a batch file.

1 Save your analysis options in a file `listofoptions.txt`. See “Specify Sources and Analysis Options in Text File”.
To specify your sources, in the options file, instead of `-sources`, use -sources-list-file. This option is available only for remote analysis and allows you to specify your sources in a separate text file.

2 Create a file `launcher.bat` in a text editor like Notepad.

3 Enter the following commands in the file.

```
echo off
set POLYSPACE_PATH=C:\Program Files\MATLAB\R2015a\polyspace\bin
set RESULTS_PATH=C:\Results
set OPTIONS_FILE=C:\Options\listofoptions.txt
"%POLYSPACE_PATH%\polyspace-bug-finder-nodesktop.exe" -batch -scheduler localhost
                    -results-dir "%RESULTS_PATH%" -options-file "%OPTIONS_FILE%"
pause
```

4 Replace the definitions of the following variables in the file:
- `POLYSPACE_PATH`: Enter the actual location of the `.exe` file.
- `RESULTS_PATH`: Enter the path to a folder. The files generated during compilation are saved in the folder.
- `OPTIONS_FILE`: Enter the path to the file `listofoptions.txt`.

Replace `localhost` with the name of the computer that hosts the head node of your MDCS cluster.

5 Double-click `launcher.bat` to run the verification.

If you run a Polyspace verification, a `.bat` file is automatically generated for you. You can relaunch verification using this file.

Manage Remote Analysis

To manage remote analyses, use this command:

```
MATLAB_Install\polyspace\bin\polyspace-jobs-manager action [options]
          [-scheduler schedulerOption]
```

where:

- *`MATLAB_Install`* is your MATLAB installation folder
- `schedulerOption` is one of the following:

- Name of the computer that hosts the head node of your MDCS cluster (*`NodeHost`*).
- Name of the MJS on the head node host (*`MJSName@NodeHost`*).
- Name of a MATLAB cluster profile (*`ClusterProfile`*).

  For more information about clusters, see “Clusters and Cluster Profiles”

  If you do not specify a job scheduler, `polyspace-job-manager` uses the scheduler specified in the **Polyspace Preferences** > **Server Configuration** > **Job scheduler host name**.

- *`action [options]`* refer to the possible action commands to manage jobs on the scheduler:

Action	Options	Task
`listjobs`	None	Generate a list of Polyspace jobs on the scheduler. For each job, the software produces the following information: • `ID` — Verification or analysis identifier. • `AUTHOR` — Name of user that submitted job. • `APPLICATION` — Name of Polyspace product, for example, Polyspace Code Prover or Polyspace Bug Finder. • `LOCAL_RESULTS_DIR` — Results folder on local computer, specified through the **Tools** > **Preferences** > **Server Configuration** tab. • `WORKER` — Local computer from which job was submitted. • `STATUS` — Status of job, for example, `running` and `completed`. • `DATE` — Date on which job was submitted. • `LANG` — Language of submitted source code.
`download`		

Action	Options	Task
	`-job` *`ID`* `-results-folder` *`FolderPath`*	Download results of analysis with specified ID to folder specified by *`FolderPath`*.
`getlog`	`-job` *`ID`*	Open log for job with specified ID.
`remove`	`-job` *`ID`*	Remove job with specified ID.

Download Results

To download verification results from the command line, use the `polyspace-jobs-manager` command:

```
MATLAB_Install\polyspace\bin\polyspace-jobs-manager -download
-job Verification_ID -results-folder FolderPath
```

After downloading results, use the Polyspace user interface to view the results. See “Open Results”.

Create Project Automatically from MATLAB Command Line

If you use build automation scripts to build your source code, you can automatically setup a Polyspace project from your scripts. The automatic project setup runs your automation scripts to determine:

- Source files.
- Includes.
- Target & compiler options. For more information on these options, see:
  - C Code: “Target & Compiler”
  - C++ Code: “Target & Compiler”

Use the `polyspaceConfigure` command to trace your build automation scripts. You can use the trace information to:

- Create a Polyspace project. You can then open the project in the user interface.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following command to create a Polyspace project `myProject.psprj` from your makefile:

  ```
  polyspaceConfigure  -prog myProject ...
               make targetName buildOptions
  ```
- Create an options file. You can then use the options file to run verification on your source code from the command-line.

  Example: If you use the command `make targetName buildOptions` to build your source code, use the following commands to create an options file `myOptions` from your makefile:

  ```
  polyspaceConfigure  -no-project -output-options-file myOptions ...
              make targetName buildOptions
  ```
  Use the options file to run verification:

  ```
  polyspaceBugFinder -options-file myOptions
  ```

You can also use advanced options to modify the default behavior of `polyspaceConfigure`. For more information, see `polyspaceConfigure`.

Polyspace Bug Finder Analysis in Simulink

Embedded Coder Considerations

In this section...

Default Options

For Embedded Coder® code, the software sets certain analysis options by default.

Default options for C:

```
-sources path_to_source_code
-results-dir results
-D PST_ERRNO
-D main=main_rtwec __restrict__=
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-OS-target no-predfined-OS
-ignore-constant-overflows true
-scalar-overflows-behavior wrap-around
-allow-negative-operand-in-shift true
-boolean-types boolean_T
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
```

Default options for C++:

```
-sources path_to_source_code
-results-dir results
-D PST_ERRNO
-D main=main_rtwec __restrict__=
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
```

```
-OS-target no-predfined-OS
-dialect iso
-ignore-constant-overflows true
-scalar-overflows-behavior wrap-around
-allow-negative-operand-in-shift true
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
```

Note: *matlabroot* is the MATLAB installation folder.

Recommended Polyspace Bug Finder Options for Analyzing Generated Code

For Embedded Coder code, you can specify other analysis options for your Polyspace Project through the Polyspace **Configuration** pane. To open this pane:

1. In the Simulink® model window, select **Code** > **Polyspace** > **Options**. The **Polyspace** pane opens.
2. Click **Configure**. The Polyspace **Configuration** pane opens.

The following table describes options that you should specify in your Polyspace project before analyzing code generated by Embedded Coder software.

Option	Recommended Value	Comments
Macros > **Preprocessor definitions** `-D`	See comments	Defines macro compiler flags used during compilation. Some defines are applied by default, depending on your `-OS-target`. Use one `-D` for each line of the Embedded Coder generated `defines.txt` file. Polyspace does not do this by default.
Target & Compiler > **Target operating system** `-OS-target`	`Visual`	Specifies the operating system target for Polyspace stubs. This information allows the analysis to use system definitions during preprocessing to analyze the included files.

Option	Recommended Value	Comments
Environment Settings > Code from DOS or Windows file system `-dos`	On	You must select this option if the contents of the include or source directory comes from a DOS or Windows file system. The option allows the analysis to deal with upper/lower case sensitivity and control characters issues. Concerned files are: • **Header files** – All include folders specified (`-I` option) • **Source files** – All source files selected for the analysis (`-sources` option)

Hardware Mapping Between Simulink and Polyspace

The software automatically imports target word lengths and byte ordering (endianess) from Simulink model hardware configuration settings. The software maps **Device vendor** and **Device type** settings on the Simulink **Configuration Parameters > Hardware Implementation** pane to **Target processor type** settings on the Polyspace **Configuration** pane.

The software creates a generic target for the analysis.

TargetLink Considerations

In this section...

TargetLink Support

For Windows, Polyspace Bug Finder is tested with releases 3.4 and 3.5 of the dSPACE® Data Dictionary version and TargetLink® Code Generator.

As Polyspace Bug Finder extracts information from the dSPACE Data Dictionary, you must regenerate the code before performing an analysis.

Default Options

The following default options are set by Polyspace:

```
-sources path_to_source_code
-results-dir results
-I path to source code
-D PST_ERRNO
-I dspaceroot\matlab\TL\SimFiles\Generic
-I dspaceroot\matlab\TL\srcfiles\Generic
-I dspaceroot\matlab\TL\srcfiles\i86\LCC
-I matlabroot\polyspace\include
-I matlabroot\extern\include
-I matlabroot\rtw\c\libsrc
-I matlabroot\simulink\include
-I matlabroot\sys\lcc\include
-functions-to-stub=[rtIsNaN,rtIsInf,rtIsNaNF,rtIsInfF]
-OS-target no-predfined-OS
-ignore-constant-overflows
-scalar-overflows-behavior wrap-around
-boolean-types Bool
```

Note: *`dspaceroot`* and *`matlabroot`* are the dSPACE and MATLAB tool installation directories respectively.

Lookup Tables

The tool by default provides stubs for the lookup table functions. This behavior can be disabled from the Polyspace menu. The dSPACE data dictionary is used to define the range of their return values. Note that a lookup table that uses extrapolation will return full range for the type of variable that it returns.

Code Generation Options

From the TargetLink Main Dialog, it is recommended to set the option `Clean code` and deselect the option `Enable sections/pragmas/inline/ISR/user attributes`.

When installing Polyspace, the `tlcgOptions` variable has been updated with `'PolyspaceSupport', 'on'` (see variable in `'C:\dSPACE\Matlab\Tl\config\codegen\tl_pre_codegen_hook.m'` file).

Related Examples

- "Run Analysis for TargetLink" on page 10-6

External Web Sites

- dSPACE – TargetLink

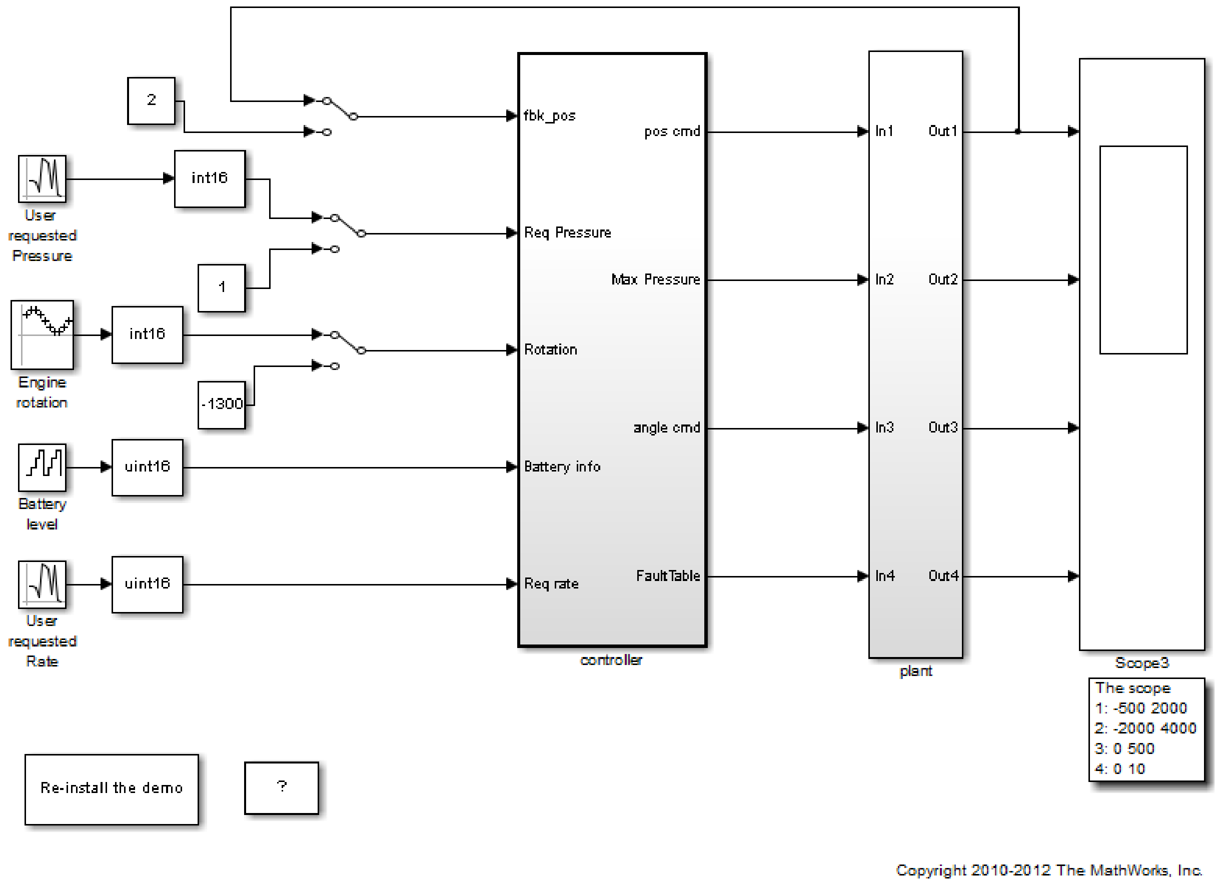
Generate and Analyze Code

This example shows you how to use Polyspace Bug Finder to analyze submodels and S-Functions.

Open the Example

1 Open the example model.

```
psdemo_model_link_sl
```



Generate and Analyze Code

1. Double-click the **Re-install Demo** block to install the S-Function `Command_Strategy.c`.
2. Right-click the `controller` subsystem.
3. From the context-menu, select **C/C++ Code** > **Build This Subsystem**.
4. In the dialog box that pops up, select **Build**.
5. After the build is completed, right-click the `controller` subsytem.
6. From the context menu, select **Polyspace** > **Verify Code Generated for** > **Selected Subsystem** to start the analysis. You can monitor progress from the Command Window.
7. Once the analysis is complete, right-click the `controller` subsystem.
8. From the context menu, select **Polyspace** > **Open Results**. The results open in the Polyspace environment.

Main Generation for Model Analysis

When you run an analysis, the software automatically reads the following information from the model:

- `initialize()` functions
- `terminate()` functions
- `step()` functions
- List of parameter variables
- List of input variables

The software then uses this information to generate a `main` function that:

1. Initializes parameters using the Polyspace option `-variables-written-before-loop`.
2. Calls initialization functions using the option `-functions-called-before-loop`.
3. Initializes inputs using the option `-variables-written-in-loop`.
4. Calls the `step` function using the option `-functions-called-in-loop`.
5. Calls the `terminate` function using the option `-functions-called-after-loop`.

If the `codeInfo` for the model does not contain the names of the inputs, the software considers all variables as entries, except for parameters and outputs.

For C++ code that is generated with Embedded Coder, the `initialize()`, `step()`, and `terminate()` functions are either class methods or have global scope. These different scopes contain the associated variables.

- For class methods in the generated code, the variables that are written before and in the loop refer to the class members.
- For functions with global scope, the associated variables are also in the global scope.

`main` for Generated Code

The following example shows the `main` generator options that the software uses to generate the `main` function for code generated from a Simulink model.

```
init parameters    \\ -variables-written-before-loop
init_fct()          \\ -functions-called-before-loop
  while(1){        \\ start main loop
  init inputs      \\ -variables-written-in-loop
  step_fct()       \\ -functions-called-in-loop
}
terminate_fct()    \\ -functions-called-after-loop
```

Review Generated Code Results

After you run a Polyspace analysis on generated code, you review the results from the Polyspace environment. From the results you can link back to the related blocks in your model.

1 Open the results using one of the following methods.

   - If you analyzed the whole model, from the Simulink toolbar, select **Code** > **Polyspace** > **Open Results**.

     If you set **Model reference verification depth** to `All` and selected **Model by model verification**. The **Select the Result Folder to Open in Polyspace** dialog box opens showing a hierarchy of referenced models from which the software generates code. To view the analysis results for a specific model, select the model from the hierarchy. Then click **OK**.
   - If you want to open results for a Model block or subsystem, right-click the Model block or subsystem, and from the context menu, select **Polyspace** > **Open Results**.
   - From the Polyspace Interface, select **File** > **Open** and navigate to your results.
   - If you selected **Add to results repository** the results are stored on the Polyspace Metrics server. See "Download Results From Polyspace Metrics" on page 5-5.

2 On the **Results Summary** tab, select a result.

   When you select a result, the **Check Details** pane shows additional information about the defect, including traceback information (if available).

3 Look at the result in the **Source** pane. Your select result is highlighted in the source code.

4 Hover over the result in the source code. The tooltip can provide additional information including variable ranges.

5 Above the defect, click a blue underlined link. For example, `<Root>/Relational Operator`.

   The Simulink model opens, highlighting the block related to the nearby source code. This back-to-model linking allows you to fix defects in the model instead of the generated code.

Related Examples

- “View Results”
- “Polyspace Bug Finder Results”

More About

- “Troubleshoot Back to Model” on page 7-13

Troubleshoot Back to Model

In this section...

Back-to-Model Links Do Not Work

You may encounter issues with the back-to-model feature if:

- Your operating system is Windows Vista™ or Windows 7; and User Account Control (UAC) is enabled or you do not have administrator privileges.
- You have multiple versions of MATLAB installed.

To reconnect MATLAB and Polyspace:

1. Close Polyspace.
2. At the MATLAB command-line, enter `PolySpaceEnableCOMserver`.

   When you open your Polyspace results, the hyper-links will highlight the relevant blocks in your model.

Your Model Already Uses Highlighting

If your model extensively uses block coloring, the coloring from this feature may interfere with the colors already in your model. To change the color of blocks when they are linked to Polyspace results use this command:

```
HILITE_DATA = struct('HiliteType', 'find', 'ForegroundColor', 'black', ...
        'BackgroundColor', color);
set_param(0, 'HiliteAncestorsData', HILITE_DATA)
```

Where *`color`* is one of the following:

- `'cyan'`
- `'magenta'`
- `'orange'`
- `'lightBlue'`

- `'red'`
- `'green'`
- `'blue'`
- `'darkGreen'`

Configure Model for Code Analysis

Configure Simulink Model

Before analyzing your generated code, there are certain settings that you should apply to your model. Use the following workflow to prepare your model for code analysis.

- If you know of results ahead of time, annotate your blocks with Polyspace annotations.
- Set the recommended configuration parameters.
- Double-check your model settings.
- Generate code.
- Set up your Polyspace options.

Recommended Model Settings for Code Analysis

For Polyspace analyses, set the following parameter configurations before generating code.

Grouping	Parameter	Recommended value	Name and Location in Configuration	If you do not use recommendation
	`SystemTargetF`	An Embedded Coder Target Language Compiler (TLC) file. For example `ert.tlc` or `autosar.tlc`	Location: **Code Generation** Name: **System target file** Value: Embedded Coder target file	Error
	`MatFileLoggin`	`'off'`	Location: **Code Generation** > **Interface** Name: **MAT-file logging** Value: ☐ Not selected	Warning
Code Generation	`GenerateRepor`	`'on'`	Location: **Code Generation** > **Report** Name: **Create code-generation report** Value: ☑ Selected	Error
	`IncludeHyperl`	`'on'`	Location: **Code Generation** > **Report** Name: **Code-to-model** Value: ☑ Selected	Error
	`GenerateSampl`	`'off'`	Location: **Code Generation** > **Templates** Name: **Generate an example main program**	Warning

Grouping	Parameter	Recommended value	Name and Location in Configuration	If you do not use recommendation
			Value: ☐ Not selected	
	`GenerateComme`	`'on'`	Location: **Code Generation > Comments** Name: **Include comments** Value: ☑ Selected	Warning
Optimization	`InlineParams`	`'on'`	Location: **Optimization > Signals and Parameters** Name: **Inline parameters** Value: ☑ Selected	Warning
	`InitFltsAndDb`	`'on'`	Location: **Optimization** Name: **Use memset to initialize floats and doubles to 0.0** Value: ☐ Not selected	Warning
	`ZeroExternalM`	`'on'` when **Configuratio Parameters > Polyspace > Data Range Management > Output** is `Global assert`	Location: **Optimization** Name: **Remove root level I/O zero initialization** Value: ☐ Not selected	Warning
Solver	`SolverType`	`'Fixed-Step'`	Location: **Solver** Name: **Type** Value: `Fixed-step`	Warning

Grouping	Parameter	Recommended value	Name and Location in Configuration	If you do not use recommendation
	`Solver`	`'FixedStepD`	Location: **Solver** Name: **Solver** Value: `discrete (no continuous states)`	Warning

Check Simulink Model Settings

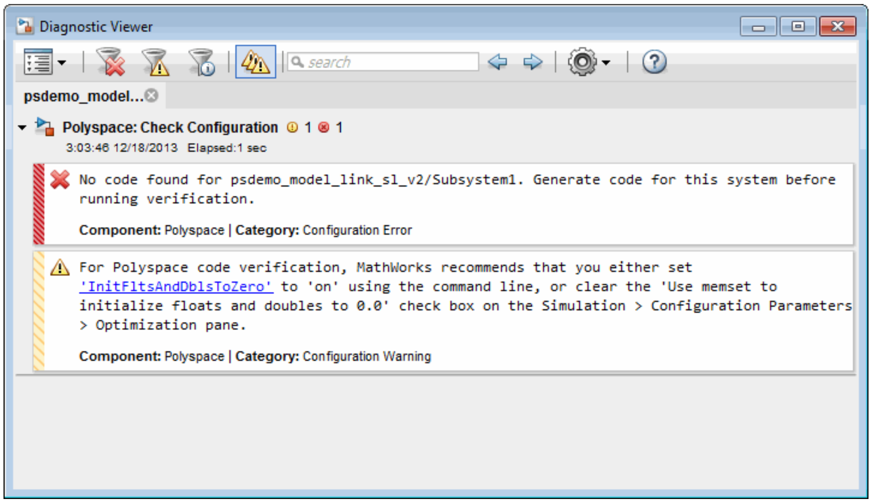
With the Polyspace plug-in, you can check your model settings before generating code or before starting an analysis. If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, warnings appear when you run the analysis.

Check Simulink Model Settings Using the Code Generation Advisor

Before generating code, you can check your model settings against the “Recommended Model Settings for Code Analysis” on page 8-3. If you do not use the recommended model settings, the back-to-model linking will not work correctly.

1. From the Simulink model window, select **Code** > **C/C++ Code** > **Code Generation Options**. The Configuration Parameters dialog box opens, displaying the **Code Generation** pane.
2. Select **Set Objectives**.
3. From the **Set Objective – Code Generation Advisor** window, add the `Polyspace` objective and any others that you want to check.
4. In the **Check model before generating code** drop-down list, select either:
   - `On (stop for warnings)`, the process stops for either errors or warnings without generating code.
   - `On (proceed with warnings)`, the process stops for errors, but continues generating code if the configuration only has warnings.
5. Select **Check Model**.

   The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.


Diagnostic Viewer
search
psdemo_model...
Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

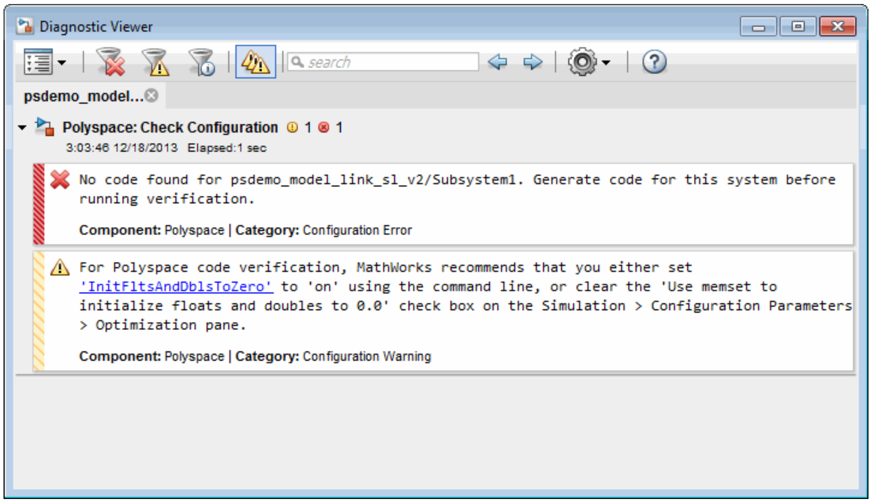
Component: Polyspace | **Category:** Configuration Warning

Check Simulink Model Settings Before Analysis

With the Polyspace plug-in, you can check your model settings before starting an analysis:

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.

2 Click **Check configuration**. If your model settings are not optimal for Polyspace, the software displays warning messages with recommendations.



Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

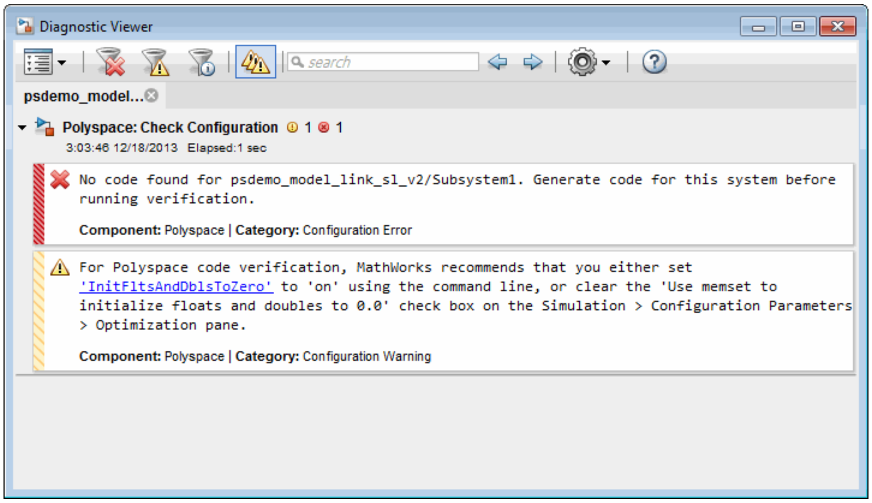
Component: Polyspace | **Category:** Configuration Warning

3 From the **Check configuration before verification** menu, select either:

- `On (stop for warnings)`, the analysis stops for either errors or warnings.
- `On (proceed with warnings)`, the analysis stops for errors, but continues the code analysis if the configuration only has warnings.

4 Select **Run verification**.

The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.



Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

Component: Polyspace | **Category:** Configuration Warning

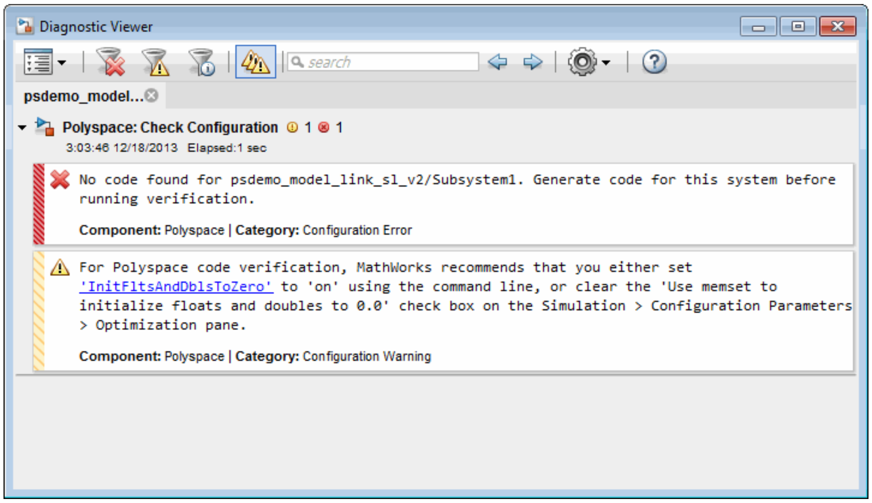
If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, the software produces warnings when you run the analysis.

Check Simulink Model Settings Automatically

With the Polyspace plug-in, you can check your model settings before starting an analysis:

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.

2 Click **Check configuration**. If your model settings are not optimal for Polyspace, the software displays warning messages with recommendations.



Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

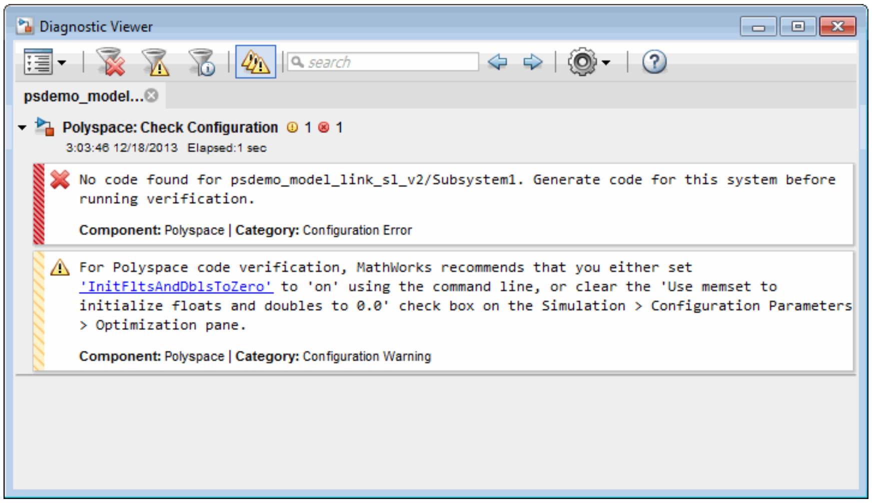
Component: Polyspace | **Category:** Configuration Warning

3 From the **Check configuration before verification** menu, select either:

- `On (stop for warnings)` — will
- `On (proceed with warnings)`

4 Select **Run verification**.

The software runs a configuration check. If your configuration check finds errors or warnings, the **Diagnostics Viewer** displays the issues and recommendations.



Polyspace: Check Configuration 1 1
3:03:46 12/18/2013 Elapsed:1 sec

```
No code found for psdemo_model_link_sl_v2/Subsystem1. Generate code for this system before
running verification.
```

Component: Polyspace | **Category:** Configuration Error

```
For Polyspace code verification, MathWorks recommends that you either set
'InitFltsAndDblsToZero' to 'on' using the command line, or clear the 'Use memset to
initialize floats and doubles to 0.0' check box on the Simulation > Configuration Parameters
> Optimization pane.
```

Component: Polyspace | **Category:** Configuration Warning

If you select:

- `On (stop for warnings)`, the analysis stops for either errors or warnings.
- `On (proceed with warnings)` — the analysis stops for errors, but continues the code analysis if the configuration only has warnings.

If you alter your model settings, rebuild the model to generate fresh code. If the generated code version does not match your model version, the software produces warnings when you run the analysis.

More About

- "Recommended Model Settings for Code Analysis" on page 8-3

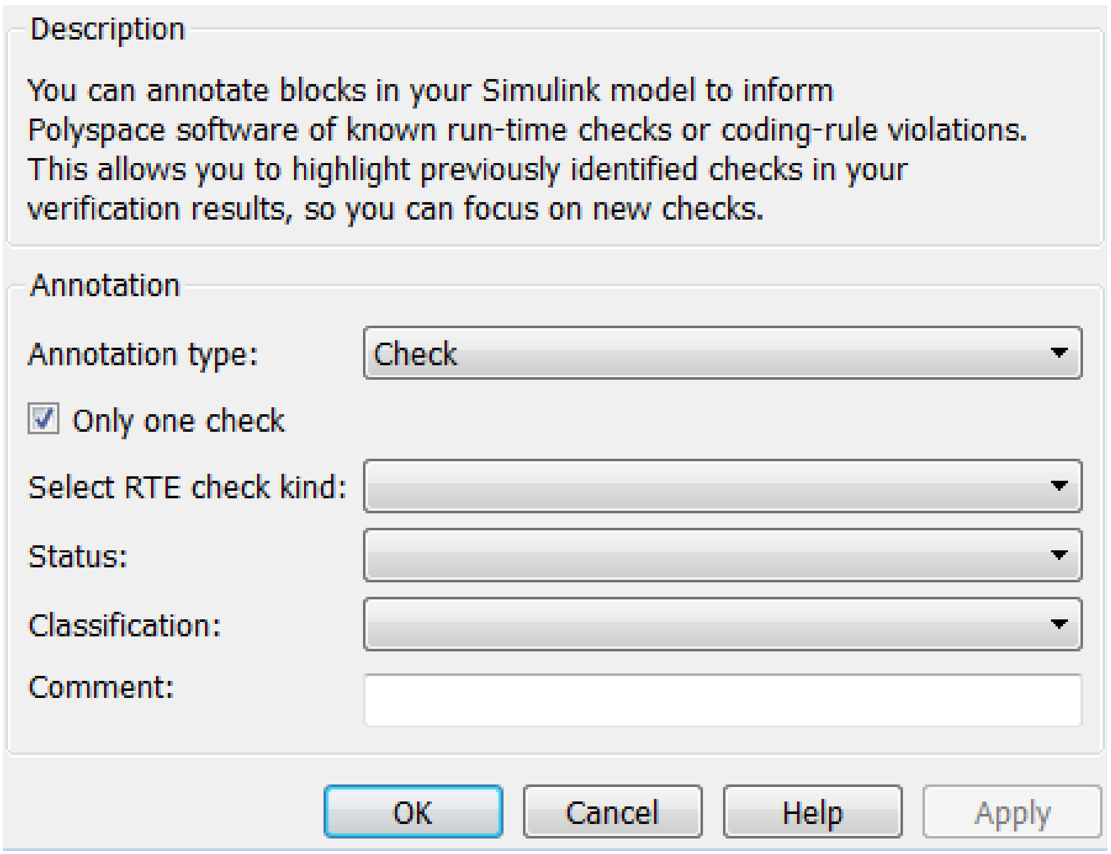
Annotate Blocks for Known Results

You can annotate individual blocks in your Simulink model to inform Polyspace software of known defects, run-time checks, or coding-rule violations. This allows you to highlight and categorize previously identified results, so you can focus on reviewing new results.

Your Polyspace results displays the information that you provide with block annotations. To annotate blocks:

1 In the Simulink model window, right-click the block you want to annotate.

2 From the context menu, select **Polyspace** > **Annotate Selected Block** > **Edit**. The Polyspace Annotation dialog box opens.



Description

You can annotate blocks in your Simulink model to inform Polyspace software of known run-time checks or coding-rule violations. This allows you to highlight previously identified checks in your verification results, so you can focus on new checks.

Annotation

Annotation type: Check

[✓] Only one check

Select RTE check kind:

Status:

Classification:

Comment:

3. From the **Annotation type** drop-down list, select one of the following:

   - `Check` — To indicate a Code Prover run-time error
   - `Defect` — To indicate a Bug Finder defect
   - `MISRA-C` — To indicate a MISRA C coding rule violation
   - `MISRA-C++` — To indicate a MISRA C++ coding rule violation
   - `JSF` — To indicate a JSF C++ coding rule violation

4. If you want to highlight only one kind of result, select **Only 1 check** and the relevant error or coding rule from the **Select RTE check kind** (or **Select defect kind**, **Select MISRA rule**, **Select MISRA C++ rule**, or **Select JSF rule**) drop-down list.

   If you want to highlight a list of checks, clear **Only 1 check**. In the **Enter a list of checks** (or **Enter a list of defects**, or **Enter a list of rule numbers**) field, specify the errors or rules that you want to highlight.
5. Select a **Status** to describe how you intend to address the issue:

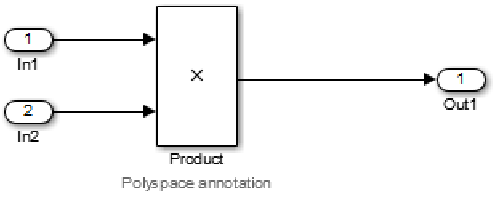
   - `Fix`
   - `Improve`
   - `Investigate`
   - `Justified`

     (This status also marks the result as justified.)
   - `No action planned`

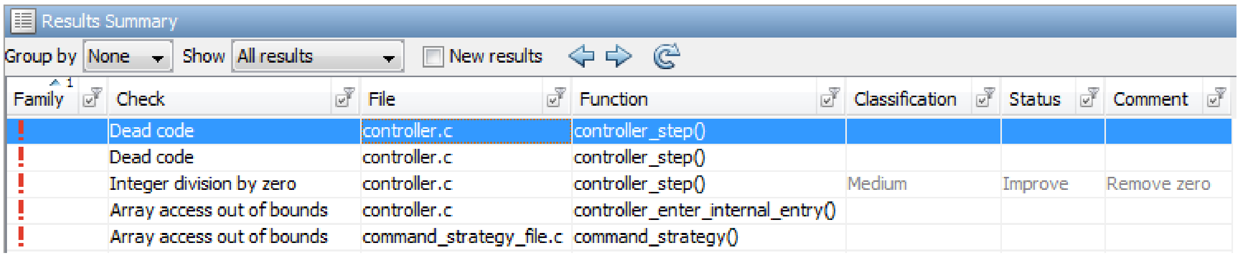
     (This status also marks the result as justified.)
   - `Other`
6. Select a **Classification** to describe the severity of the issue:

   - `High`
   - `Medium`
   - `Low`
   - `Not a defect`
7. In the **Comment** field, enter additional information about the check.

8 Click **OK**. The software adds the Polyspace annotation is to the block.



When you run an analysis, the **Results Summary** pane pre-populates the results with your annotation.



Family	Check	File	Function	Classification	Status	Comment
!	Dead code	controller.c	controller_step()			
!	Dead code	controller.c	controller_step()			
!	Integer division by zero	controller.c	controller_step()	Medium	Improve	Remove zero
!	Array access out of bounds	controller.c	controller_enter_internal_entry()			
!	Array access out of bounds	command_strategy_file.c	command_strategy()			

See Also

`pslinkfun`

Configure Code Analysis Options

Polyspace Configuration for Generated Code

You do not have to manually create a Polyspace project or specify Polyspace options before running an analysis for your generated code. By default, Polyspace automatically creates a project and extracts the required information from your model. However, you can modify or specify additional options for your analysis:

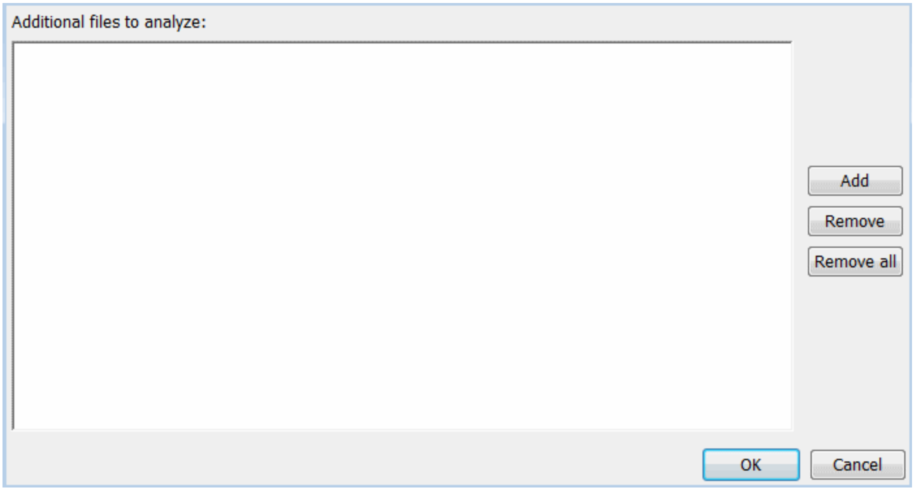
- You may incorporate separately created code within the code generated from your Simulink model. See “Include Handwritten Code” on page 9-3.
- You may customize the options for your analysis. For example, to specify the target environment or adjust precision settings. See “Configure Polyspace Analysis Options and Properties” on page 9-7 and “Recommended Polyspace Bug Finder Options for Analyzing Generated Code”.
- You may create specific configurations for batch runs. See “Save a Polyspace Configuration File Template” on page 9-8.
- If you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. See “Set Custom Target Settings” on page 9-11.

Include Handwritten Code

Files such as S-function wrappers are, by default, not part of the Polyspace analysis. However, you can add these files manually.

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.

2 Select the **Enable additional file list** check box. Then click **Select files**. The Files Selector dialog box opens.



3 Click **Add**. The Select files to add dialog box opens.

4 Use the Select files to add dialog box to:

- Navigate to the relevant folder
- Add the required files.

The software displays the selected files as a list under **Additional files to analyze**.

Note: To remove a file from the list, select the file and click **Remove**. To remove all files from the list, click **Remove all**.

5 Click **OK**.

Configure Analysis Depth for Referenced Models

From the **Polyspace** pane, you can specify the analysis of generated code with respect to model reference hierarchy levels:

- **Model reference verification depth** — From the drop-down list, select one of the following:
  - `Current model only` — Default. The Polyspace runs code from the top level only. The software creates stubs to represent code from lower hierarchy levels.
  - `1` — The software analyzes code from the top level and the next level. For subsequent hierarchy levels, the software creates stubs.
  - `2` — The software analyzes code from the top level and the next two hierarchy levels. For subsequent hierarchy levels, the software creates stubs.
  - `3` — The software analyzes code from the top level and the next three hierarchy levels. For subsequent hierarchy levels, the software creates stubs.
  - `All` — The software analyzes code from the top level and all lower hierarchy levels.
- **Model by model verification** — Select this check box if you want the software to analyze code from each model separately.

Note: The same configuration settings apply to all referenced models within a top model. It does not matter whether you open the **Polyspace** pane from the top model window (**Code** > **Polyspace** > **Options**) or through the right-click context menu of a particular Model block within the top model. However, you can run analyses for code generated from specific Model blocks. See "Run Analysis for Embedded Coder" on page 10-5.

Check Coding Rules Compliance

You can check compliance with MISRA AC AGC and MISRA C:2004, and MISRA C:2012 coding rules directly from your Simulink model.

In addition, you can choose to run coding rules checking either with or without full code analysis.

To configure coding rules checking:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The **Polyspace** pane opens.
2. In the **Settings from** drop-down menu, select the type of analysis you want to perform.

   Depending on the type of code generated, different settings are available. The following tables describe the different settings.

   C Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA AC AGC checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA AC-AGC rule set.
`Project configuration and MISRA C 2004 checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with MISRA C:2004 coding rules.
`Project configuration and MISRA C 2012 ACG checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with MISRA C:2012 coding guidelines.
`MISRA AC AGC checking`	Check compliance with the MISRA AC-AGC rule set. Polyspace stops after rules checking.

Setting	Description
`MISRA C 2004 checking`	Check compliance with MISRA C:2004 coding rules. Polyspace stops after rules checking.
`MISRA C 2012 ACG checking`	Check compliance with MISRA C:2012 coding rules using generated code categories. Polyspace stops after guideline checking.

C++ Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA C++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA C ++ coding rules.
`Project configuration and JSF C ++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with JSF C++ coding rules.
`MISRA C++ rule checking`	Check compliance with the MISRA C++ coding rules. Polyspace stops after rules checking.
`JSF C++ rule checking`	Check compliance with JSF C++ coding rules. Polyspace stops after rules checking.

3 Click **Apply** to save your settings.

Configure Polyspace Analysis Options and Properties

From Simulink, you can specify Polyspace options to change the configuration of the Polyspace Analysis. For example, you can specify the processor type and operating system of your target device.

For descriptions of options, see “Analysis Options for C” or “Analysis Options for C++”.

There are two ways to configure analysis options:

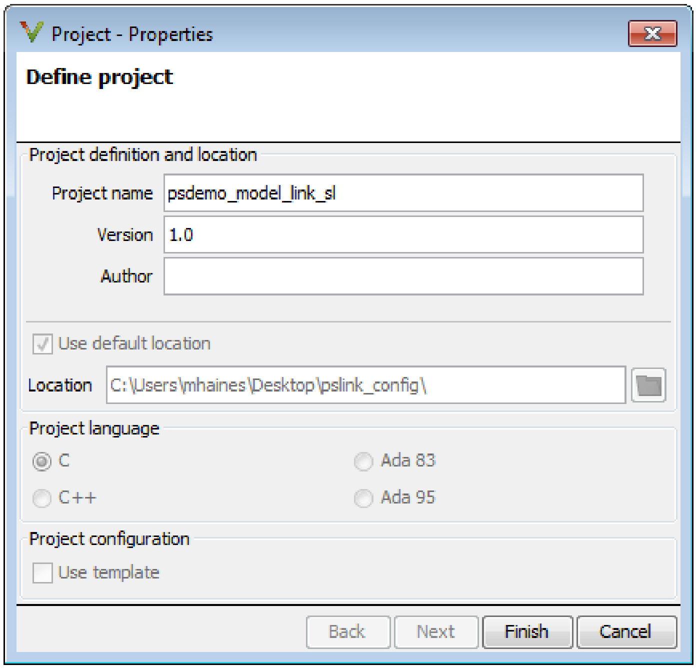
In this section...

Set Advanced Analysis Options

1. From Simulink, select **Code** > **Polyspace** > **Options**.
2. In the Polyspace parameter configuration pane, select **Configure**.

   The Polyspace Configuration window opens.
3. Set options required by your application.

   The first time you open the configuration, the software sets certain options by default depending on your code generator.
4. On the toolbar, click the **Project properties** icon .


Project - Properties

Define project

Project definition and location

Project name: psdemo_model_link_sl

Version: 1.0

Author:

[✓] Use default location

Location: C:\Users\mhaines\Desktop\pslink_config\

Project language

● C ○ Ada 83

○ C++ ○ Ada 95

Project configuration

☐ Use template

Save a Polyspace Configuration File Template

During a batch run, you may want use different configurations. At the MATLAB command-line, use `pslinkfun('settemplate',...)` to apply a configuration defined by a configuration file template.

To create a configuration file template:

1. In the Simulink model window, select **Code** > **Polyspace** > **Options**. The Parameter Configuration window opens to the Polyspace pane.
2. Click **Configure**.

   The Polyspace Configuration window opens. Use this pane to customize the target and cross compiler.
3. Save your changes and close.
4. Make a copy of the updated project configuration file, for example, `my_first_code_polyspace.psprj`.

5. Rename the copy, for example, `my_cross_compiler.psprj`. This is your new configuration file template.

To use a configuration template:

- Run the `pslinkfun` command in the MATLAB Command Window. For example:

  ```
  pslinkfun('settemplate','C:\Work\my_cross_compiler.psprj')
  ```
- Add the file in the Parameter Configuration window. See “Use a Custom Configuration File” on page 9-9.

Use a Custom Configuration File

If you already have a configuration you want to use, you can add the configuration file to your project.

1. From Simulink, select **Code** > **Polyspace** > **Options**.
2. In the Polyspace parameter configuration pane, select **Use custom project file**.
3. In the text box, enter the full path to a `.psprj` file, or click **Browse for project file** to browse for a `.psprj` file.

Remove Polyspace Options From Simulink Model

You can remove Polyspace configuration information from your Simulink model.

For a top model:

1. Select **Code** > **Polyspace** > **Remove Options from Current Configuration**.
2. Save the model.

For a Model block or subsystem:

1. Right-click the Model block or subsystem.
2. From the context menu, select **Polyspace** > **Remove Options from Current Configuration**.
3. Save the model.

See Also

`pslinkfun` | `pslinkoptions`

Related Examples

- “Save a Polyspace Configuration File Template” on page 9-8

More About

- “Embedded Coder Considerations”
- “TargetLink Considerations”
- “Recommended Polyspace Bug Finder Options for Analyzing Generated Code”

Set Custom Target Settings

If your target has specific setting, you can analyze your code in context of those settings. For example, if you want to analyze code generated for a 16-bit target processor, you must specify header files for your 16-bit compiler. The software automatically identifies the compiler from the Simulink model. If the compiler is 16-bit and you do not specify the relevant header files, the software produces an error when you try to run an analysis.

Note: For a 32-bit or 64-bit target processor, the software automatically specifies the default header file.

---

1 In the Simulink model window, select **Code** > **Polyspace** > **Options**. The Parameter Configuration window opens to the Polyspace pane.

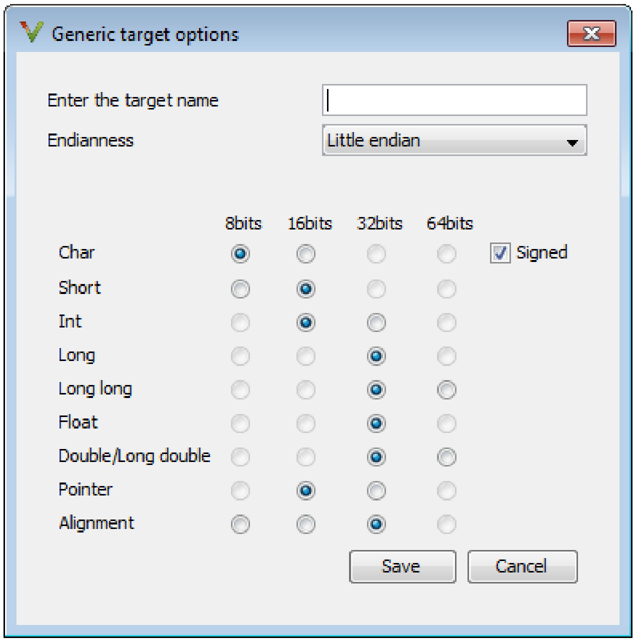
2 Click **Configure**.

   The Polyspace Configuration window opens. Use this pane to customize the target and cross compiler.

3 From the **Configuration** tree, expand the **Target & Compiler** node.

4 In the **Target Environment** section, use the **Target processor type** option to define the size of data types.

   a From the drop-down list, select `mcpu...(Advanced)`. The Generic target options dialog box opens.



Enter the target name

Endianness: Little endian

	8bits	16bits	32bits	64bits	
Char	●	○	○	○	✓ Signed
Short	○	●	○	○	
Int	○	●	○	○	
Long	○	○	●	○	
Long long	○	○	●	○	
Float	○	○	●	○	
Double/Long double	○	○	●	○	
Pointer	○	●	○	○	
Alignment	○	○	●	○	

Use this dialog box to create a new target and specify data types for the target. Then click **Save**.

5 From the Configuration tree, select **Target & Compiler** > **Macros**. Use the **Preprocessor definitions** section to define preprocessor macros for your cross-compiler.

To add a macro, in the **Macros** table, select ➕. In the new line, enter the required text.

To remove a macro, select the macro and click ✖.

Note: If you use the LCC cross-compiler, then you must specify the `MATLAB_MEX_FILE` macro.

---

6 Select **Target & Compiler** > **Environment Settings**.

7 In the **Include folders** (or **Include**) section, specify a folder (or header file) path by doing one of the following:

- Select ➕ and enter the folder or file path.
- Select 📁 and use the dialog box to navigate to the required folder (or file).

You can remove an item from the displayed list by selecting the item and then clicking ✖.

8 Save your changes and close.

To use your configuration settings in other projects, see “Save a Polyspace Configuration File Template” on page 9-8.

Set Up Remote Batch Analysis

By default, the Polyspace software runs locally. To specify a remote analysis:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens, displaying the **Polyspace** pane.
2. Select **Configure**.
3. In the Polyspace Configuration window, select the **Distributed Computing** pane.
4. Select the **Batch** check box.
5. If you use Polyspace Metrics as a results repository, select **Add to results repository**.

   Before running your must also make sure you are connected to a Server.
6. From the toolbar, select **Options** > **Preferences**. For help filling in this dialog, see "Configure Polyspace Preferences".
7. Close the configuration window and save your changes.
8. Select **Apply**.

Manage Results

In this section...
"Open Polyspace Results Automatically" on page 9-15
"Specify Location of Results" on page 9-16
"Save Results to a Simulink Project" on page 9-17

Polyspace creates a set of analysis results

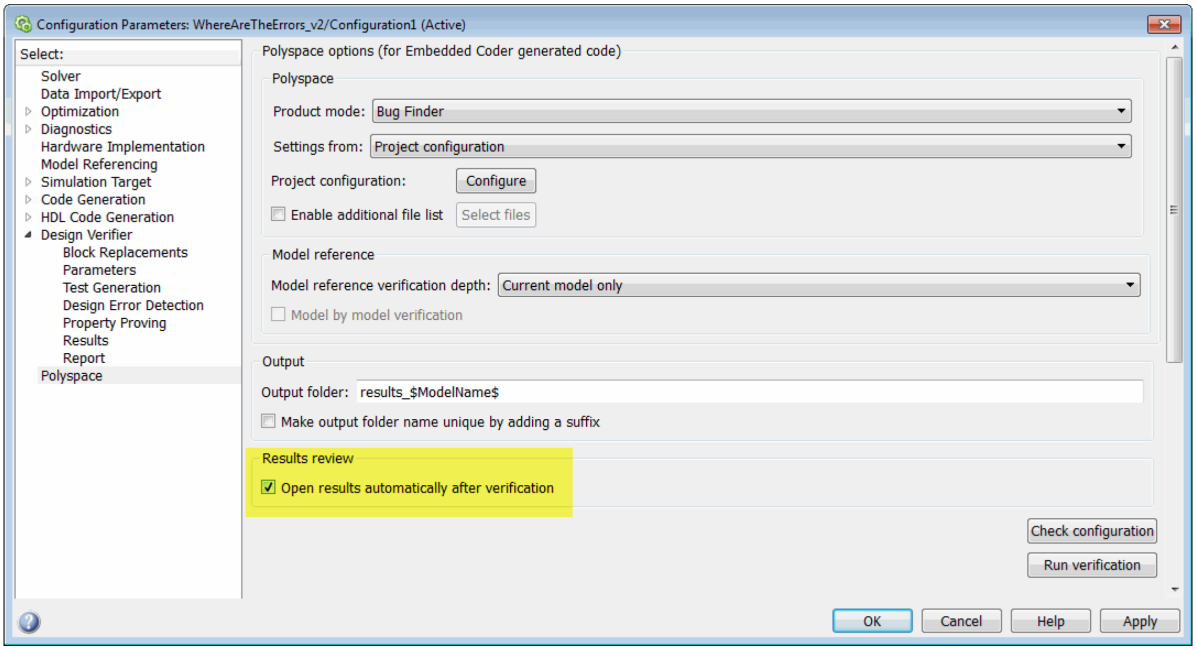
Open Polyspace Results Automatically

You can configure the software to automatically open your Polyspace results after you start the analysis. If you are doing a remote analysis, the Polyspace Metrics webpage opens. When the remote job is complete, you can download your results from Polyspace Metrics. If you are doing a local analysis, when the local job is complete, the Polyspace environment opens the results in the Polyspace interface.

To configure the results to open automatically:

1 From the model window, select **Code** > **Polyspace** > **Options**.

  The Polyspace pane opens.


Configuration Parameters: WhereAreTheErrors_v2/Configuration1 (Active)

Polyspace options (for Embedded Coder generated code)

Polyspace

Product mode: Bug Finder

Settings from: Project configuration

Project configuration: Configure

☐ Enable additional file list

Model reference

Model reference verification depth: Current model only

☐ Model by model verification

Output

Output folder: results_$ModelName$

☐ Make output folder name unique by adding a suffix

Results review

☑ Open results automatically after verification

2 In the Results review section, select **Open results automatically after verification**.

3 Click **Apply** to save your settings.

Specify Location of Results

By default, the software stores your results in *`Current Folder`*`\results_`*`model_name`*. Every time you rerun, your old results are over written. To customize these options:

1 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens to the Polyspace pane.

2 In the **Output folder** field, specify a full path for your results folder. By default, the software stores results in the current folder.

3 If you want to avoid overwriting results from previous analyses, select **Make output folder name unique by adding a suffix**.

Instead of overwriting an existing folder, the software specifies a new location for the results folder by appending a unique number to the folder name.

Save Results to a Simulink Project

By default, the software stores your results in `*Current Folder*\results_*model_name*`. If you use a Simulink project for your model work, you can store your Polyspace results there as well for better organization. To add your results to a Simulink Project:

1 Open your Simulink project.

2 From the Simulink model window, select **Code** > **Polyspace** > **Options**. The Configuration Parameters dialog box opens with the Polyspace pane displayed.

3 Select **Add results to current Simulink Project**.

4 Run your analysis.

   Your results are saved to the Simulink project you opened in step 1.

Specify Signal Ranges

If you constrain signals in your Simulink model to lie within specified ranges, Polyspace software automatically applies these constraints during verification of the generated code. This can improve the precision of your results.

You can specify a range for a model signal by:

- Applying constraints through source block parameters. See “Specify Signal Range through Source Block Parameters” on page 9-18.
- Constraining signals through the base workspace. See “Specify Signal Range through Base Workspace” on page 9-20.

Note: You can also manually define data ranges using the DRS feature in the Polyspace verification environment. If you manually define a DRS file, the software automatically appends any signal range information from your model to the DRS file. However, manually defined DRS information overrides information generated from the model for all variables.

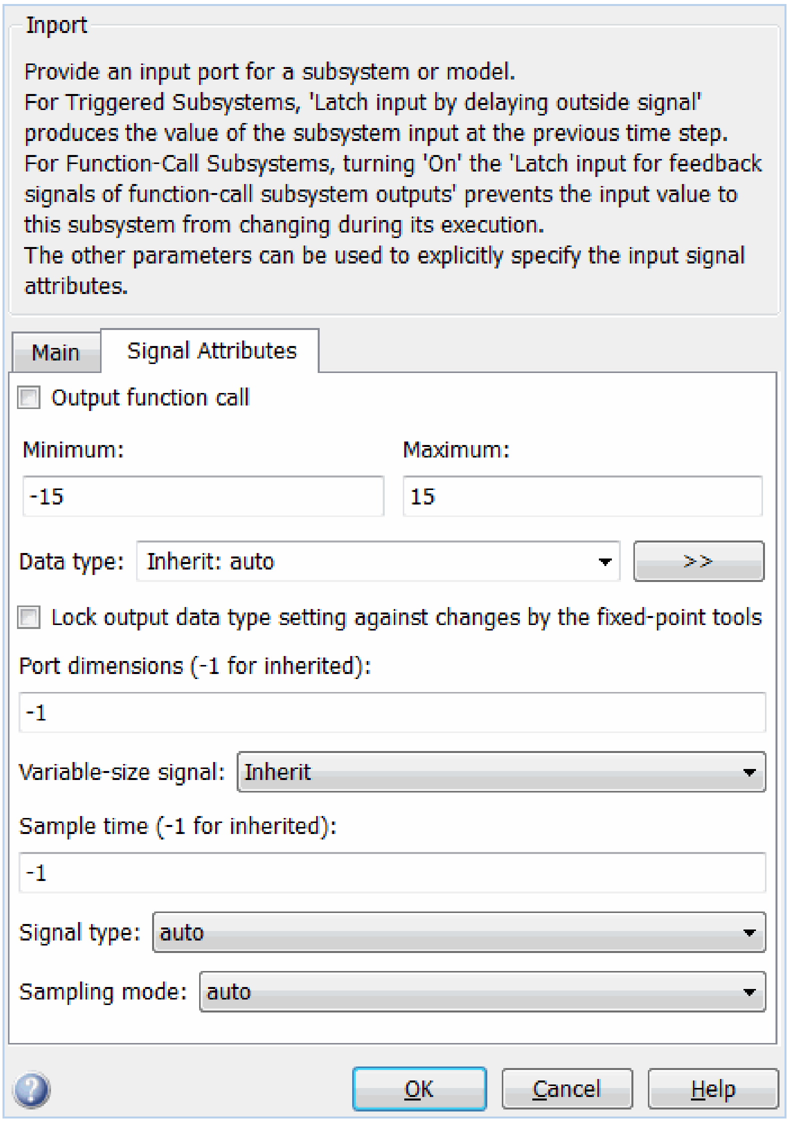
Specify Signal Range through Source Block Parameters

You can specify a signal range by applying constraints to source block parameters.

Specifying a range through source block parameters is often easier than creating signal objects in the base workspace, but must be repeated for each source block. For information on using the base workspace, see “Specify Signal Range through Base Workspace” on page 9-20.

To specify a signal range using source block parameters:

1. Double-click the source block in your model. The Source Block Parameters dialog box opens.
2. Select the **Signal Attributes** tab.
3. Specify the **Minimum** value for the signal, for example, -15.
4. Specify the **Maximum** value for the signal, for example, 15.


Inport

Provide an input port for a subsystem or model.
For Triggered Subsystems, 'Latch input by delaying outside signal' produces the value of the subsystem input at the previous time step.
For Function-Call Subsystems, turning 'On' the 'Latch input for feedback signals of function-call subsystem outputs' prevents the input value to this subsystem from changing during its execution.
The other parameters can be used to explicitly specify the input signal attributes.

☐ Output function call

Minimum: -15

Maximum: 15

Data type: Inherit: auto

☐ Lock output data type setting against changes by the fixed-point tools

Port dimensions (-1 for inherited): -1

Variable-size signal: Inherit

Sample time (-1 for inherited): -1

Signal type: auto

Sampling mode: auto

5 Click **OK**.

Specify Signal Range through Base Workspace

You can specify a signal range by creating signal objects in the MATLAB workspace. This information is used to initialize each global variable to the range of valid values, as defined by the min-max information in the workspace.

Note: You can also specify a signal range by applying constraints to individual source block parameters. This method can be easier than creating signal objects in the base workspace, but must be repeated for each source block. For more information, see "Specify Signal Range through Source Block Parameters" on page 9-18.

To specify an input signal range through the base workspace:

1 Configure the signal to use, for example, the `ExportedGlobal` storage class:

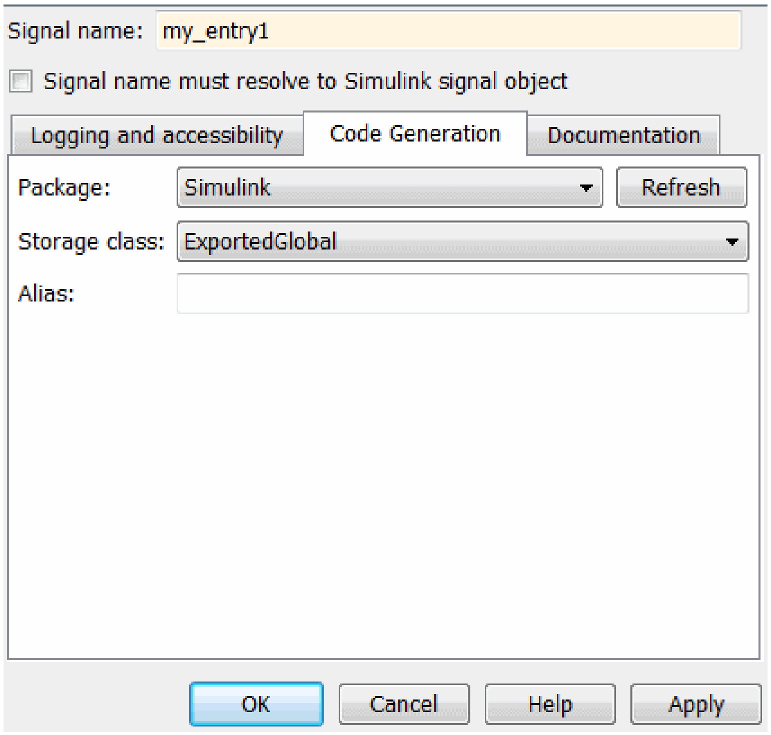
   a Right-click the signal. From the context menu, select **Properties**. The Signal Properties dialog box opens.

   b In the **Signal name** field, enter a name, for example, `my_entry1`.

   c Select the **Code Generation** tab.

   d From the **Package** drop-down menu, select `Simulink`.

   e In the **Storage class** drop-down menu, select `ExportedGlobal`.


Signal name: my_entry1

☐ Signal name must resolve to Simulink signal object

Code Generation

Package: Simulink

Storage class: ExportedGlobal

Alias:

- **f** Click **OK**, which applies your changes and closes the dialog box.

2 Using Model Explorer, specify the signal range:

- **a** Select **Tools** > **Model Explorer** to open Model Explorer.
- **b** From the **Model Hierarchy** tree, select **Base Workspace**.
- **c** Click the `Add Simulink Signal` button to create a signal. Rename this signal, for example, `my_entry1`.
- **d** Set the **Minimum** value for the signal, for example, to `-15`.
- **e** Set the **Maximum** value for the signal, for example, to `15`.
- **f** From the **Storage class** drop-down list, select `ExportedGlobal`.
- **g** Click **Apply**.

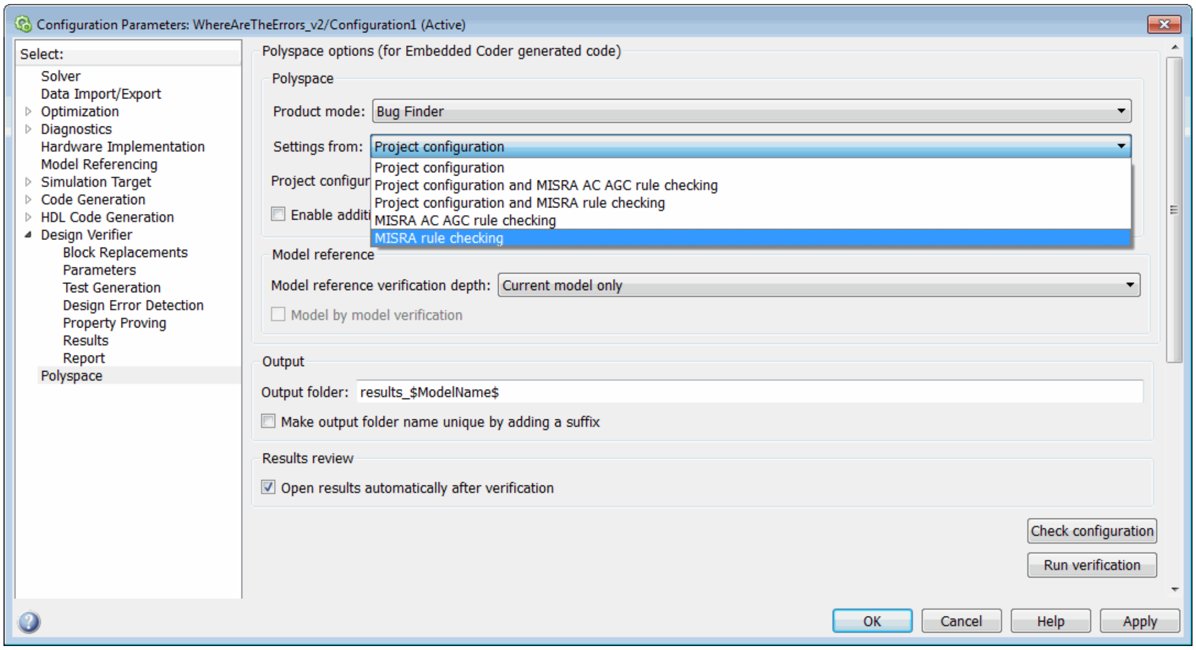
Run Polyspace on Generated Code

Specify Type of Analysis to Perform

Before running Polyspace, you can specify what type of analysis you want to run. You can choose to run code analysis, coding rules checking, or both.

To specify the type of analysis to run:

1. From the Simulink model window, select **Code** > **Polyspace** > **Options**. The **Configuration Parameter** window opens to the **Polyspace** options pane.



Configuration Parameters: WhereAreTheErrors_v2/Configuration1 (Active)

Polyspace options (for Embedded Coder generated code)

Polyspace

Product mode: Bug Finder

Settings from: Project configuration

Project configur

☐ Enable additi

Model reference

Model reference verification depth: Current model only

☐ Model by model verification

Output

Output folder: results_$ModelName$

☐ Make output folder name unique by adding a suffix

Results review

☑ Open results automatically after verification

2. In the **Settings from** drop-down menu, select the type of analysis you want to perform.

   Depending on the type of code generated, different settings are available. The following tables describe the different settings.

   C Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA AC AGC rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA AC-AGC rule set.
`Project configuration and MISRA rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with MISRA C coding rules.
`MISRA AC AGC rule checking`	Check compliance with the MISRA AC-AGC rule set. Polyspace stops after rules checking.
`MISRA rule checking`	Check compliance with MISRA C coding rules. Polyspace stops after rules checking.

C++ Code Settings

Setting	Description
`Project configuration`	Run Polyspace using the options specified in the **Project configuration**.
`Project configuration and MISRA C++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with the MISRA C ++ coding rules.
`Project configuration and JSF C ++ rule checking`	Run Polyspace using the options specified in the **Project configuration** and check compliance with JSF C++ coding rules.
`MISRA C++ rule checking`	Check compliance with the MISRA C++ coding rules. Polyspace stops after rules checking.
`JSF C++ rule checking`	Check compliance with JSF C++ coding rules. Polyspace stops after rules checking.

3 Click **Apply** to save your settings.

Run Analysis for Embedded Coder

To start Polyspace with:

- Code generated from the top model, from the Simulink model window, select **Code** > **Polyspace** > **Verify Code Generated for** > **Model**.
- All code generated as model referenced code, from the model window, select **Code** > **Polyspace** > **Verify Code Generated for** > **Referenced Model**.
- Model reference code associated with a specific block or subsystem, right-click the Model block or subsystem. From the context menu, select **Verify Code Generated for** > **Selected Subsystem**.

Note: You can also start the Polyspace software from the **Polyspace** configuration parameter pane by clicking **Run verification**.

When the Polyspace software starts, messages appear in the MATLAB Command window:

```
### Starting Polyspace verification for Embedded Coder
### Creating results folder C:\PolySpace_Results\results_my_first_code
                                           for system my_first_code
### Checking Polyspace Model-Link Configuration:
### Parameters used for code verification:
 System                : my_first_code
 Results Folder        : C:\PolySpace_Results\results_my_first_code
 Additional Files      : 0
 Remote                : 0
 Model Reference Depth : Current model only
 Model by Model        : 0
 DRS input mode        : DesignMinMax
 DRS parameter mode    : None
 DRS output mode       : None
...
```

Follow the progress of the analysis in the MATLAB Command window. If you are running a remote, batch, analysis you can follow the later stages through the Polyspace Job Monitor.

The software writes status messages to a log file in the results folder.

Run Analysis for TargetLink

To start the Polyspace software:

1. In your model, select the Target Link subsystem.
2. In the Simulink model window select **Code** > **Polyspace** > **Verify Code Generated for** > **Selected Target Link Subsystem**.

   Messages appear in the MATLAB Command window:

   ```
   ### Starting Polyspace verification for Embedded Coder
   ### Creating results folder results_WhereAreTheErrors_v2
                            for system WhereAreTheErrors_v2
   ### Parameters used for code verification:
    System               : WhereAreTheErrors_v2
    Results Folder       : H:\Desktop\Test_Cases\ModelLink_Testers
                                     \results_WhereAreTheErrors_v2
    Additional Files     : 0
    Verifier settings    : PrjConfig
    DRS input mode       : DesignMinMax
    DRS parameter mode   : None
    DRS output mode      : None
    Model Reference Depth : Current model only
    Model by Model       : 0
   ```

   The exact messages depend on the code generator you use and the Polyspace product. The software writes status messages to a log file in the results folder.

Follow the progress of the software in the MATLAB Command Window. If you are running a remote, batch analysis, you can follow the later stages through the Polyspace Job Monitor

Monitor Progress

In this section...
"Local Analyses" on page 10-7
"Remote Batch Analyses" on page 10-7

Local Analyses

For a local Polyspace runs, you can follow the progress of the software in the MATLAB Command Window. The software also saves the status messages to a log file in the results folder.

Remote Batch Analyses

For a remote analysis, you can follow the initial stages of the analysis in the MATLAB Command window.

Once the compilation phase is complete, you can follow the progress of the software using the Polyspace Job Monitor.

From Simulink, select **Code** > **Polyspace** > **Open Job Monitor**

Check Coding Rules from Eclipse

Activate Coding Rules Checker

This example shows how to activate the coding rules checker before you start an analysis. This activation enables the Polyspace Bug Finder plug-in to search for coding rule violations. You can view the coding rule violations in your analysis results.

1. Open project configuration.
2. On the **Configuration** pane, select **Coding Rules**.
3. Select the check box for the type of coding rules that you want to check.

   For C code, you can check compliance with:

   - MISRA C:2004
   - MISRA AC AGC
   - MISRA C:2012

     If you have generated code, use the **Use generated code requirements** option to use the MISRA C:2012 categories for generated code.
   - Custom coding rules

   For C++ code, you can check compliance with:

   - MISRA C++: 2008
   - JSF C++
   - Custom coding rules
4. For each rule type that you select, from the drop-down list, select the subset of rules to check.

   MISRA C:2004

Option	Description
`required-rules`	All required MISRA C:2004 coding rules.
`all-rules`	AllMISRA C:2004 coding rules (required and advisory).
`SQO-subset1`	A small subset of MISRA C:2004 rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.

Option	Description
`SQO-subset2`	A second subset of rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA C:2004 coding rules that you specify.

MISRA AC AGC

Option	Description
`OBL-rules`	All required MISRA AC AGC coding rules.
`OBL-REC-rules`	All required and recommended MISRA AC AGC coding rules.
`all-rules`	All required, recommended, and readability coding rules.
`SQO-subset1`	A small subset of MISRA AC AGC rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of MISRA AC AGC rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA AC AGC coding rules that you specify.

MISRA C:2012

Option	Description
`mandatory`	All mandatory MISRA C:2012 coding rules. If you have generated code, also use the **Use generated code requirements** option categorization for generated code.
`mandatory-required`	All mandatory and required MISRA C:2012 coding rules. If you have generated code, also use the **Use generated code requirements** option categorization for generated code.
`all`	All MISRA C:2012 coding rules (mandatory, required, and advisory).

Option	Description
`SQO-subset1`	A small subset of MISRA C rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules that include the rules in `SQO-subset1` and contain some additional rules. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A set of MISRA C:2012 coding rules that you specify.

MISRA C++

Option	Description
`required-rules`	All required MISRA C++ coding rules.
`all-rules`	All required and advisory MISRA C++ coding rules.
`SQO-subset1`	A small subset of MISRA C++ rules. In Polyspace Code Prover, observing these rules can reduce the number of unproven results.
`SQO-subset2`	A second subset of rules with indirect impact on the selectivity in addition to `SQO-subset1`. In Polyspace Code Prover, observing the additional rules can further reduce the number of unproven results.
`custom`	A specified set of MISRA C++ coding rules.

JSF C++

Option	Description
`shall-rules`	**Shall** rules are mandatory requirements. These rules require verification.
`shall-will-rules`	All **Shall** and **Will** rules. **Will** rules are intended to be mandatory requirements. However, these rules do not require verification.
`all-rules`	All **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.
`custom`	A set of JSF C++ coding rules that you specify.

5 If you select **Check custom rules**, specify the path to your custom rules file or click **Edit** to create one.

When rules checking is complete, the software displays the coding rule violations in purple on the **Results Summary** pane.

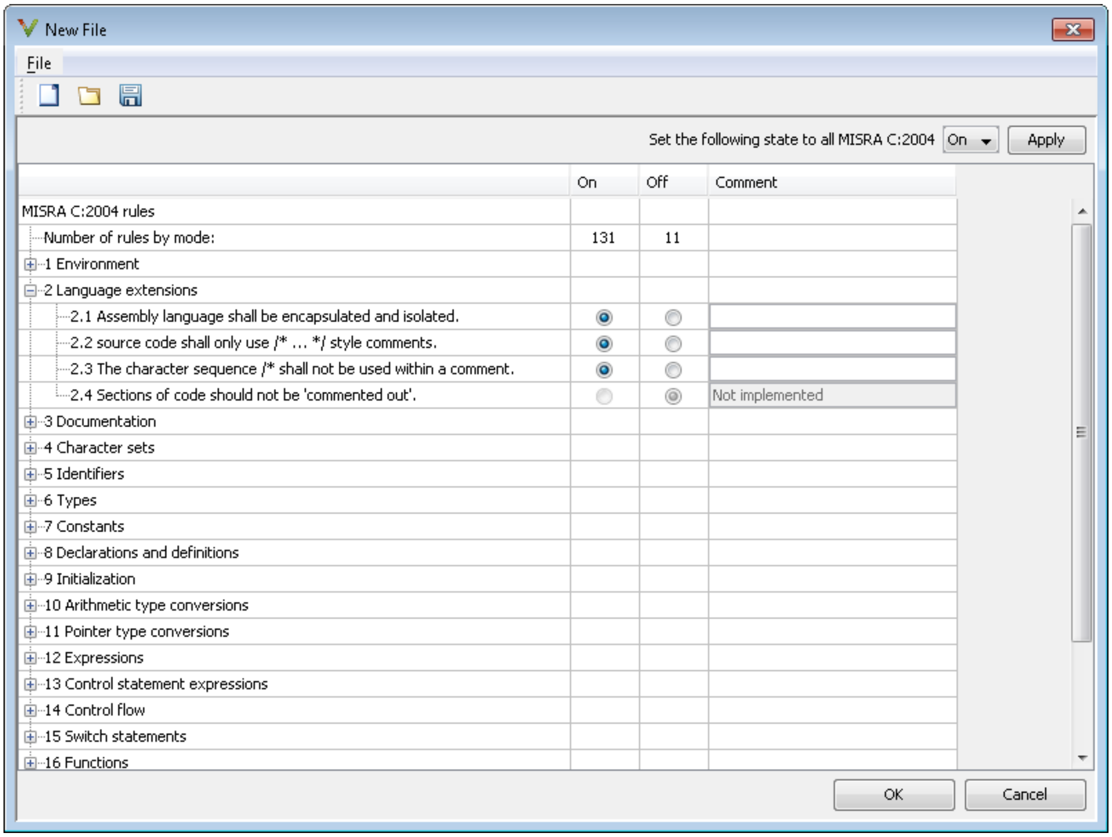
Related Examples

- “Select Specific MISRA or JSF Coding Rules”
- “Create Custom Coding Rules File”

Select Specific MISRA or JSF Coding Rules

This example shows how to specify a subset of MISRA or JSF rules for the coding rules checker. If you select `custom` from the MISRA or JSF drop-down list, you must provide a file that specifies the rules to check.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**.
3. Select the check box for the type of coding rules you wish to check
4. From the corresponding drop-down list, select `custom`. The software displays a new field for your custom file.
5. To the right of this field, click **Edit**. A New File window opens, displaying a table of rules.



	On	Off	Comment
MISRA C:2004 rules			
Number of rules by mode:	131	11	
1 Environment			
2 Language extensions			
2.1 Assembly language shall be encapsulated and isolated.	●	○	
2.2 source code shall only use /* … */ style comments.	●	○	
2.3 The character sequence /* shall not be used within a comment.	●	○	
2.4 Sections of code should not be 'commented out'.	○	●	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			

Select **On** for the rules you want to check.

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

The **Save as** dialog box opens.

7 In the **File** field, enter a name for the rules file.

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

The full path to the rules file appears. To reuse this rules file for other projects, type this path name or use the 📁 icon in the New File window.

Related Examples

- “Activate Coding Rules Checker”
- “Create Custom Coding Rules File”

Create Custom Coding Rules File

This example shows how to create a custom coding rules file. You can use this file to check names or text patterns in your source code against custom rules that you specify. For each rule, you specify a pattern in the form of a regular expression. The software compares the pattern against identifiers in the source code and determines whether the custom rule is violated.

1. **Create Coding Rules File**

   1. Create a Polyspace project. Add `printInitialValue.c` to the project.
   2. On the **Configuration** pane, select **Coding Rules**. Select the **Check custom rules** box.
   3. Click Edit.

      The New File window opens, displaying a table of rule groups.
   4. From the drop-down list **Set the following state to all Custom C**, select `Off`. Click **Apply**.
   5. Expand the **Structs** node. For the option **4.3 All struct fields must follow the specified pattern**:

Column Title	Action
On	Select ●.
Convention	Enter `All struct fields must begin with s_ and have capital letters`
Pattern	Enter `s_[A-Z0-9_]`
Comment	Leave blank. This column is for comments that appear in the coding rules file alone.

2. **Review Coding Rule Violations**

   1. Save the file and run the verification. On the **Results Summary** pane, you see two violations of rule 4.3. Select the first violation.

      a. On the **Source** pane, the line `int a;` is marked.

   b. On the **Check Details** pane, you see the error message you had entered, `All struct fields must begin with s_ and have capital letters`

2. Right-click on the **Source** pane and select **Open Editor**. The file `printInitialValue.c` opens in the **Code Editor** pane or an external text editor depending on your **Preferences**.
3. In the file, replace all instances of `a` with `s_A` and `b` with `s_B`. Rerun the verification.

   The custom rule violations no longer appear on the **Results Summary** pane.

Related Examples

- "Activate Coding Rules Checker"
- "Select Specific MISRA or JSF Coding Rules"

More About

- "Contents of Custom Coding Rules File"

Contents of Custom Coding Rules File

In a custom coding rules file, each rule appears in the following format:

```
N.n off|on
convention=violation_message
pattern=regular_expression
```

- *N.n* — Custom rule number, for example, 1.2.
- `off` — Rule is not considered.
- `on` — The software checks for violation of the rule. After verification, it displays the coding rule violation on the **Results Summary** pane.
- *violation_message* — Software displays this text in an XML file within the *Results*/`Polyspace-Doc` folder.
- *regular_expression* — Software compares this text pattern against a source code identifier that is specific to the rule. See “Custom Coding Rules”.

  The keywords `convention=` and `pattern=` are optional. If present, they apply to the rule whose number immediately precedes these keywords. If `convention=` is not given for a rule, then a standard message is used. If `pattern=` is not given for a rule, then the default regular expression is used, that is, `.*`.

  Use the symbol # to start a comment. Comments are not allowed on lines with the keywords `convention=` and `pattern=`.

  The following example contains three custom rules: 1.1, 8.1, and 9.1.

```
# Custom rules configuration file
1.1  off           # Disable custom rule number 1.1
8.1  on        # Violation of custom rule 8.1 produces a warning
convention=Global constants must begin by G_ and must be in capital letters.
pattern=G_[A-Z0-9_]*
9.1  on    # Non-adherence to custom rule 9.1 produces a warning
convention=Global variables should begin by g_.
pattern=g_.*
```

Related Examples

- “Create Custom Coding Rules File”

Exclude Files From Analysis

This example shows how to exclude certain files from coding rules checking and defect checking.

1. Open the project configuration.
2. In the **Configuration** tree view, select **Inputs & Stubbing**.
3. Select the **Files and folders to ignore** check box.
4. From the corresponding drop-down list, select one of the following:

   - `all-headers` (default) — Excludes header files in the Include folders of your project. For example `.h` or `.hpp` files.
   - `all` — Excludes all include files in the Include folders of your project. For example, if you are checking a large code base with standard or Visual headers, excluding include folders can significantly improve the speed of code analysis.
   - `custom` — Excludes files or folders specified in the **File/Folder** view. To add files to the custom **File/Folder** list, select to choose the files and folders to exclude. To remove a file or folder from the list of excluded files and folders, select the row. Then click .

Related Examples

- "Customize Analysis Options" on page 12-3

Allow Custom Pragma Directives

This example shows how to exclude custom pragma directives from coding rules checking. MISRA C rule 3.4 requires checking that pragma directives are documented within the documentation of the compiler. However, you can allow undocumented pragma directives to be present in your code.

1 Open project configuration.

2 In the **Configuration** tree view, select **Coding Rules**.

3 To the right of **Allowed pragmas**, click ➕.

   In the **Pragma** view, the software displays an active text field.

4 In the text field, enter a pragma directive.

5 To remove a directive from the **Pragma** list, select the directive. Then click ✖.

Related Examples

- "Activate Coding Rules Checker"

Specify Boolean Types

This example shows how to specify data types you want Polyspace to consider as Boolean during MISRA C rules checking. The software applies this redefinition only to data types defined by `typedef` statements. The use of this option may affect the checking of MISRA C:2004 rules 12.6, 13.2, 15.4, and MISRA C:2012 rules 14.4, 16.7.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**.
3. To the right of **Effective boolean types**, click ➕.

   In the **Type** view, the software displays an active text field.
4. In the text field, specify the data type that you want Polyspace to treat as Boolean.
5. To remove a data type from the **Type** list, select the data type. Then click ✖.

Related Examples

- "Activate Coding Rules Checker"

Find Coding Rule Violations

This example shows how to check for coding rule violations alone.

1. Open project configuration.
2. In the **Configuration** tree view, select **Coding Rules**. Activate the desired coding rule checker.
3. In the **Configuration** tree view, select **Bug Finder Analysis**.
4. Clear the **Find defects** check box.
5. Click ▷ Run to run the coding rules checker without checking defects.

   You can view the results by selecting the `*RuleSet*-report.xml` file from the results folder.

Related Examples

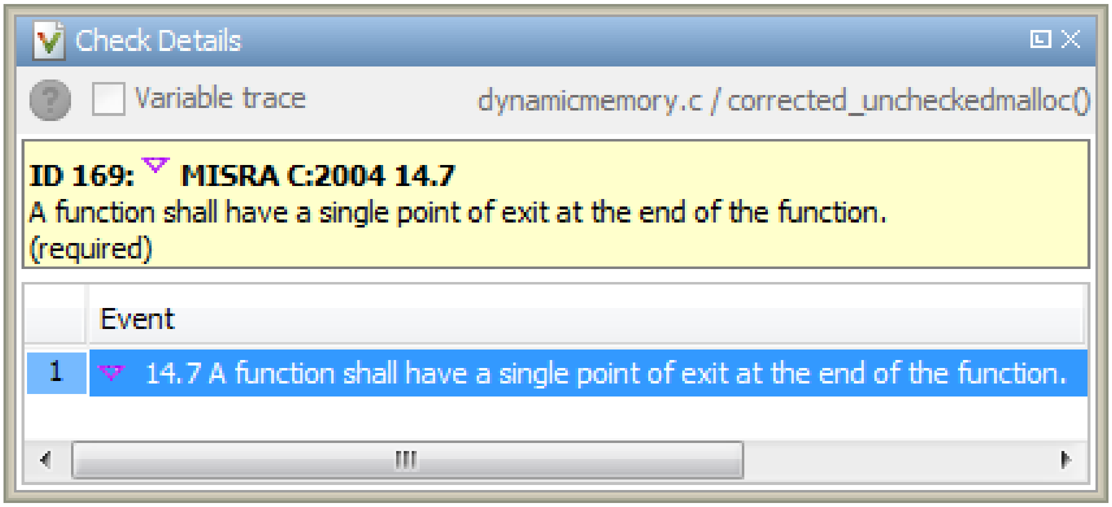
- "Activate Coding Rules Checker"
- "Select Specific MISRA or JSF Coding Rules"
- "Review Coding Rule Violations"

Review Coding Rule Violations

This example shows how to review coding rule violations once code analysis is complete. After analysis, the **Results Summary** tab displays the rule violations with a

- ▽ symbol for predefined coding rules such as MISRA C:2004.
- ▼ symbol for custom coding rules.

1. Select a coding-rule violation on the **Results Summary** pane.
   - The predefined rules such as MISRA or JSF are indicated by ▽ .
   - The custom rules are indicated by ▼ .
2. On the **Check Details** pane, view the location and description of the violated rule. In the source code, the line containing the violation appears highlighted.



dynamicmemory.c / corrected_uncheckedmalloc()

ID 169: ▽ MISRA C:2004 14.7
A function shall have a single point of exit at the end of the function.
(required)

	Event
1	▼ 14.7 A function shall have a single point of exit at the end of the function.

3. Review the violation. On the **Results Summary** pane, select a **Classification** to describe the severity of the issue:
   - `High`
   - `Medium`
   - `Low`
   - `Not a defect`

4. Select a **Status** to describe how you intend to address the issue:

   - `Fix`
   - `Improve`
   - `Investigate`
   - `Justified`

     (This status also marks the result as justified.)
   - `No action planned`

     (This status also marks the result as justified.)
   - `Other`

   You can also define your own statuses.
5. In the comment box, enter additional information about the violation.
6. To open the source file that contains the coding rule violation, on the **Source** pane, right-click the code with the purple check. From the context menu, select **Open Editor**. The file opens in the **Code Editor** pane or an external text editor depending on your **Preferences**.
7. Fix the coding rule violation.
8. When you have corrected the coding rule violations, run the analysis again.

Related Examples

- "Activate Coding Rules Checker"
- "Find Coding Rule Violations"
- "Apply Coding Rule Violation Filters"

Apply Coding Rule Violation Filters

This example shows how to use filters in the **Results Summary** pane to focus on specific kinds of coding rule violations. By default, the software displays both coding rule violations and defects.

Group Violations

1. On the **Results Summary** pane, select **Group by** > **Family**.

   The rules are grouped by numbers. Each group corresponds to a certain code construct.
2. Expand the group nodes to select an individual coding rule violation.

Filter Violations

1. On the **Results Summary** pane, place your cursor on the **Check** column header. Click the filter icon that appears.
2. From the context menu, clear the **All** check box.
3. Select the violated rule numbers that you want to focus on.
4. Click **OK**.

Related Examples

- "Activate Coding Rules Checker"
- "Review Coding Rule Violations"

Find Bugs from Eclipse

Run Analysis

1. In the **Project Explorer**, select the files that you want to analyze.
2. Do one of the following to run an analysis:

   - Right-click on your selection and from the context menu select **Run Polyspace Bug Finder**
   - From the global menu, select **Polyspace** > **Run Polyspace**

   Follow your analysis in the **Output Summary** tab of the Polyspace log window. If your analysis fails, error and warning messages appear on the same tab.

Customize Analysis Options

The software uses a set of default analysis options preconfigured for your coding language and operating system. For each project, you can customize your configuration.

1. From the global menu, select **Polyspace** > **Configure Project**.

   The Polyspace Bug Finder Configuration window appears.
2. Select the different nodes to change your analysis configuration.

   For example:

   a. Select the **Coding Rules** node.
   b. Select **Check MISRA C:2004** to check for violations of MISRA C:2004 coding rules.

For information about the different analysis options, see “Analysis Options for C” or “Analysis Options for C++”.

View Results in Eclipse

Filter and Group Results

This example shows how to filter and group defects on the **Results Summary** tab. To organize your review of results, use filters and groups when you want to:

- Review certain categories of defects in preference to others. For instance, you first want to address the defects resulting from **Missing or invalid return statement**.
- Review only new results found since the last analysis.
- Not address the full set of coding rule violations detected by the coding rules checker.
- Review only those defects that you have already assigned a certain status. For instance, you want to review only those defects to which you have assigned the status, `Investigate`.
- Review defects from a particular file or function. Because of continuity of code, reviewing these defects together can help you organize your review process.

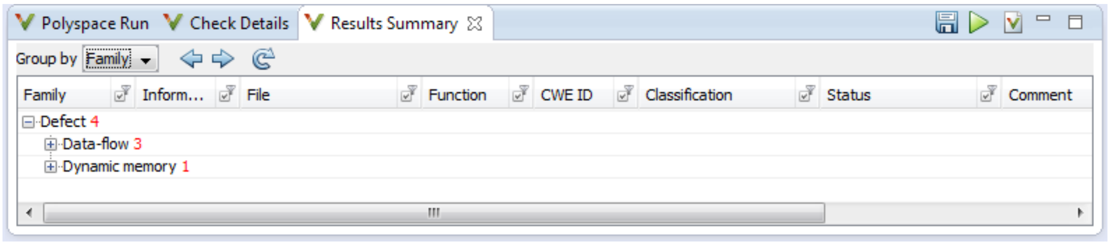
  If you have written the code for a particular source file, you can review the defects only in that file.

Review Defects in a Given Category

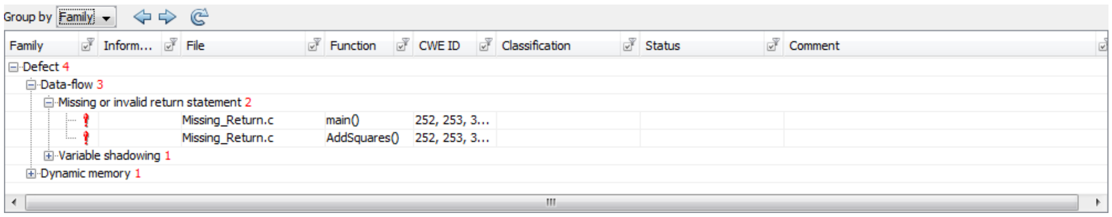
1 To review the defects resulting from **Missing or invalid return statement**:

  a On the **Results Summary** tab, select **Group by** > **Family**.

    The defects are grouped by type.



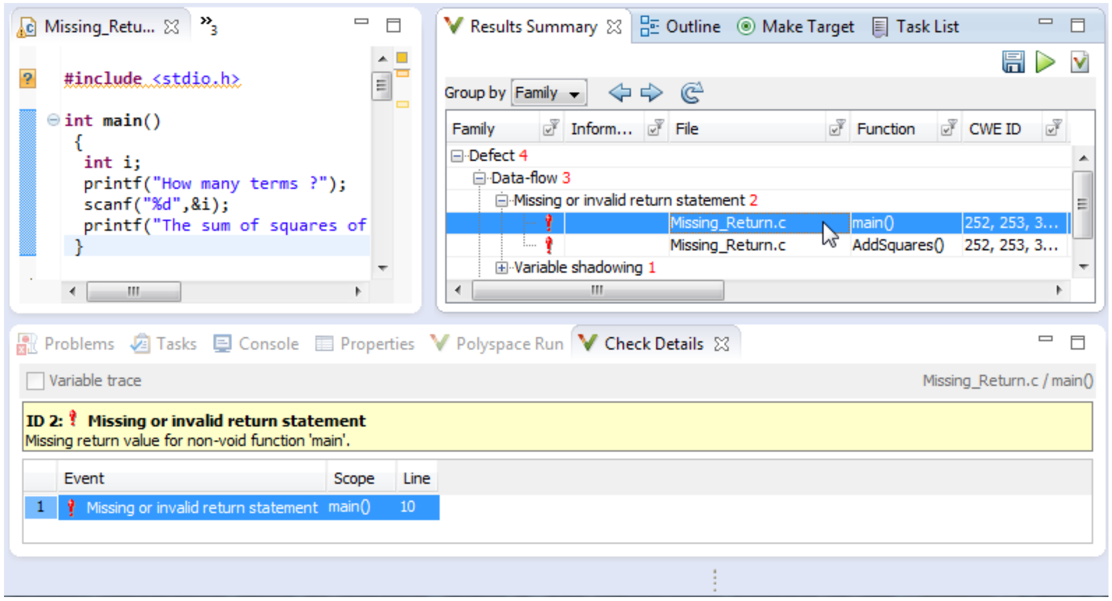
  b Under the **Data-flow** node, expand the subnode **Missing or invalid return statement**.



Group by Family

Family	Inform…	File	Function	CWE ID	Classification	Status	Comment
⊟ Defect 4							
⊟ Data-flow 3							
⊟ Missing or invalid return statement 2							
❗		Missing_Return.c	main()	252, 253, 3…			
❗		Missing_Return.c	AddSquares()	252, 253, 3…			
⊞ Variable shadowing 1							
⊞ Dynamic memory 1							

c To see further information about an instance, select it. The information appears on the **Check Details** tab.



```
#include <stdio.h>

int main()
 {
  int i;
  printf("How many terms ?");
  scanf("%d",&i);
  printf("The sum of squares of
 }
```

Group by Family

Family	Inform…	File	Function	CWE ID
⊟ Defect 4				
⊟ Data-flow 3				
⊟ Missing or invalid return statement 2				
❗		Missing_Return.c	main()	252, 253, 3…
❗		Missing_Return.c	AddSquares()	252, 253, 3…
⊞ Variable shadowing 1				

Variable trace   Missing_Return.c / main()

ID 2: ❗ Missing or invalid return statement
Missing return value for non-void function 'main'.

	Event	Scope	Line
1	❗ Missing or invalid return statement	main()	10

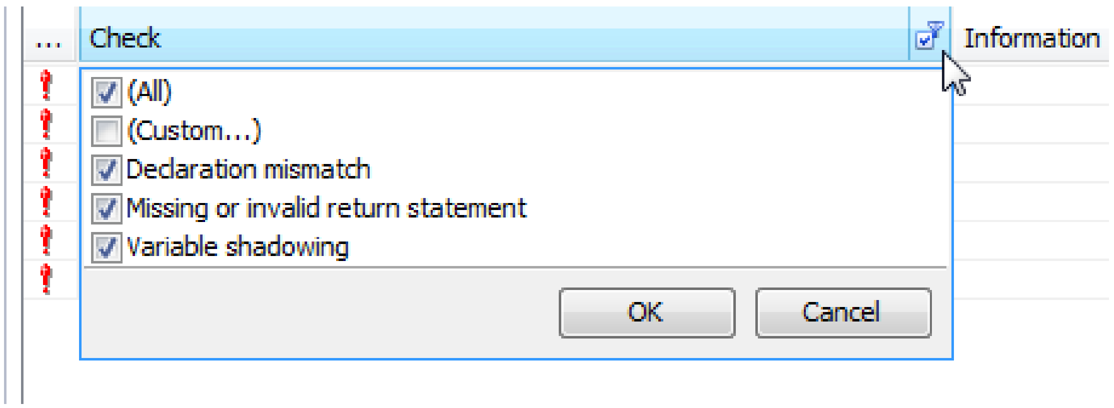
2 To view only the defects resulting from **Missing or invalid return statement**:

a On the **Results Summary** tab, select **Group by** > **None**.

The defects appear ungrouped.

b Click the filter icon on the **Check** column head.

A context menu lists the filter options available.



c Clear the **All** check box.

d Select the **Missing or invalid return statement** check box. Click **OK**.

The **Results Summary** tab displays only the defects resulting from the **Missing or invalid return statement** error.

Review New Results Only

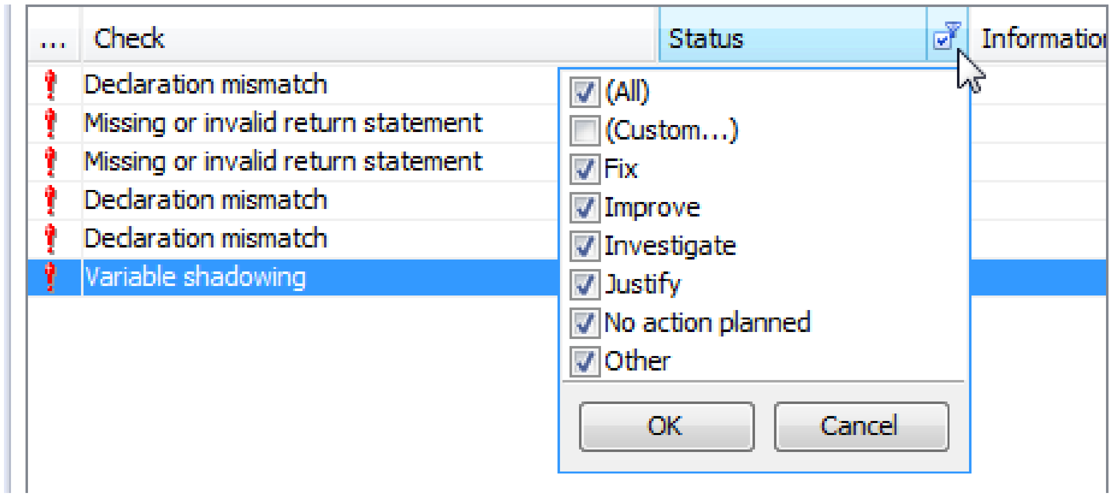
To review only new results found since the last analysis, on the **Results Summary** pane, select **New results**.

Review Defects with Given Status

To review only the defects with `Investigate` status:

1 On the **Results Summary** tab, click the filter icon on the **Status** column head.

A context menu lists the filter options available.



2 Clear the **All** check box.

3 Select the **Investigate** check box. Click **OK**.

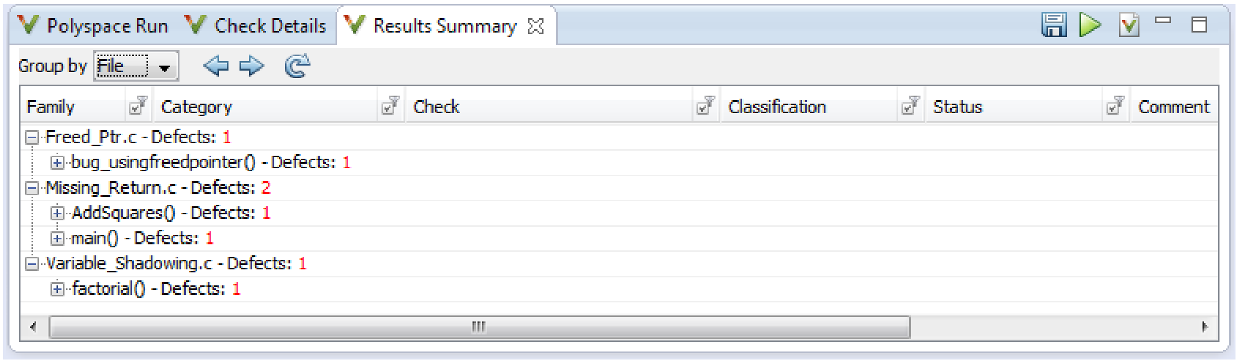
The **Results Summary** tab displays only the defects with the `Investigate` status.

Review Defects in a File

1 To review the defects in the file, `Missing_Return.c`:

a On the **Results Summary** tab, select **Group by** > **File**.

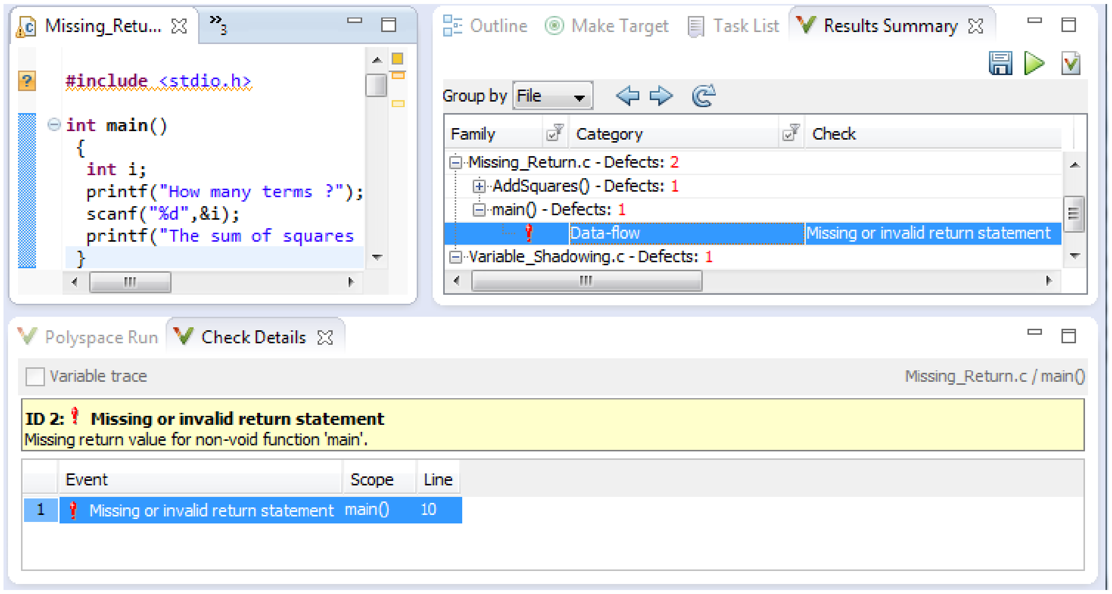
The defects displayed are grouped by files. The file names are sorted alphabetically. Within each file name, the defects are grouped by functions, sorted alphabetically.



Family	Category	Check	Classification	Status	Comment
Freed_Ptr.c - Defects: 1					
bug_usingfreedpointer() - Defects: 1					
Missing_Return.c - Defects: 2					
AddSquares() - Defects: 1					
main() - Defects: 1					
Variable_Shadowing.c - Defects: 1					
factorial() - Defects: 1					

b To view the defects in `Missing_Return.c`, expand a function name under the node, **Missing_Return.c - Defects**.

To view further information on a defect, select the defect. The information appears on the **Check Details** tab.



```
#include <stdio.h>

int main()
 {
  int i;
  printf("How many terms ?");
  scanf("%d",&i);
  printf("The sum of squares
 }
```

Family	Category	Check
Missing_Return.c - Defects: 2		
AddSquares() - Defects: 1		
main() - Defects: 1		
	Data-flow	Missing or invalid return statement
Variable_Shadowing.c - Defects: 1		

ID 2: Missing or invalid return statement
Missing return value for non-void function 'main'.

	Event	Scope	Line
1	Missing or invalid return statement	main()	10

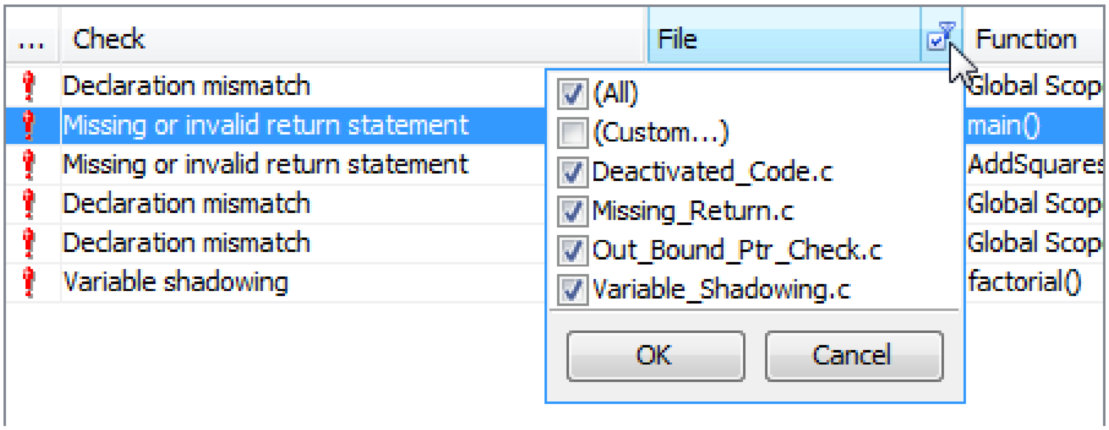
2 To view only the defects in `Missing_Return.c`:

a On the **Results Summary** tab, select **Group by** > **None**.

The **Results Summary** tab displays all results ungrouped.

b Click the filter icon on the **File** column head.

A context menu lists the filter options available.



...	Check	File	Function
	Declaration mismatch		Global Scop
	Missing or invalid return statement		main()
	Missing or invalid return statement		AddSquares
	Declaration mismatch		Global Scop
	Declaration mismatch		Global Scop
	Variable shadowing		factorial()

c Clear the **All** check box.

d Select the **Missing_Return.c** check box. Click **OK**.

The **Results Summary** tab displays only the defects in `Missing_Return.c`.

Tip If you apply a filter on a column on the **Results Summary** pane, the column header displays the number of check boxes selected in the filter menu. Use this information to keep track of the filters you applied.

Related Examples

- "View Results" on page 13-8
- "Review and Fix Results" on page 13-9

View Results

This example shows how to view Polyspace Bug Finder results. After you run an analysis, you can view the results either in Eclipse or from the Polyspace Bug Finder interface.

In this section...

View Results in Eclipse

After you run an analysis in Eclipse, your results automatically appear on the **Results Summary** tab.

- If you closed the **Results Summary** tab, select **Polyspace** > **Show View** > **Show Results Summary view** to reopen the tab.
- If you need to reload the results, select **Polyspace** > **Reload results**.

  This option is useful when you reopen Eclipse or when you are switching between Polyspace projects.

View Results in Polyspace Environment

To view your results in the Polyspace Bug Finder interface, select **Polyspace** > **Open Results in PVE**.

Note: You can view defects, coding rule violations and code metrics from the Eclipse environment. However, you can impose limits on metrics only from the Polyspace environment. For more information, see "Review Code Metrics".

Related Examples

- "Run Analysis"

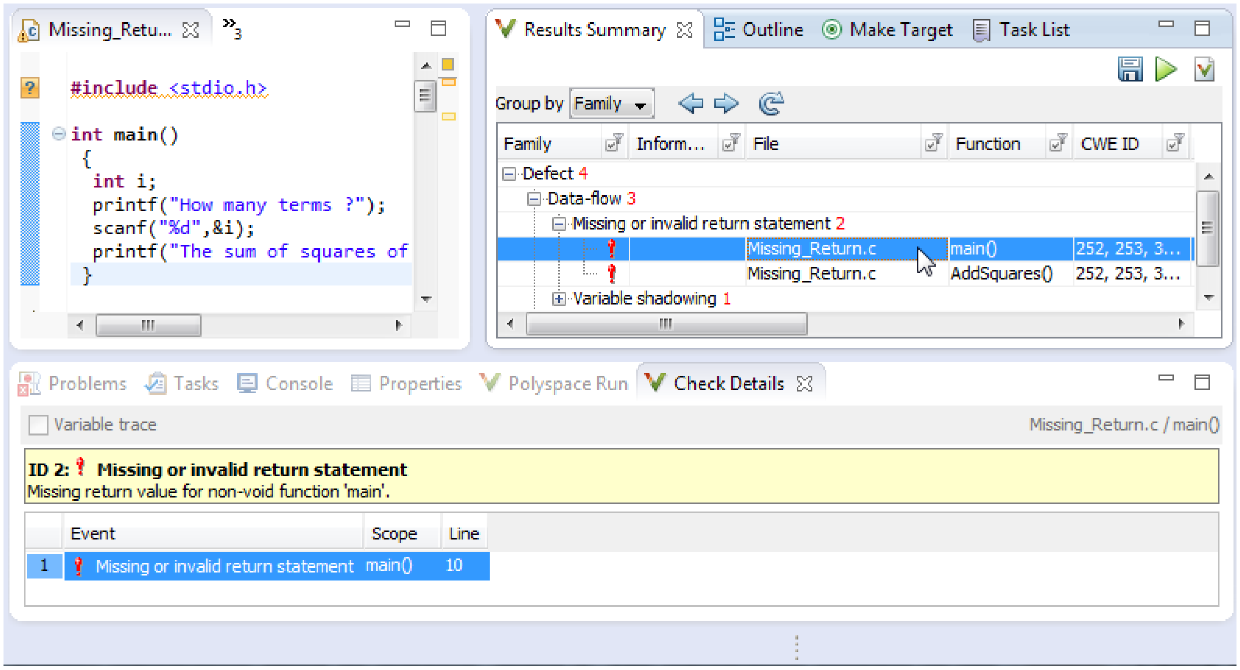
Review and Fix Results

This example shows how to review and comment results obtained from Polyspace Bug Finder analysis. When reviewing results, you can assign a status and classification to the defects and enter comments to describe the results of your review. These actions help you to track the progress of your review and avoid reviewing the same defect twice. If you run successive analyses on the same file, the review status, classification and comments from the previous analysis will be automatically imported into the next.

Review and Comment Individual Defect

1 On the **Results Summary** tab, select the defect that you want to review.

The **Check Details** tab displays information about the current defect. The source code where the defect appears is highlighted.



```
#include <stdio.h>

int main()
 {
  int i;
  printf("How many terms ?");
  scanf("%d",&i);
  printf("The sum of squares of
 }
```

Family	Inform...	File	Function	CWE ID
Defect 4				
Data-flow 3				
Missing or invalid return statement 2				
		Missing_Return.c	main()	252, 253, 3...
		Missing_Return.c	AddSquares()	252, 253, 3...
Variable shadowing 1				

ID 2: Missing or invalid return statement
Missing return value for non-void function 'main'.

	Event	Scope	Line
1	Missing or invalid return statement	main()	10

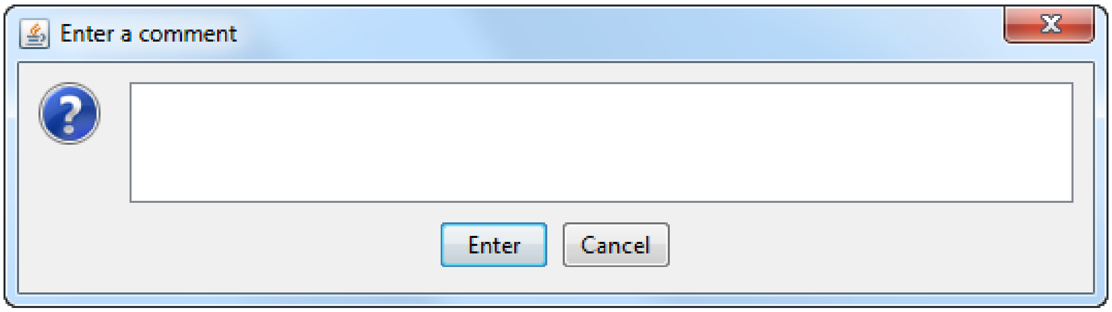
2 On the **Results Summary** tab, enter a **Classification** for the defect to describe its severity:

- `High`
- `Medium`
- `Low`
- `Not a defect`

3 On the **Results Summary** tab, enter a **Status** to describe how you intend to address the defect:

- `Fix`
- `Improve`
- `Investigate`
- `Justified`
- `No action planned`
- `Other`

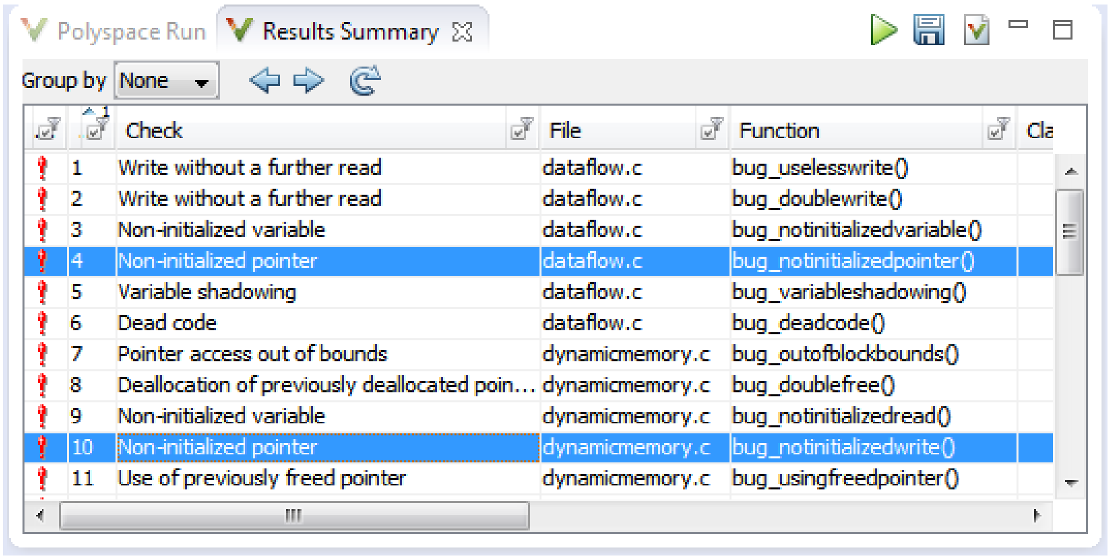
4 On the **Results Summary** tab, click the **Comment** field. Enter remarks, for example, defect or justification information, in the new window that opens.



Review and Comment a Group of Defects

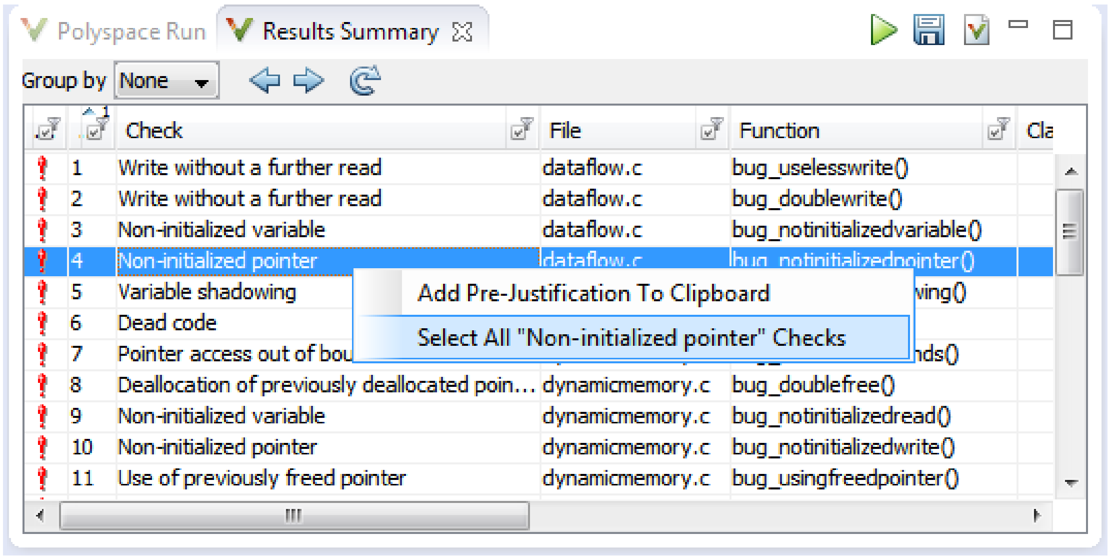
1 On the **Results Summary** tab, select a group of defects using one of the following methods:

- For contiguous defects, click the first defect. Then **Shift**-click the next defect.
- For non-contiguous defects, **Ctrl**-click each defect.



		Check	File	Function	Cla
❗	1	Write without a further read	dataflow.c	bug_uselesswrite()	
❗	2	Write without a further read	dataflow.c	bug_doublewrite()	
❗	3	Non-initialized variable	dataflow.c	bug_notinitializedvariable()	
❗	4	Non-initialized pointer	dataflow.c	bug_notinitializedpointer()	
❗	5	Variable shadowing	dataflow.c	bug_variableshadowing()	
❗	6	Dead code	dataflow.c	bug_deadcode()	
❗	7	Pointer access out of bounds	dynamicmemory.c	bug_outofblockbounds()	
❗	8	Deallocation of previously deallocated poin…	dynamicmemory.c	bug_doublefree()	
❗	9	Non-initialized variable	dynamicmemory.c	bug_notinitializedread()	
❗	10	Non-initialized pointer	dynamicmemory.c	bug_notinitializedwrite()	
❗	11	Use of previously freed pointer	dynamicmemory.c	bug_usingfreedpointer()	

- For defects of a similar category, right-click one defect from that category. From the context menu, select **Select All "*`DefectName`*" Checks**, for instance, **Select All "Non-initialized pointer" Checks**.


Polyspace Run
Results Summary
Group by None
Add Pre-Justification To Clipboard
Select All "Non-initialized pointer" Checks

		Check	File	Function	Cla
❗	1	Write without a further read	dataflow.c	bug_uselesswrite()	
❗	2	Write without a further read	dataflow.c	bug_doublewrite()	
❗	3	Non-initialized variable	dataflow.c	bug_notinitializedvariable()	
❗	4	Non-initialized pointer	dataflow.c	bug_notinitializedpointer()	
❗	5	Variable shadowing		wing()	
❗	6	Dead code			
❗	7	Pointer access out of bou		nds()	
❗	8	Deallocation of previously deallocated poin…	dynamicmemory.c	bug_doublefree()	
❗	9	Non-initialized variable	dynamicmemory.c	bug_notinitializedread()	
❗	10	Non-initialized pointer	dynamicmemory.c	bug_notinitializedwrite()	
❗	11	Use of previously freed pointer	dynamicmemory.c	bug_usingfreedpointer()	

2 On the **Results Summary** tab, enter **Classification**, **Status** and **Comments**. The software applies this information to all selected defects.

Related Examples

- “View Results” on page 13-8
- “Filter and Group Results” on page 13-2

Understanding the Results Views

In this section...
"Results Summary" on page 13-13
"Check Details" on page 13-15

Results Summary

The **Results Summary** tab lists the defects and coding rule violations along with their attributes. To organize your results review, from the **Group by** list on this tab, select one of the following options:

- **None**: Lists defects and coding rule violations in alphabetical order.
- **Family**: Lists results grouped by category. For more information on the defects covered by a category, see "Polyspace Bug Finder Results".
- **Class**: Lists results grouped by class. Within each class, the results are grouped by method. The first group, **Global Scope**, lists results not occurring in a class definition.

  This option is available for `C++` code only.
- **File**: Lists results grouped by file. Within each file, the results are grouped by function.

For each defect, the **Results Summary** pane contains the defect attributes, listed in columns:

Attribute	Description
Family	Group to which the defect belongs. For instance, if you choose the grouping `Checks by File/Function`, this column contains the name of the file and function containing the defect.
ID	Unique identification number of the defect. In the default view on the **Results Summary** pane, the defects appear sorted by this number.
Type	Defect or coding rule violation.
Category	Category of the defect. For more information on the defects covered by a category, see the defect reference pages.

Attribute	Description
Check	Description of the defect
CWE ID	CWE ID-s that correspond to the defect. For more information, see “Mapping Between CWE Identifiers and Defects”.
File	File containing the instruction where the defect occurs
Class	Class containing the instruction where the defect occurs. If the defect is not inside a class definition, then this column contains the entry, `Global Scope`.
Function	Function containing the instruction where the defect occurs. If the function is a method of a class, it appears in the format *`class_name`*`::`*`function_name`*.
Classification	Level of severity you have assigned to the defect. The possible levels are: • `High` • `Medium` • `Low` • `Not a defect`
Status	Review status you have assigned to the check. The possible statuses are: • `Fix` • `Improve` • `Investigate` • `Justified` • `No action planned` • `Other`
Comments	Comments you have entered about the check

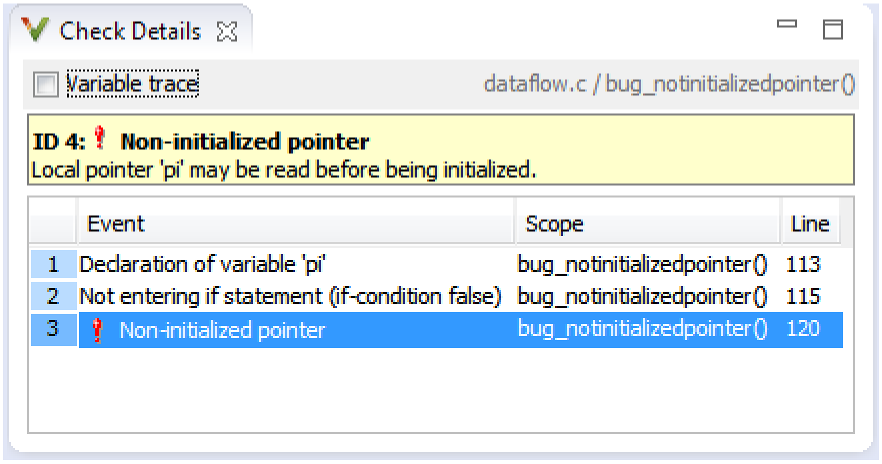
To show or hide any of the columns, right-click anywhere on the column titles. From the context menu, select or clear the title of the column that you want to show or hide.

Using this pane, you can:

- Navigate through the checks. For more information, see “Review and Fix Results” on page 13-9.
- Organize your check review using filters on the columns. For more information, see “Filter and Group Results” on page 13-2.

Check Details

The **Check Details** pane contains detailed information about a specific defect. Select a defect on the **Results Summary** pane to reveal further information about the defect on the **Check Details** pane.



ID 4: ❗ **Non-initialized pointer**
Local pointer 'pi' may be read before being initialized.

	Event	Scope	Line
1	Declaration of variable 'pi'	bug_notinitializedpointer()	113
2	Not entering if statement (if-condition false)	bug_notinitializedpointer()	115
3	❗ Non-initialized pointer	bug_notinitializedpointer()	120

- The top right hand corner shows the file and function containing the defect, in the format *`file_name`* / *`function_name`*.
- The yellow box contains the name of the defect, along with an explanation.
- The **Event** column lists the sequence of code instructions causing the defect. The **Scope** column lists the name of the function containing the instructions. The **Line** column lists the line number of the instructions.
- The **Variable trace** check box when selected reveals an additional set of instructions that are related to the defect.
- The ? button allows you to access documentation for the defect.

Check Coding Rules from Microsoft Visual Studio

Activate C++ Coding Rules Checker

To check coding rule compliance, before running an analysis, you must set an option in your project. Polyspace software finds the violations during the compile phase. You can view coding rule violations alongside your analysis results.

To set the rule checking option:

1. Select the files you wish to analyze.
2. Right-click on your selection and select **Edit Polyspace Configuration**.
3. In the Polyspace Bug Finder Configuration window, from the Configuration tree, select **Coding Rules**.
4. Under **Coding Rules**, select the check box next to the type of coding rules you wish to check.

   For C++ code, you can check compliance with MISRA C++ or JSF C++, and a custom rules file.
5. For MISRA and JSF rule checking, you can select a subset of rules to check from the corresponding drop-down list.

   The tables below show the options for each coding rule set:

   MISRA C++

Option	Explanation
`required-rules`	All *required* MISRA C++ coding rules. Violations are reported as warnings.
`all-rules`	All *required* and *advisory* MISRA C++ coding rules. Violations are reported as warnings.
`SQO-subset1`	A subset of MISRA C++ rules that have a direct impact on the selectivity. Violations are reported as warnings. For more information, see "Software Quality Objective Subsets (C++)" on page 2-62.
`SQO-subset2`	A second subset of rules that have an indirect impact on the selectivity, as well as the rules contained in `SQO-subset1`. Violations are reported as warnings. For more information, see "Software Quality Objective Subsets (C++)" on page 2-62.

Option	Explanation
`custom`	A specified set of MISRA C++ coding rules. When you select this option, you must specify the MISRA C++ rules to check and whether to report an error or warning for violations of each rule. For more information, see "Select Specific MISRA or JSF Coding Rules" on page 3-6.

JSF C++

Option	Explanation
`shall-rules`	All **Shall** rules, which are mandatory rules that require checking.
`shall-will-rules`	All **Shall** and **Will** rules. **Will** rules are mandatory rules that do not require checking.
`all-rules`	All **Shall**, **Will**, and **Should** rules. **Should** rules are advisory rules.
`custom`	A specified set of JSF C++ coding rules. When you select this option, you must specify the JSF C++ rules to check and whether to report an error or warning for violations of each rule. For more information, see "Select Specific MISRA or JSF Coding Rules" on page 3-6.

6. For Custom rule checking, in the corresponding field, specify the path to your custom rules file or click **Edit** to create one. See "Create Custom Coding Rules" on page 3-9 for more information.
7. Save you changes and close the configuration window.

   When you run an analysis, Polyspace checks coding rule compliance during the compilation phase of the analysis.

Exclude Files From Analysis

This example shows how to exclude files from coding rules checking and defect checking. Excluding header files, include files, or files your are not working on allows you focus on defects in your purview.

1. Open the project configuration.
2. In the **Configuration** tree view, select **Inputs & Stubbing**.
3. Select the **Files and folders to ignore** check box.
4. From the corresponding drop-down list, select one of the following:

   - `all-headers` (default) — Excludes header files in the Include folders of your project. For example `.h` or `.hpp` files.
   - `all` — Excludes all include files in the Include folders of your project. For example, if you are checking a large code base with standard or Visual headers, excluding include folders can significantly improve the speed of code analysis.
   - `custom` — Excludes files or folders specified in the **File/Folder** view. To add files to the custom **File/Folder** list, select 🗀 to choose the files and folders to exclude. To remove a file or folder from the list of excluded files and folders, select the row. Then click ✖.

Related Examples

- "Customize Polyspace Options"

Find Bugs from Microsoft Visual Studio

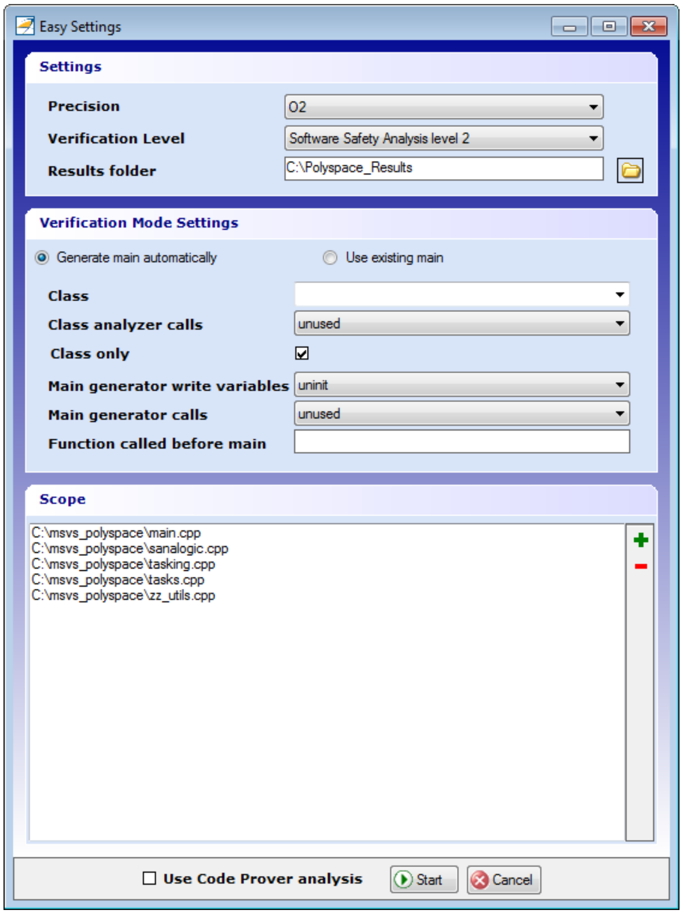
Run Polyspace in Visual Studio

To set up and start an analysis:

1 In the **Solution Explorer** view, select one or more files that you want to analyze.

2 Right-click the selection, and select **Polyspace Verification**.

The Easy Settings dialog box opens.



Easy Settings

Settings

Precision	O2
Verification Level	Software Safety Analysis level 2
Results folder	C:\Polyspace_Results

Verification Mode Settings

● Generate main automatically ○ Use existing main

Class	
Class analyzer calls	unused
Class only	☑
Main generator write variables	uninit
Main generator calls	unused
Function called before main	

Scope

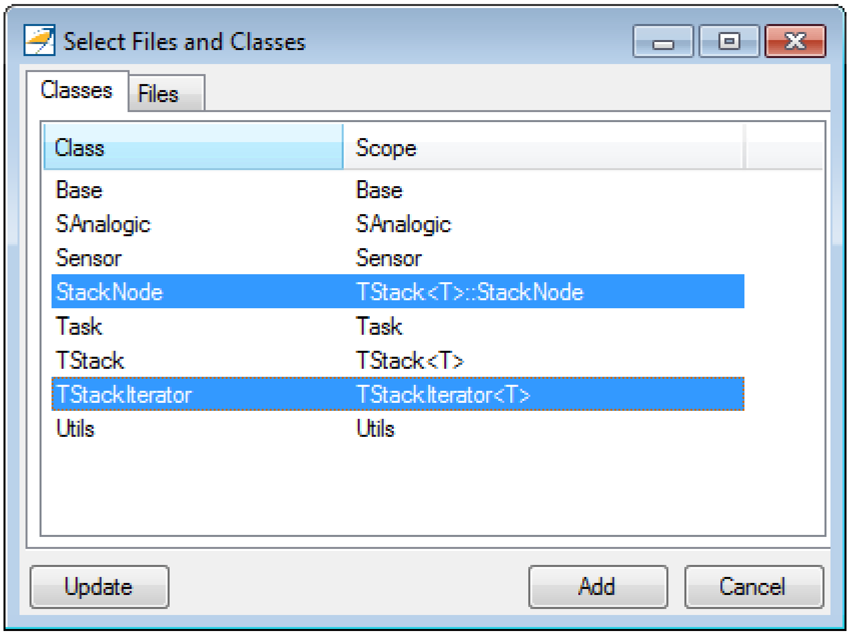
```
C:\msvs_polyspace\main.cpp
C:\msvs_polyspace\sanalogic.cpp
C:\msvs_polyspace\tasking.cpp
C:\msvs_polyspace\tasks.cpp
C:\msvs_polyspace\zz_utils.cpp
```

☐ **Use Code Prover analysis**

3 In the Easy Settings dialog box, you can specify the following options for your analysis:

- Under **Settings**, configure the following:
  - **Precision** — Precision of analysis
  - **Passes** — Level of analysis
  - **Results folder** – Location where software stores analysis results
- Under **Verification Mode Settings**, configure the following:
  - **Generate main** — Polyspace generates a `main` or **Use existing** — Polyspace uses an existing main
  - **Class** — Name of class to analyze
  - **Class analyzer calls** — Functions called by generated `main`
  - **Class only** — Analysis of class contents only
  - **Main generator write** — Type of initialization for global variables
  - **Main generator calls** — Functions (not in a class) called by generated `main`
  - **Function called before main** — Function called before the generated `main`
- Under **Scope**, you can modify the list of files and C++ classes to analyze.

  a Select ➕. The Select Files and Classes dialog box opens.



Class	Scope
Base	Base
SAnalogic	SAnalogic
Sensor	Sensor
StackNode	TStack<T>::StackNode
Task	Task
TStack	TStack<T>
TStackIterator	TStackIterator<T>
Utils	Utils

b Select the classes that you want to analyze, then click **Add**.

In the Configuration pane in the Polyspace environment, you can configure advanced options not in the Easy Settings dialog box. See “Customize Polyspace Options” on page 15-8.

4 Make sure the **Use Code Prover analysis** check box is cleared.

5 Click **Start** to start the analysis.

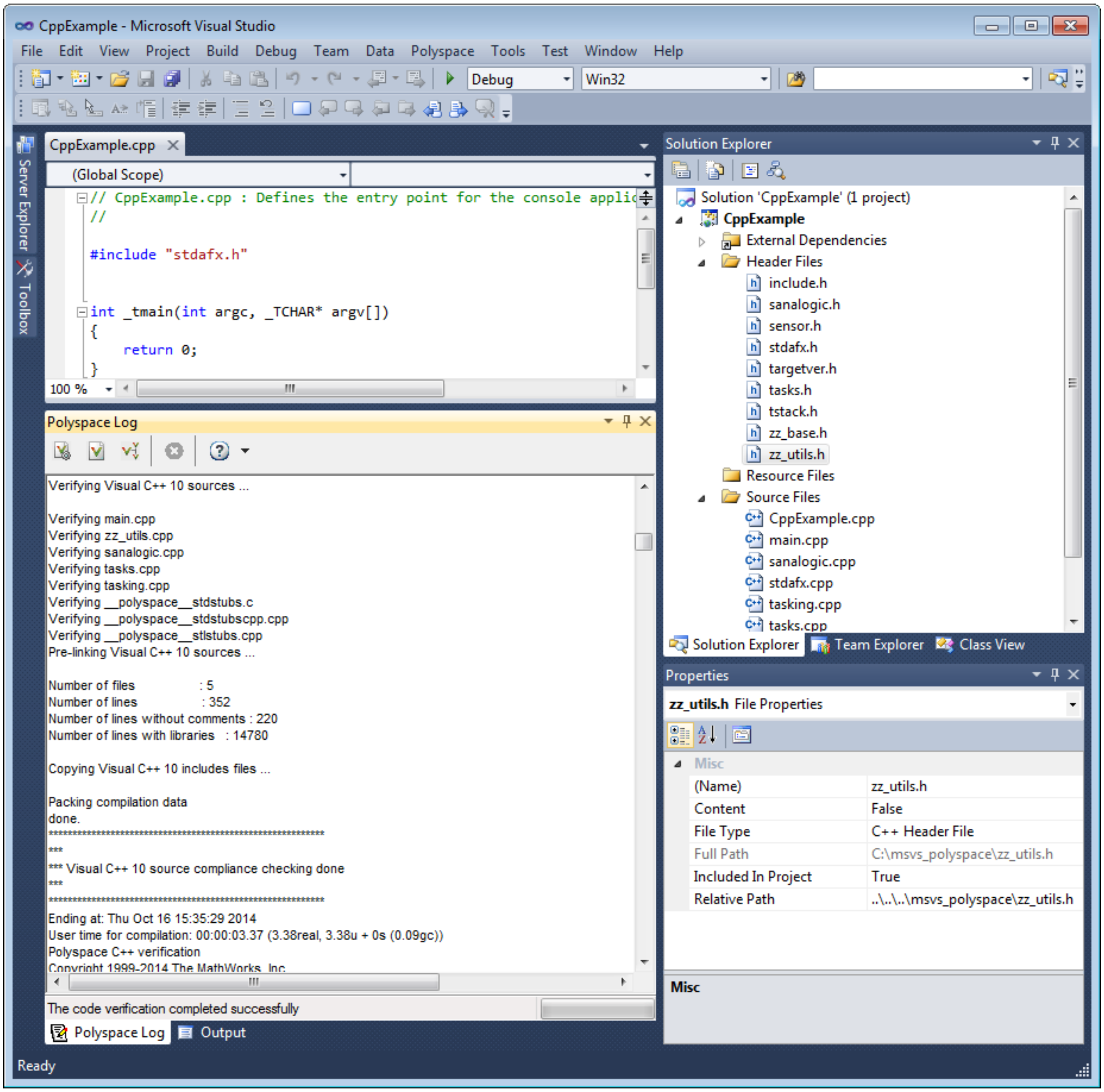
To follow the progress of an analysis, see “Monitor Progress in Visual Studio” on page 15-5

Monitor Progress in Visual Studio

Local Analysis

1 Open the **Polyspace Log** view to follow the progress of your analysis.

  If Polyspace finds compilation issues, the errors are highlighted as links. Click a link to display the file and line that produced the error.



```
// CppExample.cpp : Defines the entry point for the console applic
//

#include "stdafx.h"


int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

```
Verifying Visual C++ 10 sources ...

Verifying main.cpp
Verifying zz_utils.cpp
Verifying sanalogic.cpp
Verifying tasks.cpp
Verifying tasking.cpp
Verifying __polyspace__stdstubs.c
Verifying __polyspace__stdstubscpp.cpp
Verifying __polyspace__stlstubs.cpp
Pre-linking Visual C++ 10 sources ...

Number of files              : 5
Number of lines              : 352
Number of lines without comments : 220
Number of lines with libraries   : 14780

Copying Visual C++ 10 includes files ...

Packing compilation data
done.
************************************************************
***
*** Visual C++ 10 source compliance checking done
***
************************************************************
Ending at: Thu Oct 16 15:35:29 2014
User time for compilation: 00:00:03.37 (3.38real, 3.38u + 0s (0.09gc))
Polyspace C++ verification
Copyright 1999-2014 The MathWorks, Inc.
```

The code verification completed successfully

zz_utils.h File Properties

Misc	
(Name)	zz_utils.h
Content	False
File Type	C++ Header File
Full Path	C:\msvs_polyspace\zz_utils.h
Included In Project	True
Relative Path	..\..\..\msvs_polyspace\zz_utils.h

Misc

2. To stop an analysis, on the **Polyspace Log** toolbar, click **X**.

Remote Analysis

1. Open the **Polyspace Log** view to follow the progress of your analysis.

   If Polyspace finds compilation issues, the errors are highlighted as links. Click a link to display the file and line that produced the error.

   To stop a verification during the compilation phase, on the **Polyspace Log** toolbar, click **X**.

   After compilation, Polyspace sends your analysis to the remote server.
2. Select **Polyspace** > **Job Monitor**.
3. In the Polyspace Job Monitor, right-click your project and select **View Log File**

   To stop a remote analysis after compilation, use the Job Monitor interface.

Related Examples

- “Run Polyspace in Visual Studio” on page 15-2
- “Open Results in Polyspace Environment” on page 16-2

Customize Polyspace Options

In the Easy Settings dialog box in Visual Studio, you specify only a subset of the Polyspace analysis options.

To customize other analysis options:

1. Select the files you wish to analyze.
2. Right-click on your selection and select **Edit Polyspace Configuration** from the context menu.
3. In the Polyspace Bug Finder configuration window, use the different panes to customize your analysis options.

   For more information about specific options, see “Analysis Options for C++”.
4. Save your changes and close the configuration window.

   Next time you run an analysis, Polyspace uses the `*ProjectName*_UserSettings.psprj` settings.

Configuration File and Default Options

Some options are set by default while others are extracted from the Visual Studio project and stored in the associated Polyspace configuration file.

- The following table shows Visual Studio options that are extracted automatically, and their corresponding Polyspace options:

Visual Studio Option	Polyspace Option
`/D <name>`	`-D` *`<name>`*
`/U <name>`	`-U` *`<name>`*
`/MT`	`-D_MT`
`/MTd`	`-D_MT -D_DEBUG`
`/MD`	`-D_MT -D_DLL`
`/MDd`	`-D_MT -D_DLL -D_DEBUG`
`/MLd`	`-D_DEBUG`
`/Zc:wchar_t`	`-wchar-t-is keyword`
`/Zc:forScope`	`-for-loop-index-scope in`
`/FX`	`-support-FX-option-results`
`/Zp[1,2,4,8,16]`	`-pack-alignment-value [1,2,4,8,16]`

- Source and `include` folders (`-I`) are also extracted automatically from the Visual Studio project.
- Default options passed to the kernel depend on the Visual Studio release: `-dialect Visual7.1` (or `-dialect visual8`) `-OS-target Visual -target i386 -desktop`

Bug Finding in Visual Studio

You can apply the bug finding functionality of Polyspace software to code that you develop within the Visual Studio Integrated Development Environment (IDE).

A typical workflow is:

1. Use the Visual Studio editor to create a project and develop code within this project.
2. Set up the Polyspace analysis by configuring analysis options and settings, and then start the analysis.
3. Monitor the analysis.
4. Open the verification results and review in the Polyspace environment.

Before you can verify code in Visual Studio, you must install the Polyspace add-in for Visual Studio. For more information , see “Install Polyspace Add-In for Visual Studio”.

16

Open Results from Microsoft Visual Studio

Open Results in Polyspace Environment

After your analysis finishes running in Visual Studio, open the Polyspace environment to view your results. If you ran a server analysis, download the results before opening the Polyspace environment.

To view your results:

- From the Polyspace Log window, select ✓.
- Select **Polyspace** > **Polyspace**.

  Then, open your results from the Polyspace interface. For instructions, see "Open Results".

Related Examples

- "Review and Comment Results"
- "Run Polyspace in Visual Studio" on page 15-2