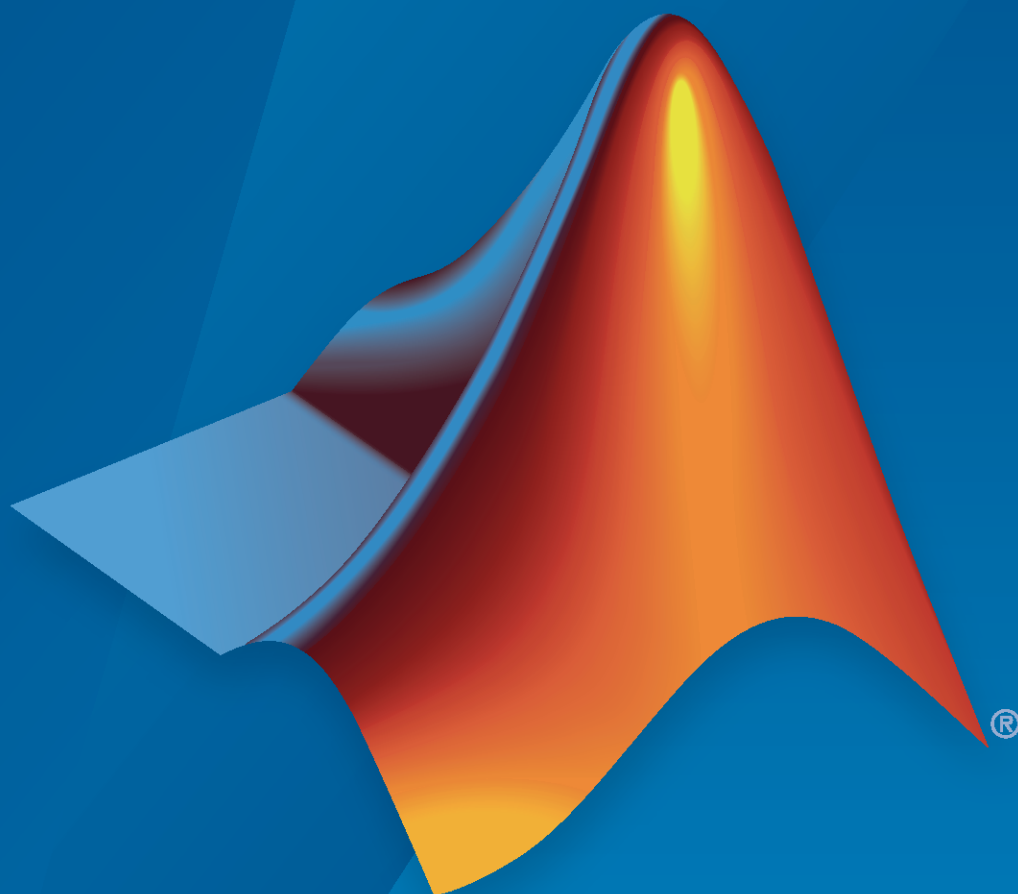


# Polyspace® Bug Finder™

## Getting Started Guide



R2023a



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*Polyspace® Bug Finder™ Getting Started Guide*

© COPYRIGHT 2013–2023 by The MathWorks, Inc.

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

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

### Trademarks

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

### Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## 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)
September 2015	Online Only	Revised for Version 2.0 (Release 2015b)
October 2015	Online Only	Rereleased for Version 1.3.1 (Release 2015aSP1)
March 2016	Online Only	Revised for Version 2.1 (Release 2016a)
September 2016	Online Only	Revised for Version 2.2 (Release 2016b)
March 2017	Online Only	Revised for Version 2.3 (Release 2017a)
September 2017	Online Only	Revised for Version 2.4 (Release 2017b)
March 2018	Online Only	Revised for Version 2.5 (Release 2018a)
September 2018	Online Only	Revised for Version 2.6 (Release 2018b)
March 2019	Online Only	Revised for Version 3.0 (Release 2019a)
September 2019	Online Only	Revised for Version 3.1 (Release 2019b)
March 2020	Online Only	Revised for Version 3.2 (Release 2020a)
September 2020	Online Only	Revised for Version 3.3 (Release 2020b)
March 2021	Online Only	Revised for Version 3.4 (Release 2021a)
September 2021	Online Only	Revised for Polyspace Bug Finder Version 3.5, Polyspace Bug Finder Server Version 3.5, and Polyspace Bug Finder Access Version 3.1 (Release 2021b)
March 2022	Online Only	Revised for Polyspace Bug Finder Version 3.6, Polyspace Bug Finder Server Version 3.6, and Polyspace Access Version 4.0 (Release 2022a)
September 2022	Online Only	Revised for Polyspace Bug Finder Version 3.7, Polyspace Bug Finder Server Version 3.7, and Polyspace Access Version 4.1 (Release 2022b)
March 2023	Online Only	Revised for Polyspace Bug Finder Version 3.8, Polyspace Bug Finder Server Version 3.8, and Polyspace Access Version 4.2 (Release 2023a)



## Introduction

### 1

<b>About This Getting Started Guide</b> .....	<b>1-2</b>
<b>Polyspace Bug Finder Product Description</b> .....	<b>1-3</b>

## Bug Finder Analysis on Desktop

### 2

<b>Run Polyspace Bug Finder on Desktop</b> .....	<b>2-2</b>
Example Files .....	<b>2-2</b>
Run Polyspace in User Interface .....	<b>2-3</b>
Run Polyspace on Windows or Linux Command Line .....	<b>2-4</b>
Run Polyspace in Eclipse .....	<b>2-5</b>
Run Polyspace in MATLAB .....	<b>2-5</b>
<b>Review Polyspace Bug Finder Results in Polyspace User Interface</b> .....	<b>2-7</b>
Example Files .....	<b>2-7</b>
Interpret Results .....	<b>2-7</b>
Address Results Through Bug Fix or Comments .....	<b>2-8</b>
Manage Results .....	<b>2-9</b>

## Bug Finder Analysis on Servers During Continuous Integration

### 3

<b>Quick Start Guide for Polyspace Server and Access Products</b> .....	<b>3-2</b>
Installation .....	<b>3-2</b>
Setting Up Polyspace Analysis .....	<b>3-3</b>
<b>Run Polyspace Bug Finder on Server and Upload Results to Web Interface</b> .....	<b>3-6</b>
Prerequisites .....	<b>3-6</b>
Check Polyspace Installation .....	<b>3-7</b>
Run Bug Finder on Sample Files .....	<b>3-7</b>
Sample Scripts for Bug Finder Analysis on Servers .....	<b>3-9</b>
Specify Sources and Options in Separate Files from Launching Scripts ...	<b>3-9</b>
Complete Workflow .....	<b>3-10</b>

<b>View Assigned Results in Polyspace Access Web Interface</b> .....	<b>3-12</b>
View Assigned Findings by Using the Polyspace Access Project Explorer and Dashboard .....	<b>3-12</b>
<b>Triage and Assign Results in Polyspace Access Web Interface</b> .....	<b>3-14</b>
Navigate the Polyspace Access Web Interface Dashboard .....	<b>3-14</b>
Navigate the Results List, Result Details, and Source Code Panels .....	<b>3-15</b>
Filter Polyspace Access Results .....	<b>3-17</b>
Assign Status and Owner to Results .....	<b>3-18</b>
<b>Send Email Notifications with Polyspace Bug Finder Server Results</b> ...	<b>3-20</b>
Creating E-mail Notifications .....	<b>3-20</b>
Prerequisites .....	<b>3-21</b>
Export Results for E-mail Attachments .....	<b>3-22</b>
Assign Owners and Export Assigned Results .....	<b>3-22</b>

## Offloading Bug Finder Analysis from Desktop to Server

### 4

<b>Send Bug Finder Analysis from Desktop to Locally Hosted Server</b> .....	<b>4-2</b>
Client-Server Workflow for Running Analysis .....	<b>4-2</b>
Prerequisites .....	<b>4-3</b>
Configure and Start Server .....	<b>4-3</b>
Configure Client .....	<b>4-5</b>
Send Analysis from Client to Server .....	<b>4-5</b>

## Bug Finder Analysis in IDEs

### 5

<b>Check Code Quality in IDE Before Submitting</b> .....	<b>5-2</b>
Install Polyspace as You Code Analysis Engine and IDE Extensions .....	<b>5-2</b>
Run Polyspace as You Code and Review Results .....	<b>5-3</b>
Configure Polyspace as You Code IDE Extension .....	<b>5-4</b>
<b>Perform Polyspace as You Code Analysis in Visual Studio Code</b> .....	<b>5-5</b>
<b>Configure Polyspace as You Code in Visual Studio Code</b> .....	<b>5-6</b>
Manually Configure Your Build .....	<b>5-6</b>
Set Automatic Quality Monitoring .....	<b>5-7</b>
Set Analysis on Save .....	<b>5-8</b>
Configure Polyspace Checkers .....	<b>5-9</b>
Analyze Header Files .....	<b>5-10</b>
Set Analysis Script .....	<b>5-11</b>
<b>Run and Review Results in Polyspace as You Code for Visual Studio Code</b> .....	<b>5-13</b>
Run Polyspace as You Code Analysis in Visual Studio Code .....	<b>5-13</b>
View, Fix, or Justify Findings .....	<b>5-13</b>
Justify Individual Findings .....	<b>5-15</b>

Provide Justification Catalog .....	5-16
View Header Findings .....	5-17
Check for Potential Duplicate Code .....	5-18
<b>Configure and Download Baseline with Polyspace as You Code .....</b>	<b>5-20</b>
Configure Baseline .....	5-20
Download Baseline .....	5-21
Show New Findings and Compare Results .....	5-22
<b>Perform Polyspace as You Code Analysis in Eclipse .....</b>	<b>5-24</b>
<b>Configure Polyspace as You Code in Eclipse .....</b>	<b>5-25</b>
Manually Configure Your Build .....	5-25
Automatically Add Files to Quality Monitoring List .....	5-26
Analyze Files Automatically .....	5-27
Configure Polyspace Checkers .....	5-28
View Header File Findings .....	5-29
Configure Analysis Script .....	5-30
<b>Run and Review Results in Polyspace as You Code for Eclipse .....</b>	<b>5-32</b>
Run Polyspace as You Code Analysis in Eclipse .....	5-32
View, Fix, or Justify Findings .....	5-33
Justify Individual Findings .....	5-34
Provide Justification Catalog .....	5-35
View Header Findings .....	5-36
<b>Configure and Download Baseline with Polyspace as You Code in Eclipse</b> .....	<b>5-38</b>
Configure Baseline .....	5-38
Download Baseline .....	5-39
Show New Findings and Compare Results .....	5-39

## Deploy Polyspace Bug Finder

# 6

<b>Polyspace Products and Software Development Workflows .....</b>	<b>6-2</b>
Using Polyspace Products in Software Development .....	6-2
Coordinating Pre-Submit and Post-Submit Usage of Polyspace .....	6-3
Polyspace Products for Ada Code .....	6-4
<b>Differences Between Polyspace Bug Finder and Polyspace Code Prover</b> .....	<b>6-5</b>
How Bug Finder and Code Prover Complement Each Other .....	6-5
<b>Workflow Using Both Polyspace Bug Finder and Polyspace Code Prover</b> .....	<b>6-11</b>





# Introduction

---

- “About This Getting Started Guide” on page 1-2
- “Polyspace Bug Finder Product Description” on page 1-3

## About This Getting Started Guide

This Getting Started Guide covers all Polyspace Bug Finder products:

- Polyspace Bug Finder
- Polyspace Bug Finder Server™
- Polyspace Access™

Depending on how you set up a Bug Finder run, you might be running an analysis from one of these locations:

- **Desktop:** If you are running an analysis and reviewing the results on your desktop, you use Polyspace Bug Finder. To get started, see “Bug Finder Analysis on Desktop”.
- **Server:** If you are running an analysis on a server or reviewing the results from a server run on a web browser, you use:
  - Polyspace Bug Finder Server to run the analysis.
  - Polyspace Access to host the analysis results (for review on a web browser).

To get started, see “Bug Finder Analysis on Servers During Continuous Integration”.

- **IDE:** If you are running an analysis on the current file in your Integration Development Environment (IDE), you use Polyspace as You Code. Polyspace as You Code is a feature available with Polyspace Access. To get started, see “Bug Finder Analysis in IDEs”.

The Bug Finder analysis engine underlies all Bug Finder products. Chapters that do not mention a particular platform typically describe the underlying Bug Finder analysis engine and apply to all three platforms.

# Polyspace Bug Finder Product Description

## Identify software bugs via static analysis

Polyspace Bug Finder identifies run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software. Using static analysis, including semantic analysis, Polyspace Bug Finder analyzes software control, data flow, and interprocedural behavior. By highlighting defects as soon as they are detected, it lets you triage and fix bugs early in the development process.

Polyspace Bug Finder checks compliance with coding rule standards such as MISRA C™, MISRA C++, JSF++, CERT® C, CERT C++, and custom naming conventions. It generates reports consisting of bugs found, code-rule violations, and code quality metrics, including cyclomatic complexity. Polyspace Bug Finder can be used with the Eclipse™ IDE to analyze code on your desktop.

For automatically generated code, Polyspace results can be traced back to Simulink® models and dSPACE® TargetLink® blocks.

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



# Bug Finder Analysis on Desktop

---

- “Run Polyspace Bug Finder on Desktop” on page 2-2
- “Review Polyspace Bug Finder Results in Polyspace User Interface” on page 2-7

## Run Polyspace Bug Finder on Desktop

Polyspace Bug Finder identifies run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software. Using static analysis, including semantic analysis, Bug Finder analyzes control flow, data flow, and interprocedural behavior. By highlighting defects as soon as they are detected, Bug Finder lets you triage and fix bugs early in the development process.

You can run Bug Finder on complete C/C++ projects from the Polyspace user interface, in a supported development environment (IDE) such as Eclipse or using scripts. See:

- “Run Polyspace in User Interface” on page 2-3

If this is your first time using Polyspace, you might want to start from the Polyspace user interface. You can get help from features such as a project setup wizard, assisted configuration and summarized analysis log.

- “Run Polyspace on Windows or Linux Command Line” on page 2-4

Once you set up a project in the Polyspace user interface and complete a few trial runs, you can export the configuration to scripts that you run automatically or on-demand. You can also run a Polyspace analysis directly from the command line in your operating system. You can then save the commands in batch files (Windows) or shell scripts (Linux) for later runs. If you are running Polyspace Server products using continuous integration tools such as Jenkins, you can reuse your scripts from the Polyspace desktop products.

- Run Polyspace in IDEs on page 2-5

Once you are familiar with running Polyspace from the command line, you can create menu items in your IDE that run your scripts and launch a Polyspace analysis in one click. In Eclipse and Eclipse-based IDEs, you can install a Polyspace plugin that does not require any additional setup at all. When you run Polyspace from the Eclipse plugin, the analysis configuration is created directly from your Eclipse project.

Instead of analyzing complete projects, in your IDE, you can analyze just the current file that you are working on using Polyspace as You Code. You can install a Polyspace as You Code extension/plugin in commonly used IDEs such as Visual Studio, Visual Studio Code, or Eclipse. See “Review Polyspace as You Code Results in IDEs”.

- “Run Polyspace in MATLAB” on page 2-5

If you have a MATLAB installation, it is particularly easy to write scripts to run a Polyspace analysis. You get all the benefits of scripting in the MATLAB environment, for instance, automatic help on function syntaxes. After analysis, you can create your own visualization of the results using MATLAB graphics and visualization tools.

### Example Files

To follow the steps in this tutorial, copy the files from *polyspaceroot*\polyspace\examples\cxx\Bug\_Finder\_Example\sources to another folder. Here, *polyspaceroot* is the Polyspace installation folder, for instance, C:\Program Files\Polyspace\R2023a.

## Run Polyspace in User Interface

### Open Polyspace User Interface

Double-click the `polyspace` executable in `polyspaceroot\polyspace\bin`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a`. See also “Installation Folder”.

If you set up a shortcut to Polyspace on your desktop or the **Start** menu in Windows®, double-click the shortcut.

### Add Source Files

To run an analysis, you have to create a new Polyspace project. A Polyspace project points to source and include folders on your file system.

On the left of the **Start Page** pane, click **Start a new project**. Alternatively, select **File > New Project**.

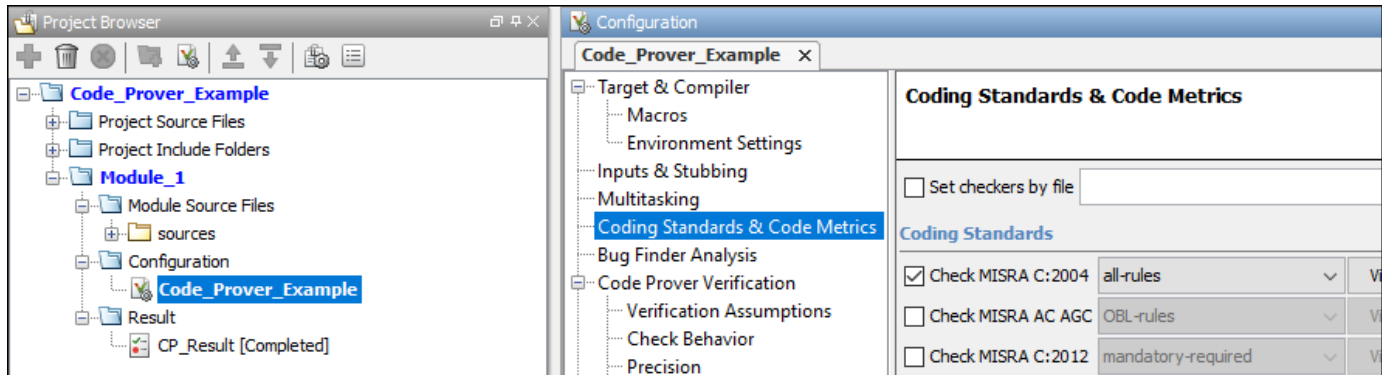
After you provide a project name, on the next screens, add your source and include folders (both folders can be the same). In this tutorial, add the path to the folder in which you saved the source and include files.

After you finish adding your source and include folders, you see a new project on the **Project Browser** pane. Your source folders are copied to the first module in the project. You can right-click a project to add more folders later. If you add folders later, you must explicitly copy them to a module.

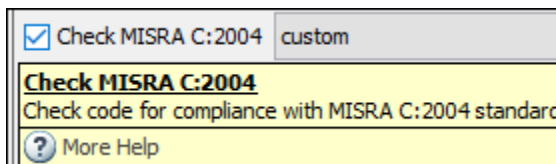
## Configure and Run Polyspace

You can change the default options associated with a Polyspace analysis.

Click the **Configuration** node in your project module. On the **Configuration** pane, change options as needed. For instance, on the **Coding Rules & Code Metrics** node, select **Check MISRA C:2004**.



For more information, see the tooltip on each option. Click the **More help** link for context-sensitive help on the options.



To start analysis, click **Run Bug Finder** in the top toolbar. If the button indicates Code Prover, click the arrow beside the button to switch to Bug Finder.

Follow the progress of analysis on the **Output Summary** window. After the analysis, the results open automatically.

### Additional Information

See:

- “Add Source Files for Analysis in Polyspace Desktop User Interface”
- “Run Analysis in Polyspace Desktop User Interface”

## Run Polyspace on Windows or Linux Command Line

You can run Bug Finder from the Windows or Linux® command line with batch (.bat) files or shell (.sh) scripts.

To run a Bug Finder analysis, use the `polyspace-bug-finder` command.

To save typing the full path to the command, add the path `polyspaceroot\polyspace\bin` to the `Path` environment variable on your operating system. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a`.

Navigate to the folder where you saved the files (using `cd`). Enter the following:



```
polyspace-bug-finder -sources numerical.c,dataflow.c -I . -results-dir .
```

Here, . indicates the current folder. The options used are:

- `-sources`: Specify comma-separated source files.
- `-I`: Specify path to include folder. Use the `-I` flag each time you want to add a separate include folder.
- `-results-dir`: Specify the path to the folder where Polyspace Bug Finder results will be saved.

Note that the results folder is cleaned up and repopulated at each run. To avoid accidental removal of files during the cleanup, instead of using an existing folder that contains other files, specify a dedicated folder for the Polyspace results.

After analysis, the results are saved in the file `ps_results.psbf`. You can open this file from the Polyspace user interface. For instance, enter the following:

```
polyspace ps_results.psbf
```

Instead of specifying comma-separated sources directly on the command line, you can list the sources in a text file (one file per line). Use the option `-sources-list-file` to specify this text file.

### Additional Information

See:

- “Run Polyspace Analysis from Command Line”
- `polyspace-bug-finder`

## Run Polyspace in Eclipse

If you develop code in Eclipse or an Eclipse-based IDE, you can run Code Prover directly from your IDE.

After installing the Eclipse plugin, you can run Polyspace directly on the files in your Eclipse projects.

In the **Project Explorer** pane in Eclipse, select your project. To use Bug Finder for the analysis, select **Polyspace > Bug Finder**. To start the analysis, select **Polyspace > Run** (Ctrl + R).

After analysis, the results open automatically in Eclipse.

### Additional Information

See “Run Polyspace Analysis on Eclipse Projects”.

## Run Polyspace in MATLAB

Before you run Polyspace from MATLAB®, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

To run an analysis, use a `polyspace.Project` object. The object has two properties:

- **Configuration**: Specify the analysis options such as sources, includes, compiler and results folder using this property.

- **Results:** After analysis, read the analysis results to a MATLAB table using this property.

To run the analysis, use the `run` method of this object.

To run Polyspace on the example file `numerical.c` in `polyspaceroot\polyspace\examples\cxx\Bug_Finder_Examples\sources`, enter the following at the MATLAB command prompt.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', ...
    'examples', 'cxx', 'Bug_Finder_Example', 'sources', 'numerical.c')};
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.EnvironmentSettings.IncludeFolders = {fullfile(polyspaceroot, ...
    'polyspace', 'examples', 'cxx', 'Bug_Finder_Example', 'sources')}
proj.Configuration.ResultsDir = fullfile(pwd, 'results');

% Run analysis
bfStatus = proj.run('bugFinder');

% Read results
resObj = proj.Results;
bfSummary = getSummary(resObj, 'defects');
bfResults = getResults(resObj, 'readable');
```

After analysis, the results are saved in the file `ps_results.psbf`. You can open this file from the Polyspace user interface. For instance, enter the following:

```
resultsFile = fullfile(proj.Configuration.ResultsDir, 'ps_results.psbf');
polyspaceBugFinder(resultsFile)
```

### Additional Information

See:

- “Run Polyspace Analysis by Using MATLAB Scripts”
- `polyspace.Project`
- `polyspace.Project.Configuration` Properties

### See Also

### Related Examples

- “Review Polyspace Bug Finder Results in Polyspace User Interface” on page 2-7
- “Run Polyspace Analysis on Code Generated with Embedded Coder”

## Review Polyspace Bug Finder Results in Polyspace User Interface

Polyspace Bug Finder checks C/C++ code for defects, coding rule violations or security vulnerabilities. After you run an analysis, you open the results in the Polyspace user interface (or if you ran the analysis in Eclipse, the results open in Eclipse).

This topic shows how to review some sample results in the Polyspace user interface. The Polyspace user interface is available with the desktop products, Polyspace Bug Finder and Polyspace Code Prover™.

- If you run a single-file analysis in your IDE using Polyspace as You Code, you can review the results directly within your IDE. See “Review Polyspace as You Code Results in IDEs”.
- If you run an analysis using a Polyspace Server product and upload to Polyspace Access, you can review the results in a web browser. See “Review Polyspace Bug Finder Results in Web Browser”.


### Example Files

To follow the steps in this tutorial, run Polyspace using the steps in “Run Polyspace Bug Finder on Desktop” on page 2-2. Alternatively, in the Polyspace user interface, open example results using **Help > Examples > Bug\_Finder\_Example.psprj**. If you have loaded the example results earlier and made some changes, to load a fresh copy, select **Help > Examples > Restore Default Examples**.

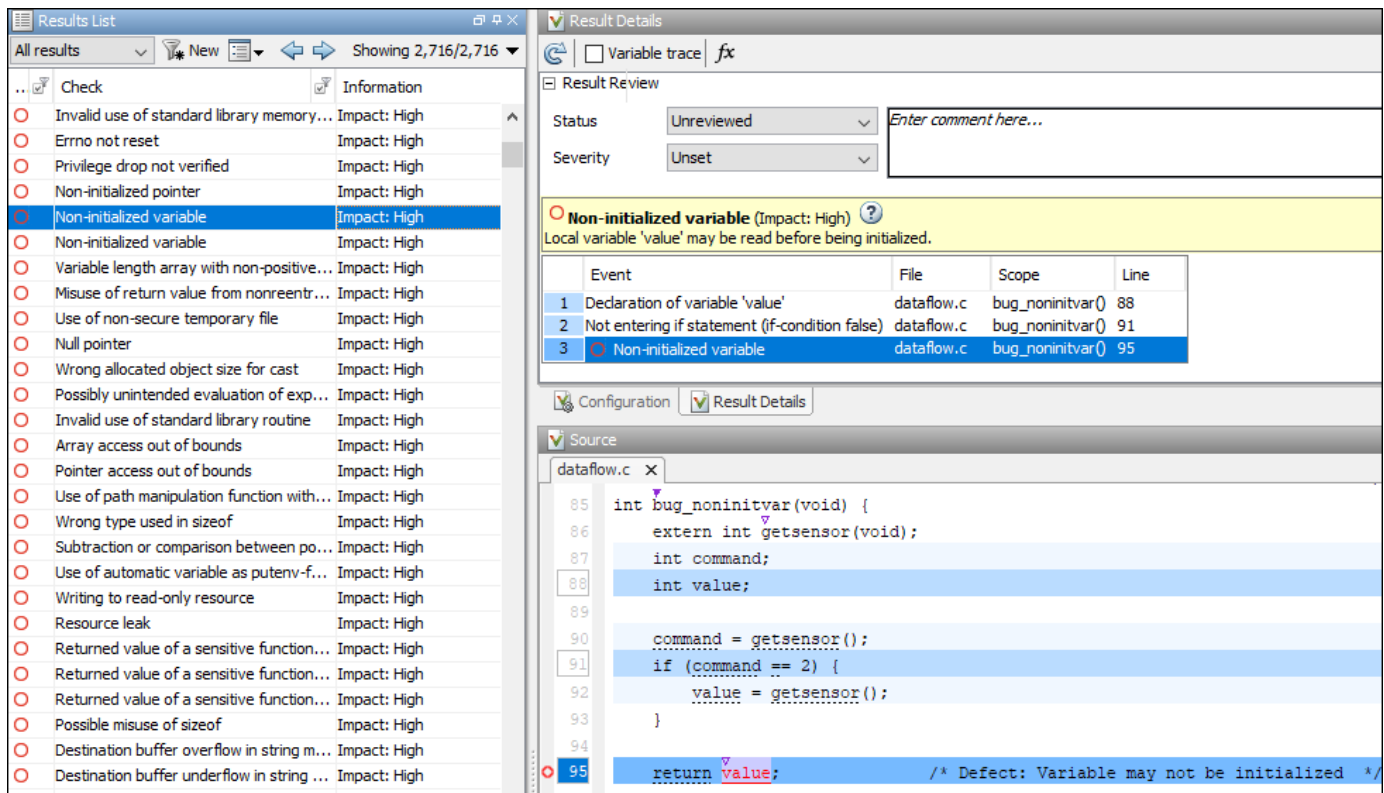
### Interpret Results

Review each Polyspace result. Find the root cause of the issue.

Start from the list of results on the **Results List** pane.

- If the **Results List** pane covers the entire window, select **Window > Reset Layout > Results Review**.
- If you do not see a flat list of results, but instead see them grouped, from the  list, select **None**.

Click the **Check** column header to sort the results alphabetically. Select one of the **Non-initialized variable** results.



See the source code on the **Source** pane and further information about the result on the **Result Details** pane.

The **Result Details** pane also highlights a sequence of events leading to the result. For instance, for the **Non initialized variable** result, you see the following events:

- The variable `value` is declared.
- The `if` statement where `value` gets initialized is skipped.
- The variable `value` is read.

You also see these events highlighted in blue on the source code. Sometimes, these events can be located far apart in the source code. Click an event on the **Result Details** pane to navigate to the corresponding location on the source code.

### Additional Information

See:

- “Interpret Bug Finder Results in Polyspace Desktop User Interface”
- “Complete List of Polyspace Bug Finder Results”

## Address Results Through Bug Fix or Comments

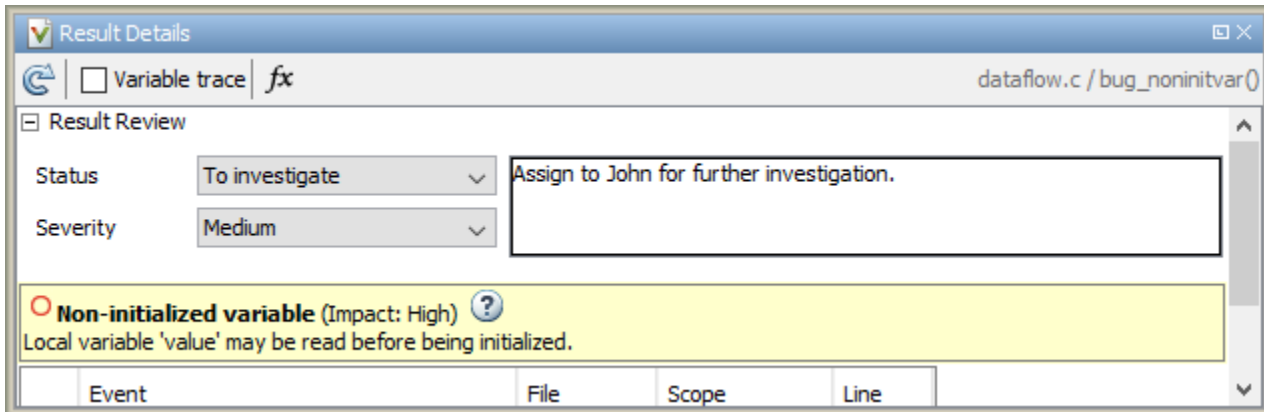
Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add comments to your Polyspace results to fix the code later or to justify the result. You can use the comments to keep track of your review progress.

Right-click the variable `value` on the **Source** pane. Select **Open Editor**. The code opens in a text editor. Fix the issue. For instance, you can initialize `value` during declaration.

```
int value = -1;
```

If you rerun the analysis, you do not see the **Non-initialized variable** defect.

Alternatively, if you do not want to fix the defect immediately, assign a status **To investigate** to the result. Optionally, add comments with further explanation.



If you assign a status **No action planned**, the result does not appear in subsequent runs on the same project.

### Additional Information

See:


- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
- “Annotate Code and Hide Known or Acceptable Results”

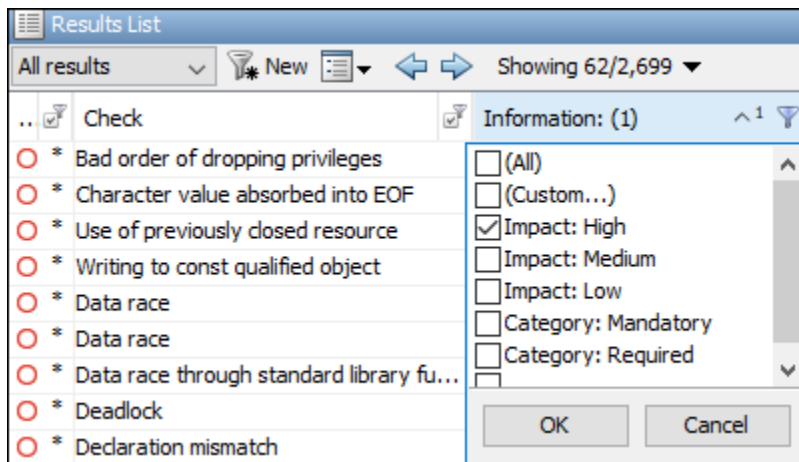
### Manage Results

When you open the results of a Bug Finder analysis, you see a flat list of defects, coding rule violations or other results. To organize your review, you can narrow down the list or group results by file or result type.

For instance, you can:

- Review only high impact defects.

Click the **Information** column header to sort defects by impact. Alternatively, you can filter out results other than high-impact defects. To begin filtering, click the  icon on the column header.



- Review only the new results since the last analysis.

On the **Results List** pane toolbar, click the **New** button.

- Review results in certain files or functions.

On the **Results List** pane toolbar, from the  list, select **File**.

### **Additional Information**

See “Filter and Group Results in Polyspace Desktop User Interface”.

# Bug Finder Analysis on Servers During Continuous Integration

---

## Quick Start Guide for Polyspace Server and Access Products

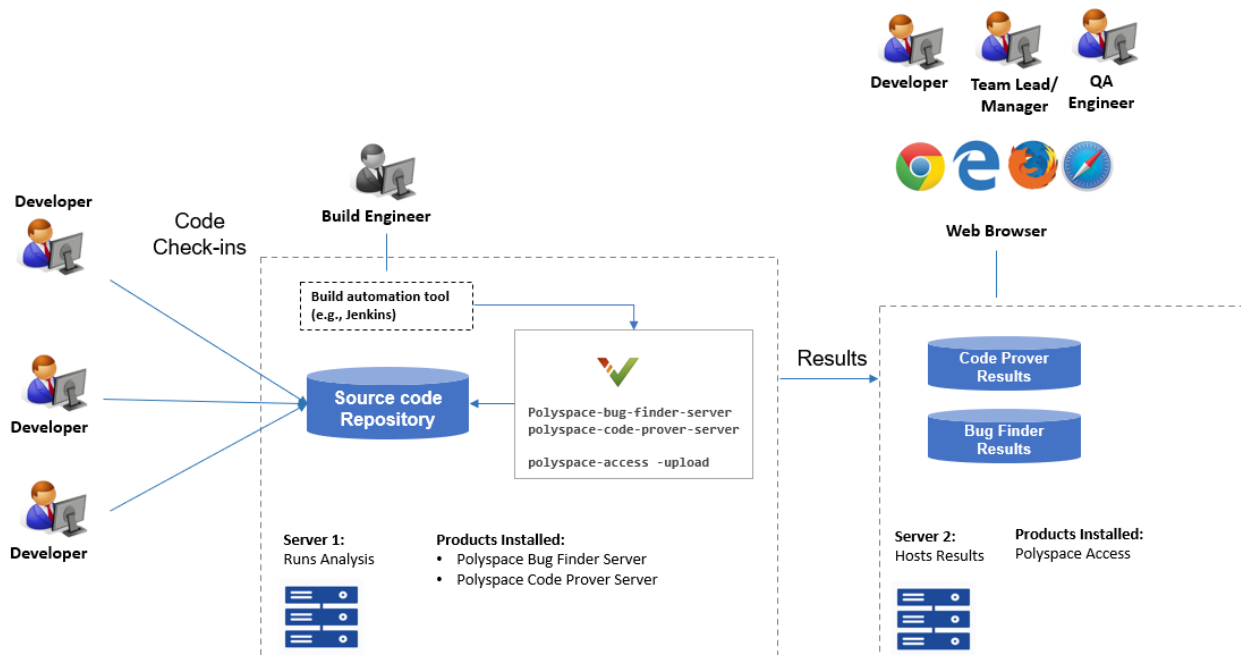
To avoid finding bugs late in the development process, run static analysis by using Polyspace products.

- **Polyspace Bug Finder** checks C/C++ code for bugs, coding standard violations, security vulnerabilities, and other issues.
- **Polyspace Code Prover** performs exhaustive checks for divide by zero, overflow, array access out of bounds, and other common types of run-time errors.

See also “Differences Between Polyspace Bug Finder and Polyspace Code Prover” on page 6-5.

If you run Polyspace checkers regularly as part of continuous integration, you can protect against regressions from new code check-ins. To run Polyspace on a server during continuous integration, use **Polyspace Bug Finder Server** and **Polyspace Code Prover Server**. To host the Polyspace analysis results, use **Polyspace Access**.

A typical workflow looks like this figure.



Note: Depending on the specifications, the same computer can serve as both Server 1 and Server 2.

## Installation

### Prerequisites

Depending on the needs of your project, team or organization, you have decided to obtain a certain number of licenses of Polyspace Server and Polyspace Access products. This guide helps you to install individual instances of these products on a machine.



## Install Polyspace Server

To install Polyspace Server products, download and run the MathWorks installer. Enter a license for the Polyspace Server products (or request a trial license). See also Request a Trial License. The Polyspace Server products are installed in a separate folder from other MathWorks products. See also “Install Polyspace Server and Access Products”.

## Install Polyspace Access

Before installing Polyspace Access, consider the number of users who will potentially review Polyspace results simultaneously. The system requirements depend on the number of simultaneous reviewers. See also “System Requirements for Polyspace Access”.

Polyspace Access consists of several services: a user manager to authenticate user logins, an issue tracker to integrate your bug tracking tool with Polyspace, a database to manage results, a web server to show results, and a gateway to handle communications. The services are deployed in Docker containers. You can start the services from a common interface called the Cluster Admin.

To install Polyspace Access:

- Download the installer as a zip file.
- Unzip the file and start the Cluster Admin. From the Cluster Admin interface, start the various services. See “Install Polyspace Access for Web Reviews”.

After installation, to see uploaded results, you and other reviewers can log in to:

`https://hostName:portNumber/metrics/index.html`

## Install Network License Manager

Both Polyspace Server and Polyspace Access use licenses that require communication with a network license manager for license checkouts.

- To install, configure and start the network license manager for Polyspace Server, see “Administer Network Licenses”.
- To install, configure and start the network license manager for Polyspace Access, see “Manage Polyspace NNU Licenses”.

## Setting Up Polyspace Analysis

### Prerequisites

You or your IT department in your organization must install the required number of Polyspace Server and Polyspace Access instances. This guide helps you to set up a Polyspace analysis as part of continuous integration using a single instance of Polyspace Server and Polyspace Access.

To check that your Polyspace Server and Polyspace Access installations can communicate with each other, see “Check Polyspace Installation”.

### Run Polyspace Server and Upload Results to Polyspace Access

You can run the Polyspace Server products at the command line of your operating system:

- To run the analysis, use the `polyspace-bug-finder-server` and `polyspace-code-prover-server` executables.

- To upload analysis results, use the `polyspace-access` executable. You can also use this executable to export the results from Polyspace Access as text files for archiving or email attachments.

You can run all Polyspace executables from the `polyspace/bin` subfolder of the Polyspace installation folder (for instance, `/usr/local/Polyspace Server/R2023a`, see also “Installation Folder”). To start running Polyspace Server by using sample C source files and sample scripts, see:

- “Run Polyspace Bug Finder on Server and Upload Results to Web Interface” on page 3-6
- “Send Email Notifications with Polyspace Bug Finder Server Results” on page 3-20

You can also preconfigure the Polyspace analysis options from your build command (makefile), and then append a second options file with analysis specifications such as checkers. See “Create Polyspace Analysis Configuration from Build Command (Makefile)”.

If you have an installation of the Polyspace desktop products, you can prepare the analysis configuration in the user interface of the desktop products. You can then generate Polyspace options files to run during continuous integration. See “Configure Polyspace Analysis Options in User Interface and Generate Scripts”.

#### **Include Polyspace Runs in Continuous Integration by Using Tools Such as Jenkins**

Once you have working scripts to run a Polyspace analysis, you can run those scripts at predefined intervals using continuous integration tools such as Jenkins and Bamboo. In Jenkins, you can use a Polyspace plugin to point to your Polyspace installations and send email notifications to developers after the analysis, based on criteria such as new defects.

From within the Jenkins interface, search for and install the Polyspace plugin. For a quick start on using the Jenkins plugin and sample scripts, see the Polyspace plugin GitHub repository. For the full workflow with Jenkins, see “Sample Scripts for Polyspace Analysis with Jenkins”.

#### **Create a Workflow for Result Reviewers**

Depending on tools that you already use, you can set up a convenient workflow for result reviewers. For example:

##### **Reviewers receive alerts for new results and log into Polyspace Access**

- When new results are available, the continuous integration tool alerts a group of users. The email alert contains the Polyspace Access URL of the project where the results are uploaded.
- In the Polyspace Access interface, a reviewer can open this project URL, filter results based on files, and fix the issues or set a status for the results. See also:
  - “Filter and Sort Results in Polyspace Access Web Interface”
  - “Address Results in Polyspace Access Through Bug Fixes or Justifications”

##### **Reviewers get customized email alerts with results in attachment**

- Before upload to Polyspace Access, using the `-set-unassigned-findings` option of the `polyspace-access` executable, the continuous integration (CI) tool assigns owners to new analysis results based on file or component ownership or another criteria.
- After upload, using the `-export` option of the `polyspace-access` executable, the CI tool exports analysis results for each owner to a separate text file. The tool then sends the text file in

an email attachment to the owner. The text file contains results with the corresponding URLs in the Polyspace Access interface.

If you use Jenkins as your CI tool, the Polyspace plugin in Jenkins directly supports this workflow. See “Sample Scripts for Polyspace Analysis with Jenkins”.

- On receiving the email, the owner opens the attached text file, copies the URL of each result to their web browser and reviews the result.

### **Reviewers open tickets from bug tracking tools**

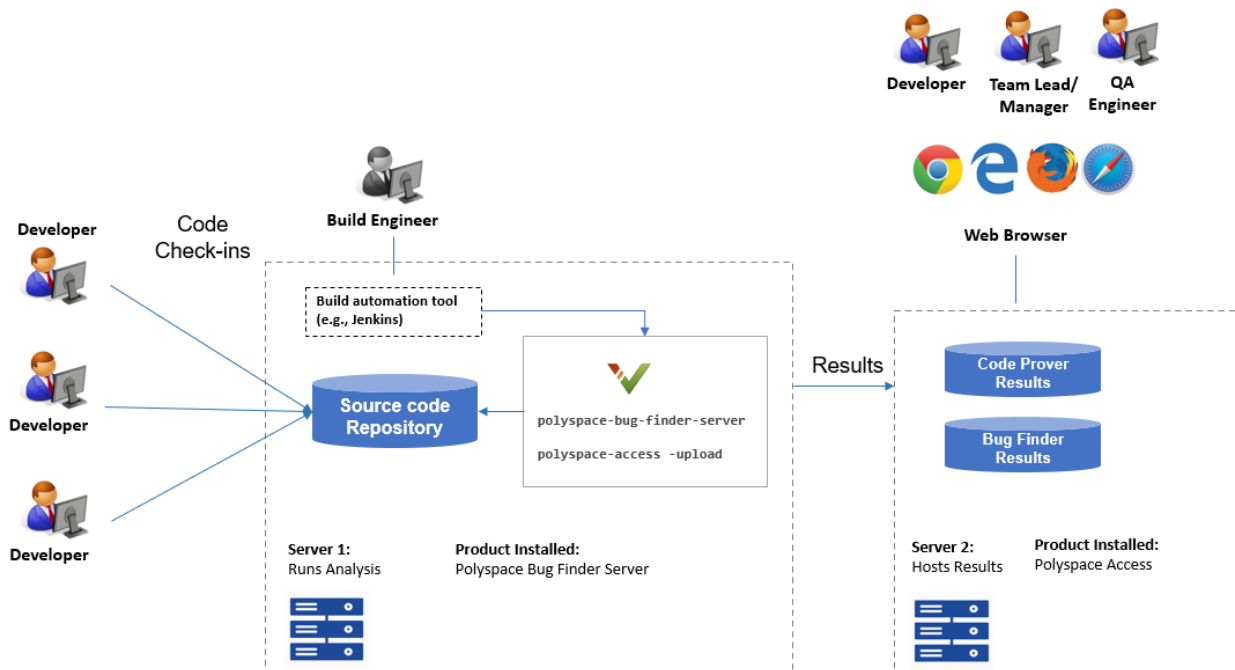
- A reviewer, such as a quality engineer, reviews all new results and creates JIRA tickets for developers. See “Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface”.
- Developers open each JIRA ticket and navigate to the corresponding Polyspace result in the Polyspace Access interface.

## Run Polyspace Bug Finder on Server and Upload Results to Web Interface

Polyspace Bug Finder Server checks C/C++ code for defects and coding standard violations, and then uploads findings to a web interface for code review.

You can run Bug Finder as part of continuous integration. Set up scripts that run a Bug Finder analysis at regular intervals or based on new code submissions. The scripts can upload the analysis results for review in the Polyspace Access web interface and optionally send emails to owners of source files with Polyspace findings. The owners can open the web interface to review only the new findings from their submission, and then fix or justify the issues.

In a typical project or team, Polyspace Bug Finder Server runs periodically on a few testing servers and uploads the results for review. Each developer and quality engineer in the team has a Polyspace Access license to view the results in the web interface for investigation and bug fixing.



Note: Depending on the specifications, the same computer can serve as both Server 1 and Server 2.

### Prerequisites

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

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

See “Install Polyspace Server and Access Products”.

## Check Polyspace Installation

To check if Polyspace Bug Finder Server is installed:

- 1 Open a command window. Navigate to `polyspaceserverroot\polyspace\bin`. Here, `polyspaceserverroot` is the Polyspace Bug Finder Server installation folder, for instance, `C:\Program Files\Polyspace Server\R2023a`. See also “Installation Folder”.
- 2 Enter:

```
polyspace-bug-finder-server -help
```

You should see the list of options allowed for a Bug Finder analysis.

To check if the Polyspace Access web interface is set up for upload:

- 1 Navigate again to `polyspaceserverroot\polyspace\bin`.
- 2 Enter:

```
polyspace-access -host hostName -port portNumber -create-project testProject
```

Here, `hostName` is the name of the server hosting the Polyspace Access web server. For a locally hosted server, use `localhost`. The `portNumber` is the optional port number of the server. If you omit the port number, `9443` is used.

If the setup was complete, a project called `testProject` is created in the Polyspace Access web interface.

You are prompted for your login and password each time that you use the `polyspace-access` command. To avoid entering login information each time, provide the login and an encrypted version of your password with the command. To create an encrypted password, enter:

```
polyspace-access -encrypt-password
```

Enter your login and password. Copy the encrypted password and provide this encrypted password with the `-encrypted-password` option when using the `polyspace-access` command.

- 3 In a web browser, open this URL:

```
https://hostName:portNumber/metrics/index.html
```

Here, `hostName` and `portNumber` are the host name and port number from the previous step.

In the **Project Explorer** pane on the Polyspace Access web interface, you see the newly created project `testProject`.

## Run Bug Finder on Sample Files

To run Bug Finder, in your operating system, open a command window.

- 1 To run a Bug Finder analysis, use the `polyspace-bug-finder-server` command.
- 2 To upload the results to the Polyspace Access web interface, use the `polyspace-access` command.

To avoid typing the full path to the command, add the path `polyspaceserverroot\polyspace\bin` to the Path environment variable on your operating system.

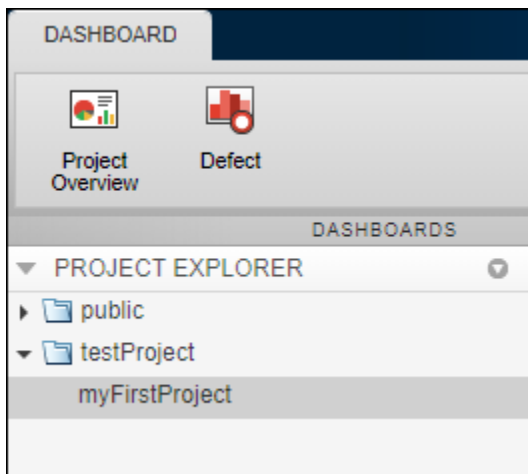
Try out the commands on sample files provided with your Polyspace installation.

- 1 Copy the sample source files from `polyspaceserverroot\polyspace\examples\cxx\Bug_Finder_Example\sources` to another folder where you have write permissions. Navigate to this folder at the command line.
- 2 Enter:

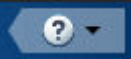
```
polyspace-bug-finder-server -sources numerical.c,dataflow.c -I .
                           -checkers numerical,data_flow -results-dir .
polyspace-access -host hostName -port portNumber
                 -login username -encrypted-password pwd
                 -create-project testProject
polyspace-access -host hostName -port portNumber
                 -login username -encrypted-password pwd
                 -upload . -project myFirstProject -parent-project testProject
```

Here, *username* is your login name and *pwd* is the encrypted password that you created previously. See “Check Polyspace Installation” on page 3-7.

Refresh the Polyspace Access web interface. You see a folder `testProject` on the **Project Explorer** pane. The folder contains one project `myFirstProject`.



To see the results in the project, click **Review**. For more information, see “Review Polyspace Bug

Finder Results in Web Browser”. You can also access the documentation using the  button in the upper right of the Polyspace Access interface.

The analysis options used with the `polyspace-bug-finder-server` command are:

- `-sources`: Specify comma-separated source files.
- `-I`: Specify path to include folder. Use the `-I` flag each time you want to add a separate include folder.
- Find defects (`-checkers`): Specify the defects (bugs) to check for.

- `-results-dir`: Specify the path to the folder where Polyspace Bug Finder results will be saved.

Note that the results folder is cleaned up and repopulated at each run. To avoid accidental removal of files during the cleanup, instead of using an existing folder that contains other files, specify a dedicated folder for the Polyspace results.

For the full list of options available for a Bug Finder analysis, see “Complete List of Polyspace Bug Finder Analysis Engine Options”. To open the Bug Finder documentation in a help browser, enter:

```
polyspace-bug-finder-server -doc
```

## Sample Scripts for Bug Finder Analysis on Servers

To run the analysis, instead of typing the commands at the command line, you can use scripts. The scripts can execute each time that you add or modify source files.

A sample Windows batch file is shown below. Here, the path to the Polyspace installation is specified in the script. To use this script, replace `polyspaceserverroot` with the path to your installation. You must have already generated the encrypted password for use in the scripts. See “Check Polyspace Installation” on page 3-7.

```
echo off
set POLYSPACE_PATH=polyspaceserverroot\polyspace\bin
set LOGIN=-host hostName -port portNumber -login username -encrypted-password pwd
"%POLYSPACE_PATH%\polyspace-bug-finder-server" -sources numerical.c,dataflow.c -I . ^
-checkers numerical,data_flow -results-dir .
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -create-project testProject
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -upload . -project myFirstProject
-parent-project testProject
pause
```

A sample Linux shell script is shown below.

```
POLYSPACE_PATH=polyspaceserverroot/polyspace/bin
LOGIN=-host hostName -port portNumber -login username -encrypted-password pwd
${POLYSPACE_PATH}/polyspace-bug-finder-server -sources numerical.c,dataflow.c -I . \
-checkers numerical,data_flow -results-dir .
${POLYSPACE_PATH}/polyspace-access $LOGIN -create-project testProject
${POLYSPACE_PATH}/polyspace-access $LOGIN -upload . -project myFirstProject
-parent-project testProject
```

## Specify Sources and Options in Separate Files from Launching Scripts

Instead of listing the source files and analysis options within the launching scripts, you can list them in separate text files.

- Specify the text file listing the sources by using the option `-sources-list-file`.
- Specify the text file listing the analysis options by using the option `-options-file`.

By removing the source files and analysis option specifications from the launching scripts, you can modify these specifications as required with new code submissions while leaving the launching script untouched.

Consider the script in the preceding example. You can modify the `polyspace-bug-finder-server` command to use text files with sources and options. Instead of:

```
polyspace-bug-finder-server -sources numerical.c,dataflow.c  
-I . -checkers numerical,data_flow -results-dir .
```

use:

```
polyspace-bug-finder-server -sources numerical.c,dataflow.c  
-I . -checkers numerical,data_flow -results-dir .
```

Here:

- `sources.txt` lists the source files in separate lines:  

```
numerical.c  
dataflow.c
```
- `polyspace_opts.txt` lists the analysis options in separate lines:  

```
-I .  
-checkers numerical,data_flow
```

Typically, your source files are specified in a build command (makefile). Instead of specifying the source files directly, you can trace the build command to create a list of source specifications. See `polyspace-configure`.

## Complete Workflow

In a typical continuous integration workflow, you run a script that executes these steps:

- 1 Extract Polyspace options from your build command.

For instance, if you use makefiles to build your source code, you can extract analysis options from the makefile. The command below first executes `make` and then determines the analysis options from the processes executed.

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

See also:

- `polyspace-configure`
  - “Create Polyspace Analysis Configuration from Build Command (Makefile)”
- 2 Run the analysis with the previously created options file. Append a second options file that contains the remaining options required for the analysis.

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

See “Options Files for Polyspace Analysis”.

- 3 Upload the results to Polyspace Access.

```
polyspace-access login -upload resultsFolder -project projName  
-parent-project parentProjName
```

Here, `login` is the combination of options required to communicate with the web server that is hosting Polyspace Access:



```
-host hostName -port portNumber -login username -encrypted-password pwd
```

*resultsFolder* is the folder containing the Polyspace results. *projName* and *parentProjName* are names of the project and parent folder as they would appear in the Polyspace Access web interface.

- 4 Optionally, send email notifications to developers with new results from their code submission. The email contains attachments with links to the results in the Polyspace Access web interface.

See “Send Email Notifications with Polyspace Bug Finder Server Results” on page 3-20.

See examples of scripts executing these steps in “Sample Scripts for Polyspace Analysis with Jenkins”.

## See Also

polyspace-access | polyspace-bug-finder-server

## More About

- “Send Email Notifications with Polyspace Bug Finder Server Results” on page 3-20
- “Send Bug Finder Analysis from Desktop to Locally Hosted Server” on page 4-2
- “Complete List of Polyspace Bug Finder Analysis Engine Options”

## View Assigned Results in Polyspace Access Web Interface

In a typical collaborative review workflow, results in Polyspace Access are assigned to individuals for further analysis, fixing, or justification. Three common ways to access your assigned results are:

- You receive a direct link to a finding or set of findings.
- You open Polyspace Access and navigate to your assigned results from the **Project Overview** dashboard.
- You open Polyspace Access and navigate to your assigned results using the **Review** button in the taskbar and the **Assigned to me** filter from the filters drop-down list.

If you receive a link to your results, click the link to view your assigned results. Links are commonly sent through email or a bug tracking tool ticket.

### View Assigned Findings by Using the Polyspace Access Project Explorer and Dashboard

Before you start reviewing your assigned results in Polyspace Access, make sure that:

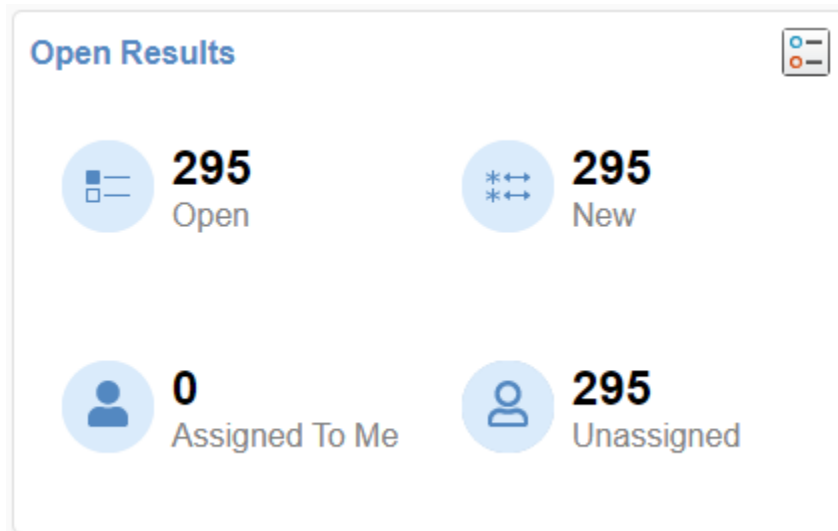
- You have a valid login for Polyspace Access.
- You know the URL for your company's Polyspace Access Web Interface. If you do not know the URL, contact your Polyspace Access administrator.
- The results must be uploaded to Polyspace Access.

To view your assigned findings:

- 1** Log into your company's Polyspace Access Web Interface by using a web browser.
- 2** Open the **Project Explorer** on the left side and select your project run. Projects are listed in a file-folder organization system. A project folder can contain additional sub-folders or individual project runs. After you select your results, the **Project Overview** dashboard opens, displaying your results.

If you select the folder, the dashboard shows an aggregate of the statistics for all the project runs in that folder.

- 3** Click **Assigned To Me** on the Summary card of the **Project Overview** dashboard. The **REVIEW** window opens and shows the **Results List** filtered to all the results that are assigned to you. If you select a folder instead of a project run, **Assigned To Me** is not clickable.



You can reassign your results to another user if needed. See “Triage and Assign Results in Polyspace Access Web Interface” on page 3-14. At this point, all results assigned to you are displayed and you can begin investigating your results.

## See Also

### More About

- “Create Custom Filter Groups in Polyspace Access Web Interface”
- “Review Polyspace Bug Finder Results in Web Browser”
- “Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface”

## Triage and Assign Results in Polyspace Access Web Interface

Polyspace Access offers a centralized database where you can store Polyspace analysis results for sharing with your team and performing collaborative reviews. Once analysis results are uploaded to Polyspace Access, a common next step is to filter and assign results to team members. Use the Polyspace Access dashboards and links to access results. Use filters to review and sort the results that you want to assign.

- 1 Log into the Polyspace Access web interface by using a web browser.
- 2 Open the **Project Explorer** on the left side and select your project run. Projects are listed in a file-folder organization system. A project folder can contain additional sub-folders or individual project runs. You can use the filter at the top of the **Project Explorer** to search for uploaded results. After you select your results, the **Project Overview** dashboard opens, displaying your results.

If you select a folder, the dashboard shows an aggregate of the statistics for all the project runs in that folder.

### Navigate the Polyspace Access Web Interface Dashboard

After you select your project run in the **Project Explorer**, the **Project Overview** dashboard opens for those results. The **Project Overview** dashboard shows a snapshot of the project including:

- What findings currently exist.
- The type and status of the findings.
- Tracking of open findings over time.

The dashboard is split into multiple sections:

- **Summary**

The **Summary** is the main section of the **Project Overview** dashboard and shows a snapshot of the project. This section contains cards showing **Open Issues**, **Code Metrics**, **Quality Objectives**, **Defects** (Bug Finder only), **Run-time Checks** (Code Prover only), and **Coding Standards**.

- **Trends**

The **Trends** section uses a graph to show the number of open findings over time. When you select a project run, the graph shows the open findings trend over time, starting from the first run uploaded to the project up to the currently selected run. Each dot on the trend line represents a project run.

- **Details**

The **Details** section enables users to take a closer look at the project in a table. This table shows the total number of coding standards violations and their status. The status and number of defects is shown (Bug Finder only). The status and number of global variables and red, gray, orange, and green checks is shown (Code Prover only). Click any table entry to view the corresponding results in the **Results List**.

#### Summary Section Overview

The Summary section contains the cards listed in this table:

Card	Description
<b>Open Issues</b>	Shows the total number of open issues, new issues compared to the previous run, the number of open issues that are assigned to the current user, and the total number of unassigned issues.  Click any of these links to view the corresponding results in the <b>Results List</b> .
<b>Code Metrics</b>	Shows the total number of subprojects, number of files, number of lines without comments, and the biggest cyclomatic complexity value of the code. Click the "Code Metrics" link to open the <b>Code Metrics</b> dashboard in a new tab. See "Code Metrics Dashboard in Polyspace Access Web Interface"
<b>Quality Objectives</b>	Shows the completion percentage of all quality objectives and the remaining quality objectives as defined by the current threshold. A label next to the percentage bar shows the analysis status. For example, the label reads <b>Incomplete</b> if checkers required for the selected threshold were not activated in the analysis.  Click the "Quality Objectives" link to open the <b>Quality Objectives</b> dashboard in a new tab. You can create user-defined thresholds for quality objectives. See "Quality Objectives Dashboard in Polyspace Access"
<b>Defects</b> (Bug Finder only)	Shows the current number of open defects along with their status. Click the "Defects" link to open the <b>Defects</b> dashboard in a new tab. This dashboard shows a more detailed breakdown of all open defects and provides the ability to view defects by category or by file. See "Defects"
<b>Run-time Checks</b> (Code Prover Only)	Shows the current number of red, orange, gray, and green run-time checks. Click the "Run-time Checks" link to open the <b>Run-time Checks</b> dashboard in a new tab. This dashboard shows a more detailed breakdown of open run-time checks and provides the ability to view run-time checks by category or by file. See "Run-Time Checks" (Polyspace Code Prover)
<b>Coding Standards</b>	Shows the current number of open coding standard issues and their status. Click the "Coding Standards" link to open dashboards for the coding rules in new tabs. These dashboards can include the <b>Custom Rules</b> dashboard, the <b>Guidelines</b> dashboard, and the dashboards for whichever coding standards are activated for the project such as MISRA C:2012 or SEI CERT C. See "Coding Standards"  The different coding standards dashboards enable you to view a more detailed breakdown of all open coding standard issues including the ability to view coding standard issues by category or by file.

Clicking any link within the tables takes you to the **REVIEW** page with the relevant filters applied.

## Navigate the Results List, Result Details, and Source Code Panels

In many cases, clicking a link on the **Project Overview** dashboard opens the **REVIEW** page. The **REVIEW** page is separated into three major panes:

- **Results List**

- **Result Details**
- **Source Code**

To view additional panes available in the Polyspace user interface, including **Review History** and **Call Hierarchy**, on the toolbar, click **Window** and select a pane. See also “Interpret Results”.

#### **Results List**

The **Results List** contains all the results matching the filters that are set. No other issues are displayed unless you remove these filters. Click the pink eraser icon next to the filters to remove all filters. Place your cursor over an individual filter to open the option to remove the filter.

The **Results List** is organized in a table format. You can sort each column by clicking the column title. You can further filter results at an item level. For the item that you want to filter, right-click the row of the item in the column you want to filter by. This shows the options to filter out or show only the value in the cell. You can also set the **Show only** and **Filter out** values in the **Filters** section on the toolbar. See also “Results List in Polyspace Access Web Interface”

#### **Result Details**

**Result Details** shows detailed information about individual results, including additional information about the result, links to relevant documentation, and review information such as status, severity, and comments. Select a result in the **Results List** to display the result information in the **Result Details** pane. See “Result Details in Polyspace Access Web Interface”

When applicable, the trace of events shows the events that lead to the error. Click the event to highlight the relevant line of code in the **Source Code** pane.

You can also create a bug tracking ticket and assign an owner to a result. See “Assign Status and Owner to Results” on page 3-18

#### **Source Code**

The **Source Code** pane shows the location of the result in the source code. You cannot make edits in the **Source Code** pane. Select a result in the **Results List** to see it in the **Source Code** pane. Right-click in the **Source Code** pane to:

- Quickly navigate to a line in the file.
- Search for all references of a variable.
- Copy the file path to your clipboard.
- Expand or collapse macros.

If multiple results are at the same location in the code, right-click the relevant code to select one of the results to focus on with the **Select Results** option. See “Source Code in Polyspace Access Web Interface”

```

272 void bug_declmismatch() {
273     extern bigstruct_diff
274     S_for_programming; /* Defect: Struct has different definition than in previous definitio
275     read_pstruct_diff(&S_for_programming); /* Defect: Struct has different definition than in previous definitio
276     file) */
277     read_pstruct_diff(&S_for_programming); /* Defect: Struct has different definition than in previous definitio
278 }
279
280 void corrected_declmismatch() {
281     extern bigstruct S_for_programming; /* Defect: Struct has different definition than in previous definitio
282     Using same struct definition */
283     read_pstruct(&S_for_programming);

```

## Filter Polyspace Access Results

The toolstrip displays several additional options for navigation and filtering.

### Custom Filters

Apply and create custom filters. See also “Create Custom Filter Groups in Polyspace Access Web Interface”.

### Family Filters

Quickly apply filters by result type. For instance, clicking **Defects** filters to show only defect type results. Clicking the arrow next to the **Defects** filter specifies viewing high, medium, or low defects. Similarly, Run-time Checks, Coding Standards, and Code Metrics enable further narrowing the scope of your review with additional options in the drop down list.

### Filters

The Filters section contains quick filters as listed in this table:

Filter	Value
Workflow	<ul style="list-style-type: none"> <li>• <b>Open</b> — Findings with the status 'Unreviewed', 'To fix', 'To investigate', or 'Other' that need to be addressed with a fix or a justification.</li> <li>• <b>To Do</b> — Findings with the status 'Unreviewed' that need to be addressed with a fix or a justification.</li> <li>• <b>In Progress</b> — Findings with the status 'To fix', 'To investigate', 'Other' that need to be addressed with a fix or a justification.</li> <li>• <b>Done</b> - Findings with the status 'Justified', 'Not a defect', or 'No action planned'.</li> <li>• <b>Annotated</b> - Findings with a status, severity, or comment assigned from the source code.</li> </ul> <p><b>Note</b> Green run-time checks, green shared variables, non-shared variables, and code metrics do not need to be addressed or justified. These findings do not count toward the number of findings that are <b>To Do</b>, <b>In Progress</b>, and <b>Done</b>.</p>

Filter	Value
Resolution	<ul style="list-style-type: none"> <li>• <b>New</b> — Findings discovered in current run.</li> <li>• <b>Unresolved</b> — Baseline findings still open in current run.</li> <li>• <b>Resolved</b> — Baseline findings fixed or done in current run.</li> <li>• <b>Fixed</b> — Baseline findings no longer present in current run.</li> </ul> <p>See also “Compare Results in Polyspace Access Project to Previous Runs and View Trends”</p>
Assignee	<ul style="list-style-type: none"> <li>• <b>Unassigned</b></li> <li>• <b>Assigned To Me</b></li> </ul>
Status	<ul style="list-style-type: none"> <li>• <b>Unreviewed</b></li> <li>• <b>To Investigate</b></li> <li>• <b>To Fix</b></li> <li>• <b>Justified</b></li> <li>• <b>No Action Planned</b></li> <li>• <b>Not A Defect</b></li> <li>• <b>Other</b></li> </ul>
Severity	<ul style="list-style-type: none"> <li>• <b>High Severity</b></li> <li>• <b>Medium Severity</b></li> <li>• <b>Low Severity</b></li> </ul> <p>See also “Classification of Defects by Impact”</p>
Software Quality Objectives	<p>Individual filters for <b>SQO1</b> through <b>Exhaustive</b>.</p> <p>See also “Evaluate Polyspace Bug Finder Results Against Bug Finder Quality Objectives”</p>

Use the **Show only** filter to show results associated with the keyword, file name, or comment in the **Show only** field. Use **Filter out** to remove results associated with the keyword, file name, or comment in the **Filter out** field.

You can apply the **Show only** and **Filter out** filters by right-clicking the **Results List** table. Each column allows for the filtering of different parameters. Right-click in the cell containing the phrase that you want to filter and select **Show only** or **Filter out** to apply the filter.

## Assign Status and Owner to Results



You can set up email alerts so that component owners get notified when Polyspace results appear in their components. See “Send Email Notifications with Polyspace Bug Finder Server Results” on page 3-20

To assign a user to a result, select the result that you want to assign from the **Results List**. In the **Result Details** pane, use the **Assigned to** drop-down list to select the user you want to assign the results to. Alternatively, begin typing the user name and select them from the autocomplete list. To unassign a user, click the x icon to the right of the **Assigned to** drop-down list.



To assign a status, severity, or comment, in the **Result Details** pane, choose a **Status** and **Severity** from the drop-down lists. Comments are entered in the text field to the right of the drop down.

To select multiple results, hold the **Ctrl** key and click each result. If you want to select a group of results, click the first result, then hold the **Shift** key and click the last result to select all results.

If your bug tracking tool is integrated with Polyspace Access, you can use the **Ticket** section to create a ticket based on the result. Click  to create a ticket or click  to link an existing ticket. See “Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface”

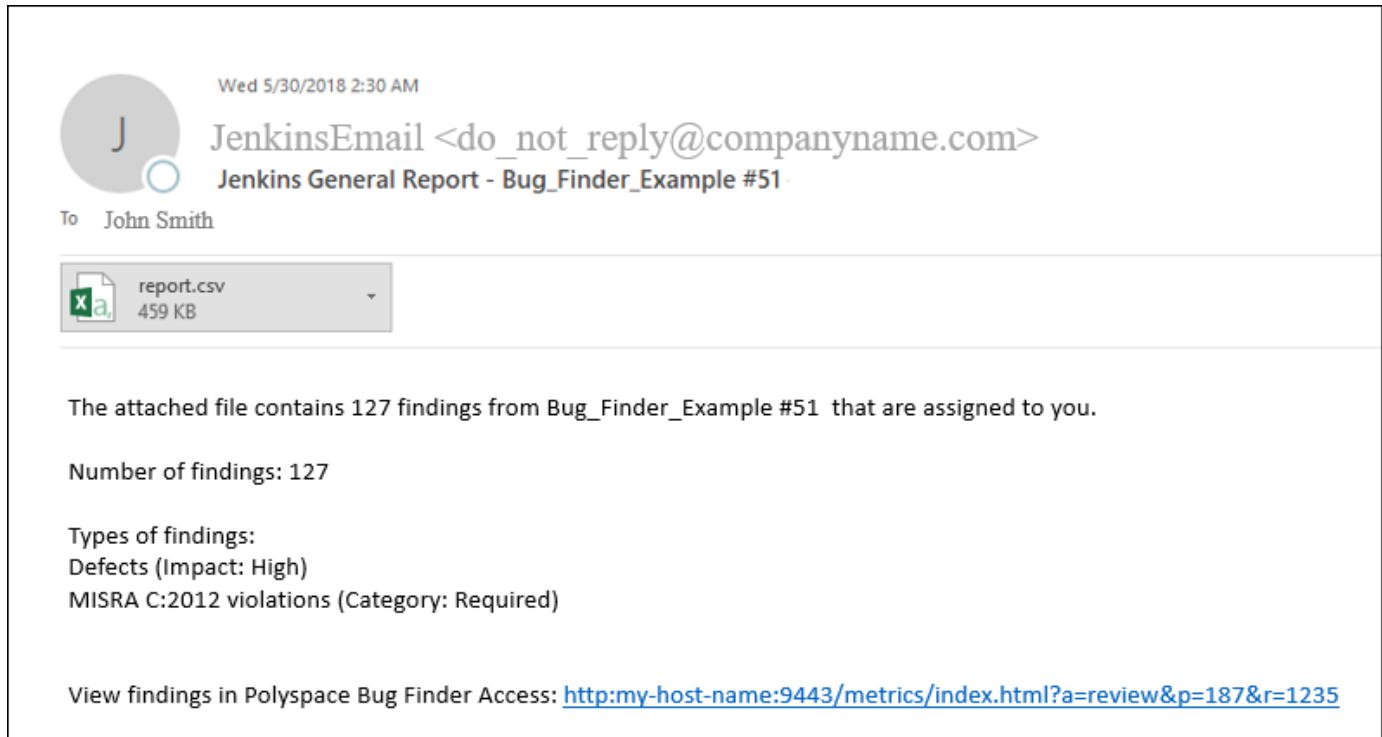
## See Also

### More About

- “Create Custom Filter Groups in Polyspace Access Web Interface”
- “Review Polyspace Bug Finder Results in Web Browser”
- “Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface”

## Send Email Notifications with Polyspace Bug Finder Server Results

If you run a Polyspace analysis as part of continuous integration, each new code submission produces new results. You not only see new results in components that were modified but also in components that depended on the modified components. You can set up e-mail alerts so that component owners get notified when new Polyspace results appear in their components.



### Creating E-mail Notifications

To create e-mail notifications:

- 1 Export new analysis results to a tab-delimited text file (.tsv format). For each result, the file contains links to open the result in the Polyspace Access web interface.

Apply filters to export specific types of results, for instance, defects with high impact. If required, you can also apply additional filters to the exported files using search and replace utilities. See “Export Results for E-mail Attachments” on page 3-22.

- 2 Send an email with the results file in attachment.

For instance, if you use an e-mail plugin in Jenkins, you can create a post-build step to send an e-mail after the analysis is complete.

If you use the Polyspace plugin in Jenkins, you can use Polyspace helper utilities for the entire e-mail notification process. See “Sample Scripts for Polyspace Analysis with Jenkins”.

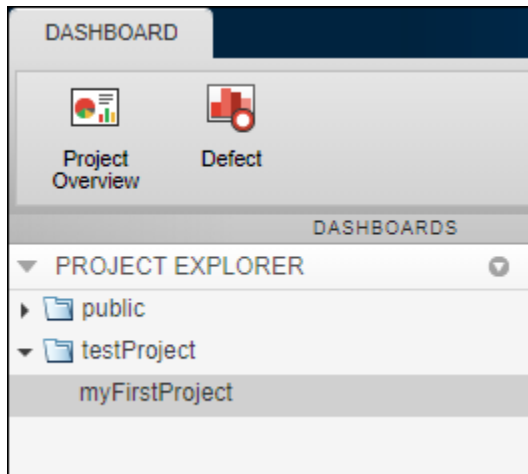
Alternatively, results can be directly assigned to owners based on their file paths. You can set up email notifications that exports a separate results file per owner and sends an email to each owner

with the corresponding results file in attachment. See “Assign Owners and Export Assigned Results” on page 3-22.

## Prerequisites

To run this tutorial:

- You must have uploaded some result to the Polyspace Access web server. If you complete the tutorial “Run Polyspace Bug Finder on Server and Upload Results to Web Interface” on page 3-6, you should see a folder `testProject` on the **Project Explorer** pane of the Polyspace Access web interface. The folder contains one project `myFirstProject`.



To see the results in the project, with `myFirstProject` selected, click the **Review** button. You see a list of defects. The **Information** column shows the impact of the defects. In this tutorial, only high-impact defects will be exported for e-mail attachments.

- You must be able to interact with the Polyspace Access interface from the command line. For instance, navigate to `polyspaceserverroot\polyspace\bin` and enter:

```
polyspace-access login -list-project
```

Here, `polyspaceserverroot` is the Polyspace Bug Finder Server installation folder, for instance, `C:\Program Files\Polyspace Server\R2023a`. The variable `login` refers to the following combination of options. You provide these options with every use of the `polyspace-access` command.

```
-host hostName -port portNumber -login username -encrypted-password pwd
```

Here, `hostName` is the name of the Polyspace Access web server. For a locally hosted server, use `localhost`. `portNumber` is the optional port number of the server. If you omit the port number, 9443 is used. `username` and `pwd` refer to the login and an encrypted version of your password. To create an encrypted password, enter:

```
polyspace-access -encrypt-password
```

Copy the encrypted password and provide this password with later uses of the `polyspace-access` command.

## Export Results for E-mail Attachments

You can export all results in a project or only certain types of results.

Open a command window. Navigate to the folder where you want to export the results.

- To export all results, enter the following:

```
polyspace-access login -export testProject/myFirstProject -output .\result.txt
```

- To export only defects with high impact, enter the following:

```
polyspace-access login -export testProject/myFirstProject -defects High  
-output .\result_high_impact.txt
```

Open each text file in a spreadsheet viewing utility such as Microsoft® Excel®. In the first file, you see all defects but in the second file, you only see the defects with high impact. Instead of `-defects High`, you can apply other filters. For instance:

- To see only new defects compared to the previous analysis of the same project, use the option `-new-findings`.
- To apply a more fine-grained set of filters, you can use software quality objectives (SQOs). The software quality objectives are specified through a progressively stricter set of SQO levels, numbered from 1 to 6. You can customize the requirements of each level in the Polyspace Access web interface, and then use the option `-open-findings-for-sqo` with the level number to export only those results that must be reviewed to meet the requirements. See also “Evaluate Polyspace Bug Finder Results Against Bug Finder Quality Objectives”.

To see all filtering options, enter:

```
polyspace-access -h -export
```

You can configure your e-mail utility to send these exported files in attachment.

If required, you can also apply additional filters to the exported files using search and replace utilities. For instance, use search and replace utilities on the results file to include results only from specific files and functions. In Linux, you can use `grep` and `sed` to retain only results in specific files.

Instead of exporting to text files, you can also generate reports in PDF or Word using predefined report templates. For more information, see `polyspace-report-generator`.

## Assign Owners and Export Assigned Results

You can assign owners to results in specific files or folders. You can then export one result file per owner and send an email to each owner with the corresponding file in attachment.

You can assign owners in the Polyspace Access web interface or at the command line.

In this tutorial, assign all results in the file `numerical.c` to `jsmith` and all results in the file `dataflow.c` to `jboyd`.

```
polyspace-access login
  -set-unassigned-findings testProject/myFirstProject
  -owner jsmith -source-contains numerical.c
polyspace-access login
  -set-unassigned-findings testProject/myFirstProject
  -owner jboyd -source-contains dataflow.c
```

After assignment, export one results file per owner.

```
polyspace-access login
  -export testProject/myFirstProject -output .\results.txt -output-per-owner
```

These files contain the exported results:

- `results.txt` contains all results.
- `results_jsmith.txt` and `results_jboyd.txt` contains results assigned to `jsmith` and `jboyd` respectively.
- `results.txt.owners.list` contains the list of owners, in this case:

```
jsmith
jboyd
```

Before assigning owners to results, use the option `-dryrun` to perform a dry run of the assignments. Without performing the assignment, the option shows the files with results that are assigned and the owner that the results are assigned to.

## See Also

`polyspace-access`



# Offloading Bug Finder Analysis from Desktop to Server

---

# Send Bug Finder Analysis from Desktop to Locally Hosted Server

You can perform a Polyspace analysis locally on your desktop or offload the analysis to one or more dedicated servers. This topic shows a simple server-client configuration for offloading the Polyspace analysis. In this configuration, the same computer acts as a client that submits a Polyspace analysis and a server that runs the analysis.

You can extend this tutorial to more complex configurations. For full setup and workflow instructions, see related links below.

---

**Note** The versions of Polyspace on the client and server machines must match.

---

## Client-Server Workflow for Running Analysis

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

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

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

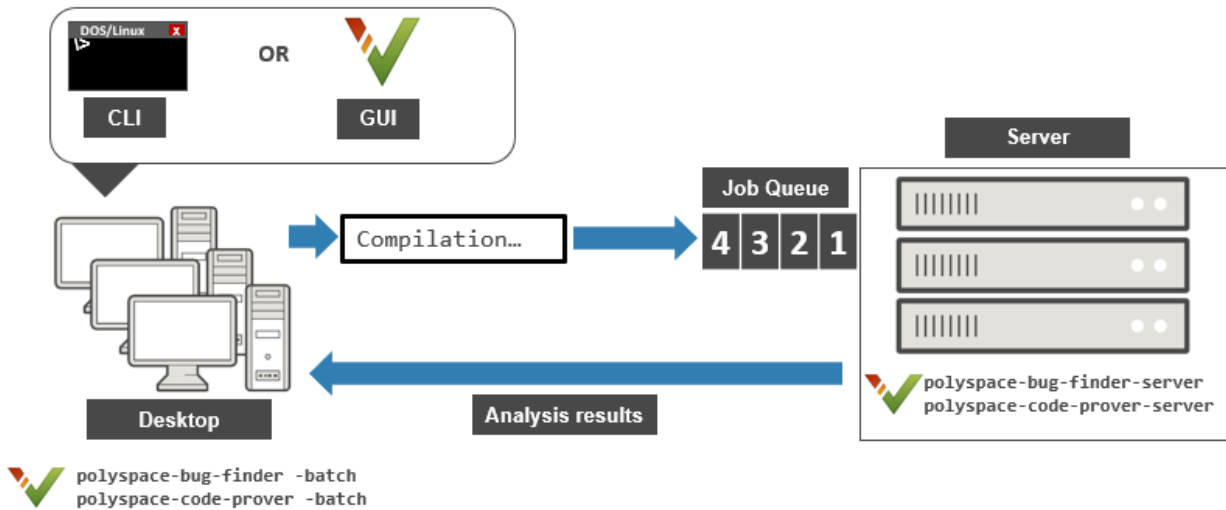
- 2 Head node:** The server consists of a head node and several worker nodes. The head node uses a job scheduler to manage submissions from multiple client desktops. The jobs are then distributed to the worker nodes as they become available.

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

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

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





See also “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

## Prerequisites

This tutorial uses the same computer as client and server. You must install the following on the computer:

- Client-side product: Polyspace Bug Finder
- Server-side products: MATLAB Parallel Server and Polyspace Bug Finder Server

For more information, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

You must know the host name of your computer. For instance, in Windows, open a command-line terminal and enter:

```
hostname
```

## Configure and Start Server

### Stop Previous Services

If you started the services of MATLAB Parallel Server (mjs services) previously, make sure that you have stopped all services. In particular, you might have to:

- Check your temporary folder, for instance, `C:\Windows\Temp` in Windows, and remove the MDCE folder if it exists.
- Stop all services explicitly. You do not require this step in Linux.

Open a command-line terminal. Navigate to `matlabroot\toolbox\parallel\bin` (using `cd`) and enter the following:

```
mjs uninstall -clean
```

Here, *matlabroot* is the MATLAB Parallel Server installation folder, for instance, C:\Program Files\MATLAB\R2023a.

If this is the first time you are starting the services, you do not have to do these steps.

### Configure mjs Service Settings

Before starting services, you have to configure the *mjs* service settings. Navigate to *matlabroot* \toolbox\parallel\bin, where *matlabroot* is the MATLAB Parallel Server installation folder, for instance, C:\Program Files\MATLAB\R2023a. Modify these two files. To edit and save these files, you have to open your editor in administrator mode.

- *mjs\_def.bat* (Windows) or *mjs\_def.sh* (Linux)

Read the instructions in the file and uncomment the lines as needed. At a minimum, you might have to uncomment these lines:

- Hostname:

```
REM set HOSTNAME=myHostName
```

in Windows or

```
#HOSTNAME=`hostname -f`
```

in Linux. Remove the REM or # and explicitly specify your computer host name.

- Security level:

```
REM set SECURITY_LEVEL=
```

in Windows or

```
#SECURITY_LEVEL=""
```

in Linux. Remove the REM or # and explicitly specify a security level.

Otherwise, you might see an error later when starting the job scheduler.

- *mjs\_polyspace.conf*

Modify and uncomment the line that refers to a Polyspace server product root. The line should refer to the release number and root folder of your Polyspace server product installation. For instance, if the R2023a release of Polyspace Bug Finder Server is installed in the root folder C:\Program Files\Polyspace Server\R2023a, modify the line to:

```
POLYSPACE_SERVER_ROOT=C:\Program Files\Polyspace Server\R2023a
```

Otherwise, the MATLAB Parallel Server installation cannot locate the Polyspace Bug Finder Server installation to run the analysis.

### Start Services

Start the *mjs* services and assign the current computer as both the head node and a worker node.

Navigate to `matlabroot\toolbox\parallel\bin`, where `matlabroot` is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2023a`. Run these commands (directly at the command line or using scripts):

```
mjs install
mjs start
startjobmanager -name JobScheduler -remotehost hostname -v
startworker -jobmanagerhost hostname -jobmanager JobScheduler
               -remotehost hostname -v
```

Here, `hostname` is the host name of your computer. This is the host name that you specified in the file `mjs_def.bat` (Windows) or `mjs_def.sh` (Linux). Note that in Linux, you do not require the command `mjs install`.

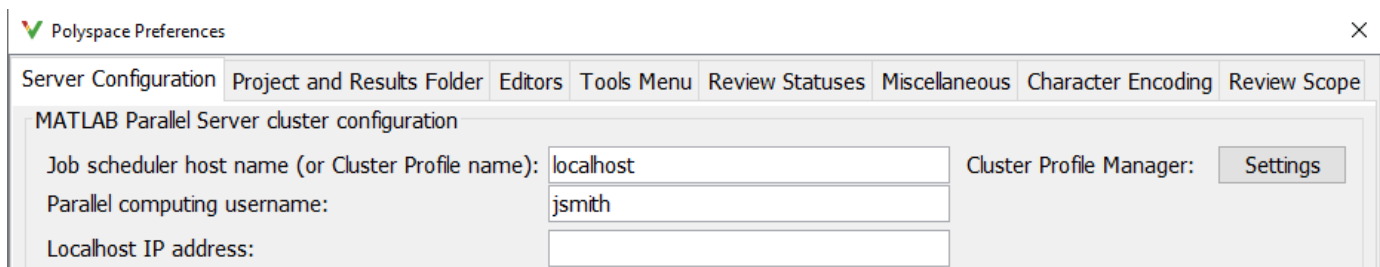
Instead of the command line, you can also start the services from the Admin Center interface. See “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

For more information on the commands, see “Configure Advanced Options for MATLAB Job Scheduler Integration” (MATLAB Parallel Server).

## Configure Client

Open the user interface of the desktop product, Polyspace Bug Finder. Navigate to `polyspaceroot\polyspace\bin`, where `polyspaceroot` is the Polyspace desktop product installation folder, for instance, `C:\Program Files\Polyspace\R2023a` and double-click the `polyspace` executable.

Select **Tools > Preferences**. On the **Server configuration** tab, enter the host name of your computer for **Job scheduler host name**.



You are now set up for the server-client workflow.

## Send Analysis from Client to Server

Run Bug Finder on the file `numerical.c` provided with your installation.

Before running these steps, to avoid entering full paths to the Polyspace executables, add the path `polyspaceroot\polyspace\bin` to the PATH environment variable on your operating system. Here `polyspaceroot` is the Polyspace desktop product installation folder, for instance, `C:\Program Files\Polyspace\R2023a`. To check if the path is already added, open a command line terminal and enter:

```
polyspace-bug-finder -h
```

If the path to the command is already added, you see the full list of options.

- 1 Copy the file `numerical.c` from `polyspaceroot\polyspace\examples\cxx\Bug_Finder_Example\sources` to a folder with write permissions.
- 2 Open a command terminal. Navigate to the folder where you saved `numerical.c` and enter the following:

```
polyspace-bug-finder -sources numerical.c -checkers numerical  
-results-dir . -batch -scheduler hostname
```

Here, *hostname* is the host name of your computer.

After compilation, the analysis is submitted to a server and returns a job ID. To run a Code Prover analysis, use `polyspace-code-prover` instead of `polyspace-bug-finder`. You can run the `polyspace-code-prover` command with a Polyspace Bug Finder license only, provided you use the `-batch` option.

- 3 See the status of the current job.

```
polyspace-jobs-manager listjobs -scheduler hostname
```

You can locate the current job using the job ID.

- 4 Once the job is completed, you can explicitly download the results.

```
polyspace-jobs-manager download -job jobID -results-folder .  
-scheduler hostname
```

Here, *jobID* is the job ID from the submission.

The results folder contains the downloaded results file (with extension `.psbf`). Open the results in the user interface of the desktop product, Polyspace Bug Finder.

## See Also

### More About

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

# Bug Finder Analysis in IDEs

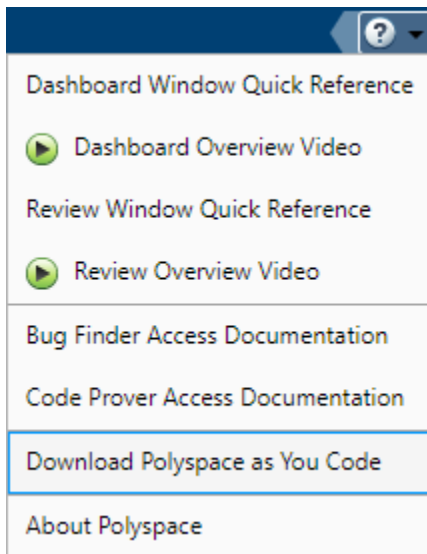
---

## Check Code Quality in IDE Before Submitting

Polyspace as You Code is a static code analysis software meant for regular use by C/C++ developers within their Integrated Development Environments (IDEs). Polyspace as You Code can find bugs and coding standard violations on the file that is currently active in the IDE. (For full integration analysis of a project, use Polyspace Bug Finder or Polyspace Bug Finder Server.)

### Install Polyspace as You Code Analysis Engine and IDE Extensions

Polyspace as You Code comes bundled with a Polyspace Access installation meant for teams or organizations. Once the Polyspace Access web server is set up, any of the licensed users can download the Polyspace as You Code installer as a zipped file from the Polyspace Access web interface.



Conceptually, Polyspace as You Code consists of these parts:

- An analysis engine
- An IDE extension that allows you to launch an analysis and view results in your IDE

IDE extensions are provided for these IDEs: Visual Studio, Visual Studio Code and Eclipse. If you use another IDE, you can still install the analysis engine and run the analysis from the command line or IDE console.

Unzip and start the installer and follow the on-screen instructions. After the analysis engine is installed, you have a choice to install one or more IDE extensions. For more information, see “Install Polyspace as You Code Using Installer”.

Alternatively, you can install the IDE extensions later. For more information, see:

- “Install Polyspace as You Code Extension in Visual Studio”
- “Install Polyspace as You Code Extension in Visual Studio Code”
- “Install Polyspace as You Code Plugin in Eclipse”

## Run Polyspace as You Code and Review Results

After installation, each time you open your IDE, the Polyspace as You Code extension is ready to start an analysis. If you open a C or C++ file, make some changes and save the file, an analysis starts automatically. (You can disable the automatic analysis and choose to launch an analysis explicitly instead.)

To start an analysis, in your IDE, open the project or workspace that you are currently working on, and open a file in the project. Alternatively, copy the following function into a `.c` or `.cpp` file and open the file in your IDE (using a project or otherwise). The function contains bugs such as array access out of bounds, unnecessary code, and use of assignment operator instead of equality.

```
#define MAXBUF 20
int buf[MAXBUF];

int saturateAndShift(int limit, int* stream, int size) {
    int i;
    int numMax = 0;

    if(size > MAXBUF) {
        return -1;
    }

    if(size <= MAXBUF) {
        for(i=0; i<size; i++) {
            if(stream[i] > limit || stream[i] < 0) {
                buf[i+1] = 0;
            }
            else if(stream[i] = limit){
                buf[i+1] = stream[i];
                numMax ++;
            }
            else {
                buf[i+1] = stream[i];
            }
        }
    }
    return numMax;
}
```

After a Polyspace as You Code analysis, you can see the results (bugs and coding standard violations) as source code markers on the currently active file. You can also see the results in a separate list in the IDE. For more information, see:

- “Run Polyspace as You Code in Visual Studio and Review Results”
- “Run Polyspace as You Code in Visual Studio Code and Review Results”
- “Run Polyspace as You Code in Eclipse and Review Results”

You can also export the results on a command line terminal or IDE console. For richer results, you can export the results to a JSON format and manipulate them further before display. For more information, see “Run Polyspace as You Code from Command Line and Export Results”.

## Configure Polyspace as You Code IDE Extension

The default analysis is preconfigured to work on small projects in IDEs. In practice, you might have to configure the IDE extension settings further to emulate your build closely, to enable or disable checkers, to see new results only or for other adjustments.

For instance, by default, Polyspace as You Code runs each time you save your code. You can disable the automatic runs using an extension setting (and run the analysis explicitly with right-click options on the source code). For the complete list of extension settings and how to open them, see:

- “Configure Polyspace as You Code Extension in Visual Studio”
- “Configure Polyspace as You Code Extension in Visual Studio Code”
- “Configure Polyspace as You Code Plugin in Eclipse”

The extension settings fall into three major groups:

- *Build options:*

Using these settings, you specify whether to extract build information from existing artifacts in IDEs such as a Visual Studio solution or a Visual Studio Code build task, or to manually enumerate build-related Polyspace options in an options file. For more information, see “Analyzing Build in Polyspace as You Code”.

- *Checkers:*

Using these settings, you can enable or disable checkers. For more information, see “Setting Checkers in Polyspace as You Code”.

- *Baselining options:*

Using these settings, you can connect your Polyspace as You Code installation with a Polyspace Access instance, and baseline your results using a project in Polyspace Access. Baselining allows you to focus only on new results because of recent code changes. See “Baselining in Polyspace as You Code”.



## Perform Polyspace as You Code Analysis in Visual Studio Code

Polyspace as You Code helps you find defects and coding standard violations while developing in the Visual Studio Code IDE. You can run an analysis and fix as you code, saving you from finding bugs late in the development cycle. When connected to the Polyspace Access central repository, Polyspace as You Code highlights new issues compared to the development baseline.

The tasks in this guide give an overview of basic actions:

- 1 Configure Polyspace as You Code in the Visual Studio Code IDE.
- 2 Run an analysis and justify or fix findings.
- 3 Download and sync a baseline from Polyspace Access to compare with new findings.

The guide assumes you have a working knowledge of Visual Studio Code.

Before beginning the tutorial, confirm Polyspace as You Code is installed on your machine. See “Install Polyspace as You Code Extension in Visual Studio Code”.

Use your own code or copy and paste this code into Visual Studio Code to follow along with the guide.

`example.hpp`

```
#include <string>

class Demo {
    void myFunction(std::string buffer) const{

    }
};
```

`example_source.cpp`

```
#include "example.hpp"

int simple_defect(){
    uint32_t x = -1;
    x++;
    return 1 / x;
}
```


To start the guide, see “Configure Polyspace as You Code in Visual Studio Code” on page 5-6.

## Configure Polyspace as You Code in Visual Studio Code

Configure the Polyspace as You Code extension before you begin your first analysis. Configuration of the extension allows for customization of your workstation and analysis preferences. Settings are preserved between sessions.

In this tutorial, you:


- 1 Manually configure the build.
- 2 Set automatic quality monitoring.
- 3 Set analysis on save.
- 4 Configure Polyspace checkers.
- 5 Analyze header files.
- 6 Set analysis script.

Open your files in Visual Studio Code. Click the Polyspace icon  displayed in the Visual Studio Code sidebar to open the Polyspace extension sidebar. The sidebar contains different panes that are referenced in this guide, such as **Quality Monitoring**, **Configuration**, and **Baseline**.

### Manually Configure Your Build

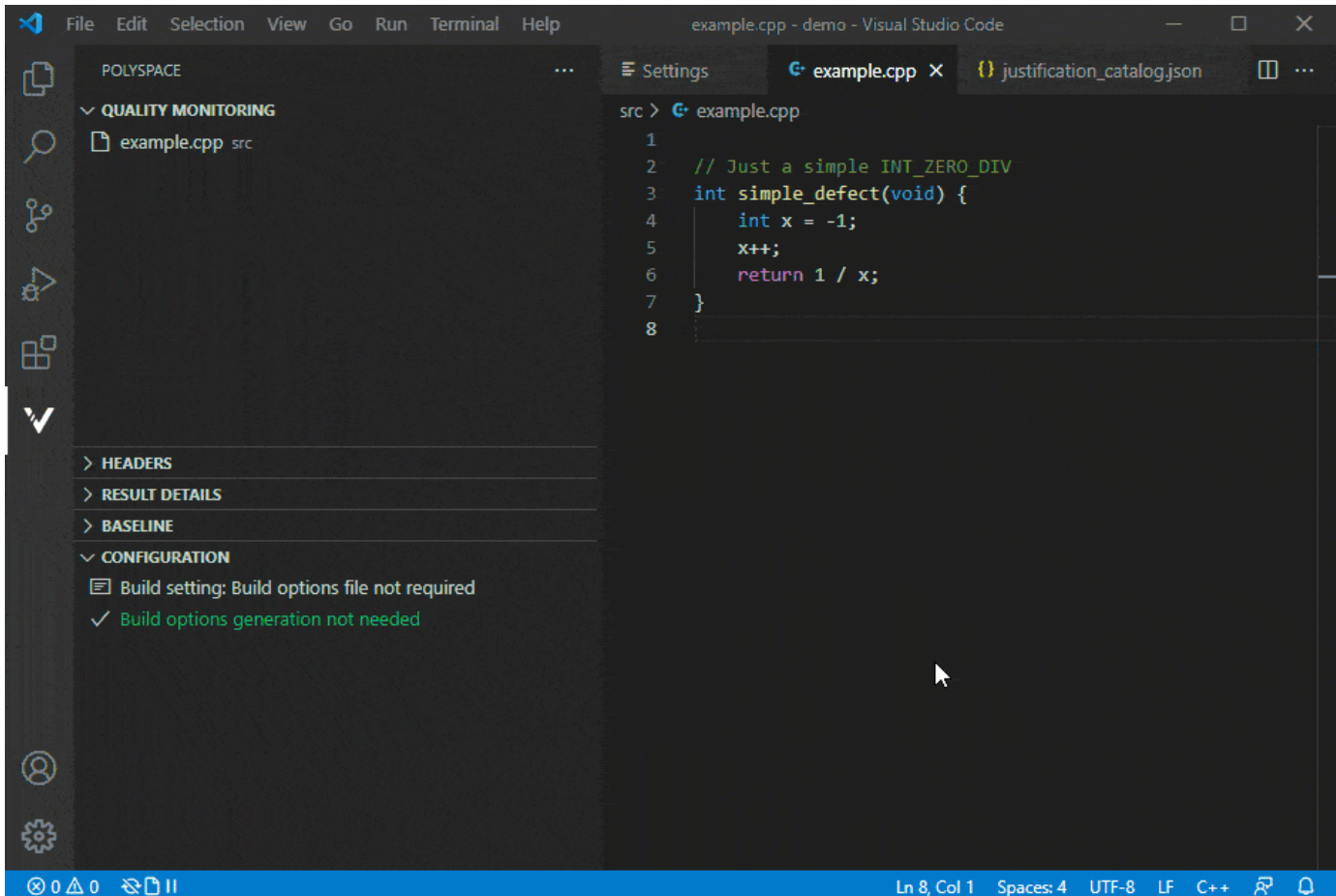
You have the option of manually configuring your build. Manual setup of the analysis involves specifying build options. You can extract build options from a Visual Studio Code build task or JSON Compilation Database file, or specify them in a build options file.

For this tutorial, you do not set the build options manually.

Click the settings icon  in the **Configuration** pane of the **Polyspace** sidebar or go to **Settings**. Search for `polyspace.analysisOptions` to view the various build options. The **Manual Setup: Build** setting allows you to choose how to provide your options file. Use the **Manual Setup: Build** option of the same file type to provide the path to your options file.

For more information on build options, see “Analysis Setup”.


## Manual Build Configuration



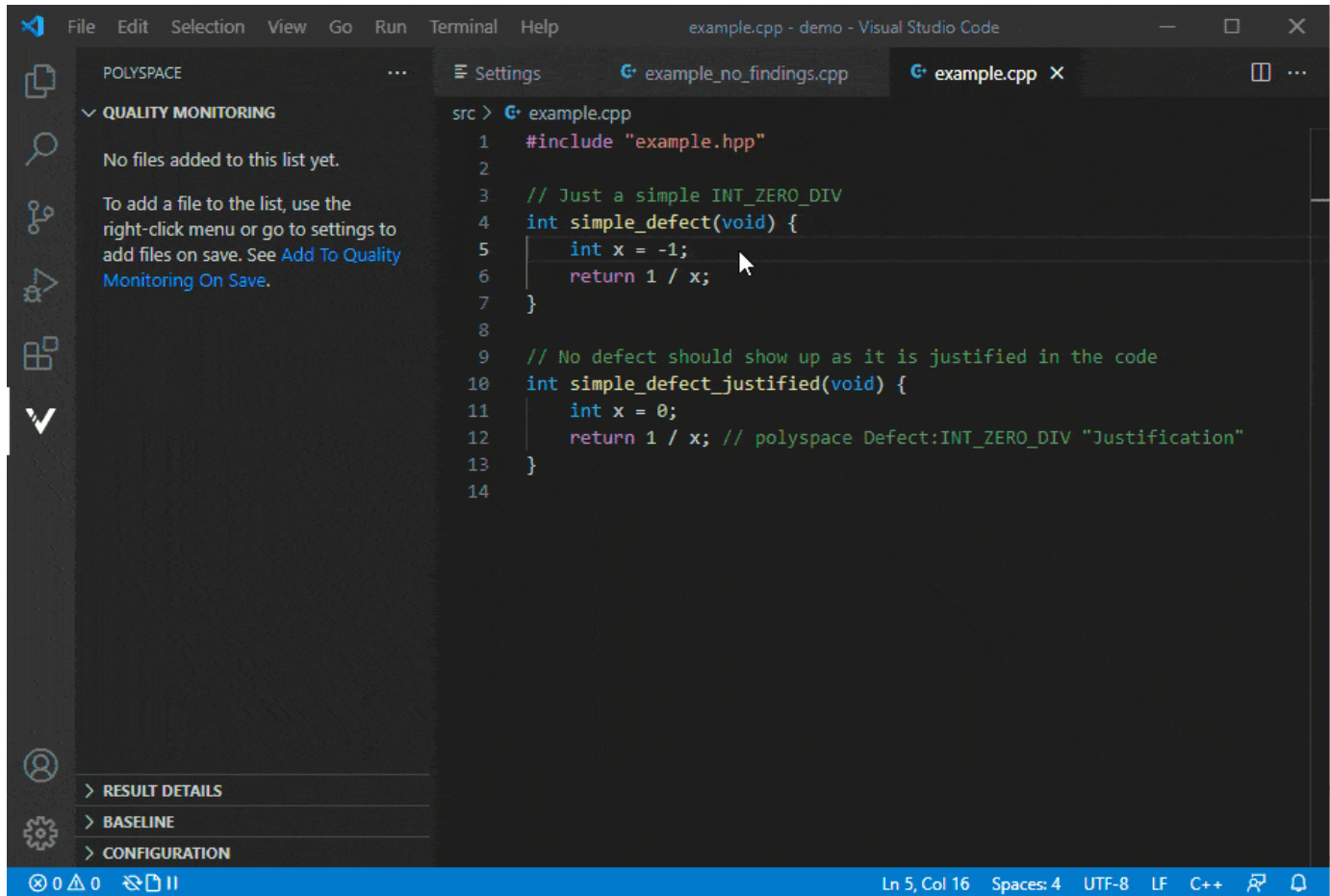
## Set Automatic Quality Monitoring

Polyspace has a **Quality Monitoring** list that keeps track of edited files. Polyspace can analyze all files listed in the **Quality Monitoring** list at the same time. You can choose to have Polyspace automatically add files to the **Quality Monitoring** list or manually add files.


If the setting `polyspace.analysisOptions.addToQualityMonitoringOnSave` is active, Polyspace adds files to the **Quality Monitoring** list automatically when a file is saved. You can manually add files to the list by right-clicking inside the file editor and selecting **Add file to the Polyspace Quality Monitoring list**.


Turn on automatic quality monitoring. Click the settings icon in the **Quality Monitoring** pane of the **Polyspace** sidebar  and select the check box for **Add To Quality Monitoring On Save**. Alternatively, open the **Settings** tab and search for the option `polyspace.analysisOptions.addToQualityMonitoringOnSave`. Select the check box to activate the setting.

## Automatically Add Files to Quality Monitoring List

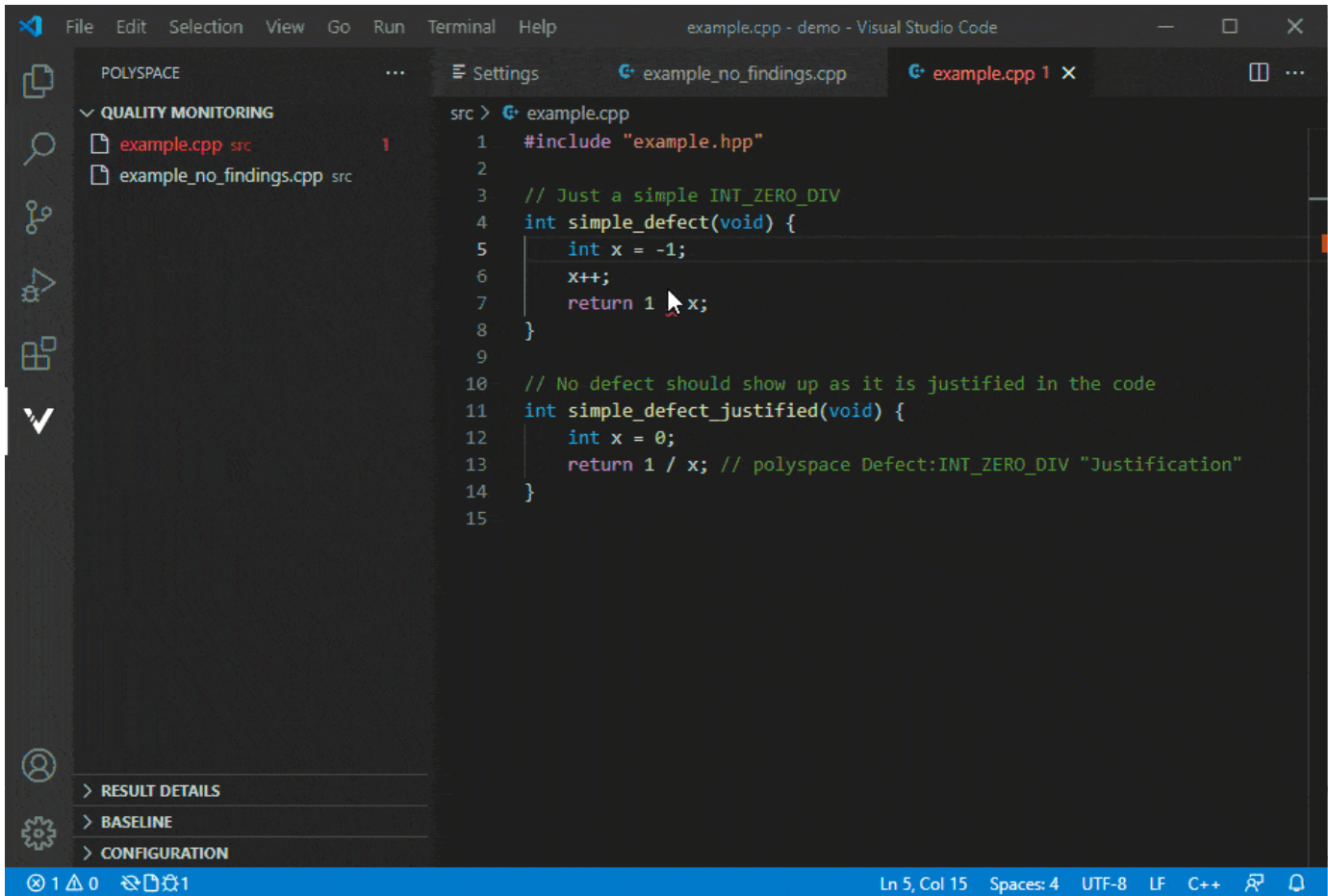


## Set Analysis on Save

Similarly to quality monitoring, Polyspace can automatically run an analysis of any file in the **Quality Monitoring** list when the file is saved. Alternatively, you can manually run analysis of individual files or a group of files. Manually trigger an analysis by clicking the Run Polyspace Analysis icon  next to the file in the **Quality Monitoring** list or right-click inside the file editor and select **Run Polyspace Analysis**.


Turn on automatic analysis of files. Click the settings icon in the **Quality Monitoring** pane of the **Polyspace** sidebar  and select the check box for **Analysis Of Files On Save**. Alternatively, open the **Settings** tab and search for the option `polyspace.analysisOptions.analysisOfFilesOnSave`. Select the check box to activate the setting.

## Automatically Run Analysis on Save



## Configure Polyspace Checkers

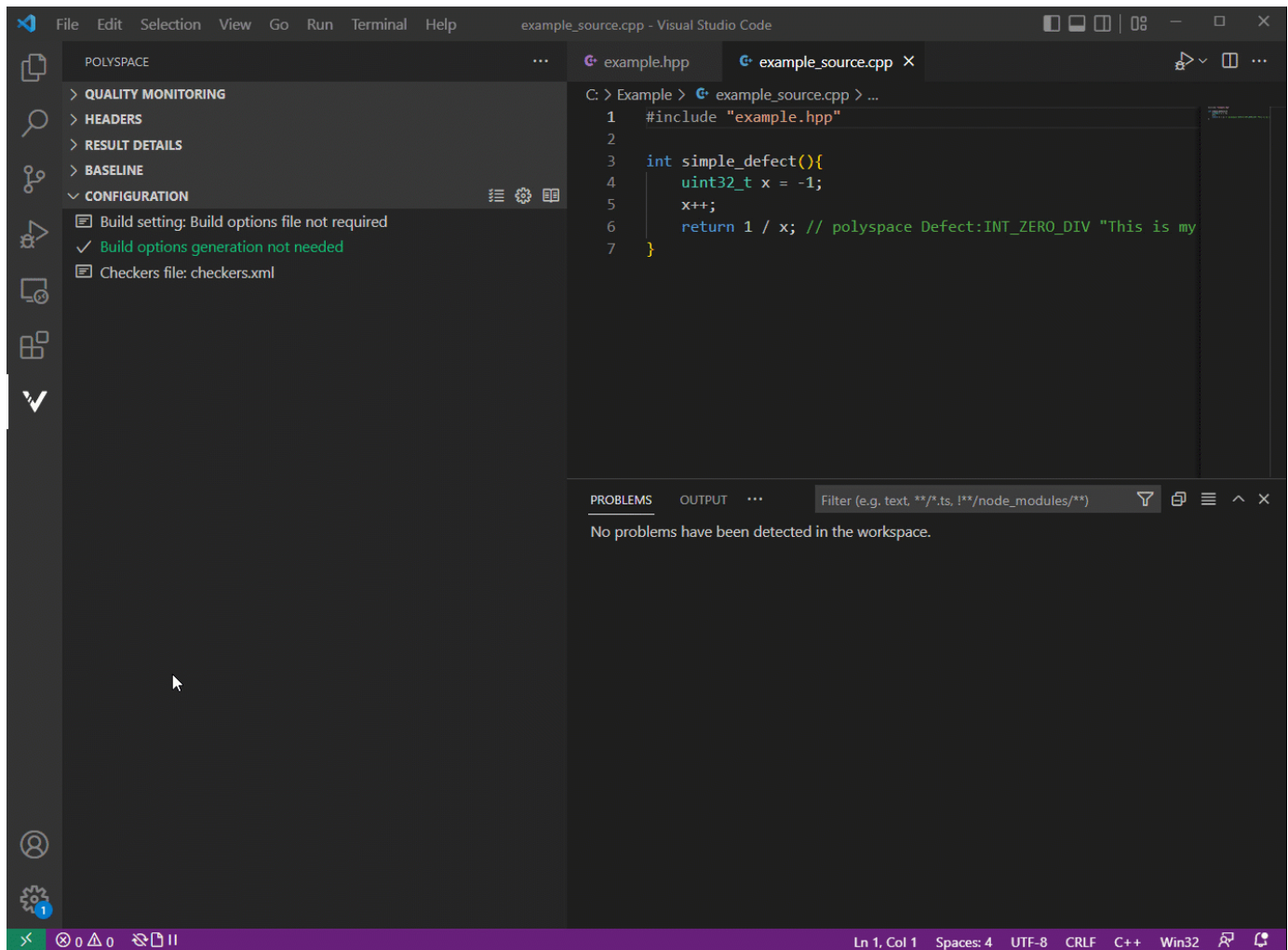
Polyspace checks for a default set of defects and standards. You can customize this set of standards and defects based on which certification standards you want to check for. Use the settings option `polyspace.analysisOptions.manualSetup:checkersfile` to set the checkers XML file for your project.

Turn on the AUTOSAR C++14 checkers. In the **Configuration** section of the **Polyspace** sidebar, click the **Configure Checkers** icon  to open the **Checkers Selection** window.

You can create, save, and open saved checkers selection files. For more information, see “Configure Checkers for Polyspace as You Code in Visual Studio Code”.


Click on **AUTOSAR C++14** on the left pane of the **Checkers Selection** window. In the right pane, select the **All** check box to enable all of the AUTOSAR C++14 rules. Click **Save Changes** in the top right corner to save your checker selection and close the window to return to the Visual Studio Code application.

## Configure Polyspace Checkers




## Analyze Header Files

You can run an analysis on header files as if they were source files.

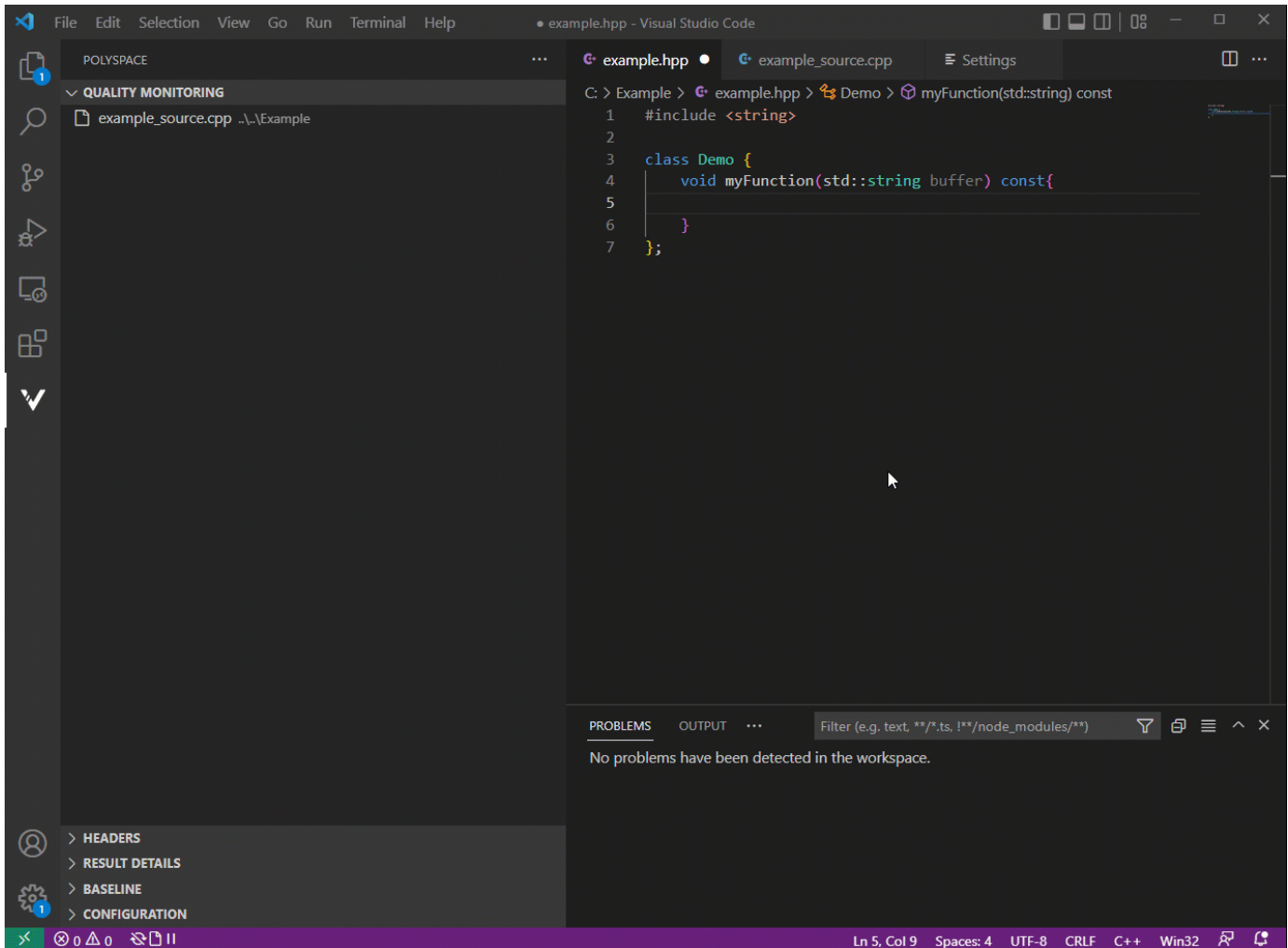
First, define the extensions of your header files. Click the settings icon in the **Headers** pane of the Polyspace sidebar . Alternatively, open the **Settings** tab and search for the option `polyspace.otherSettings.headersExtensions`.

Click **Add Item** and add the file extension `.hpp` if it is not already there. Click **OK** to save the file extension in the list.

Manually add the `example.hpp` header file to the **Quality Monitoring** list and initiate an analysis by clicking the **Run Polyspace Analysis** button .

See “Headers Included in Current File Not Analyzed”.

## Analyze Header Files



## Set Analysis Script

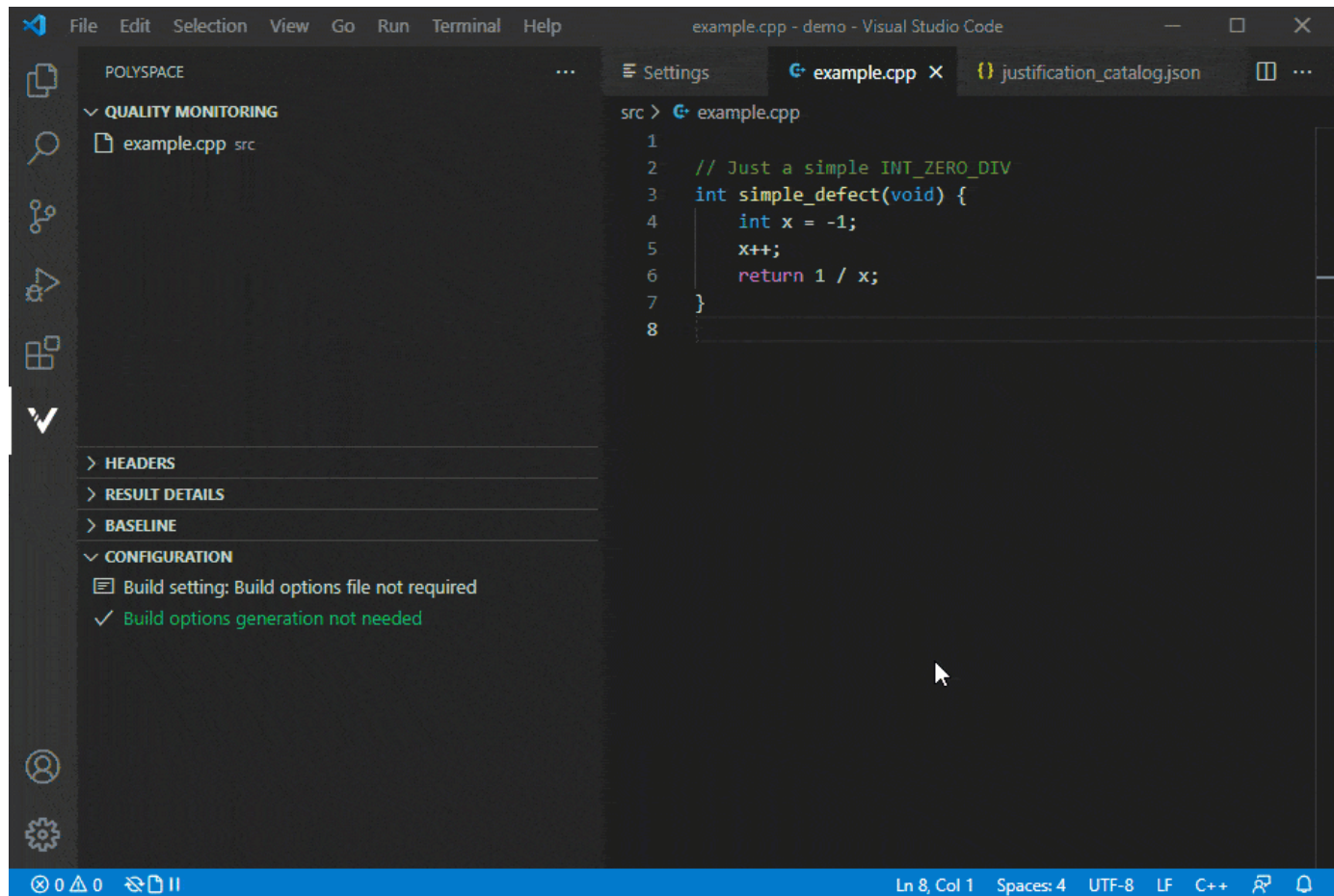
You can provide a script to configure and run your Polyspace analysis in Polyspace as You Code.

For this tutorial, you do not set a script.

If you have a script, open the **Settings** tab and search for the option `polyspace.analysisOptions.script`.

Provide the path to the script in the **Script File** setting text box. Turn on **Script** for the **Analysis Setup** setting using the list.

## Set Analysis Script



Once you have configured the Polyspace as You Code extension, you are ready to run an analysis and review your results.




## Run and Review Results in Polyspace as You Code for Visual Studio Code

After you configure the Polyspace as You Code extension, you are ready to run an analysis and fix or justify any findings. Before continuing this guide, confirm you have added `example.hpp` and `example_source.cpp` to the **Quality Monitoring** list.

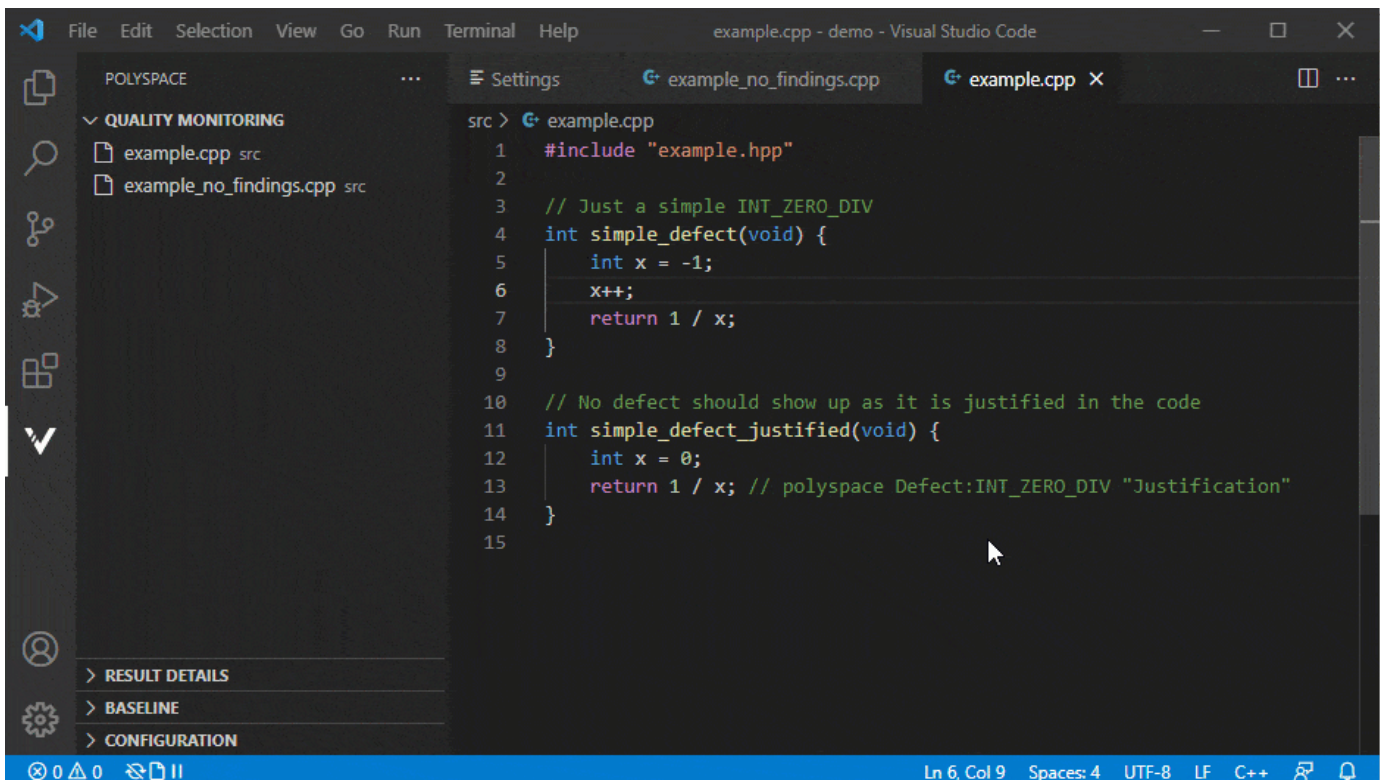
### Run Polyspace as You Code Analysis in Visual Studio Code

You can run an analysis in these ways.

- If you set **Analysis Of Files On Save**, save your edited file to run an analysis.
- Right-click a file and select **Run Polyspace Analysis**.
- Manually add your file to the **Quality Monitoring** list and click the **Run Polyspace Analysis** button  to analyze your file.

Click the **Run Polyspace Analysis** next to `example_source.cpp` to run an analysis.

#### Manually Run Analysis



### View, Fix, or Justify Findings

After Polyspace runs the analysis, the filename in the **Quality Monitoring** pane is green if there are no findings and red if there are findings. A number to the right of the filename indicates the number

of findings. View your findings in the **Problems** pane. Open the **Problems** pane by clicking the filename in the **Quality Monitoring** pane.

The **Problems** pane shows all findings as a list. Use the filter within the **Problems** pane to search for specific findings. Each finding contains a line and column number to indicate where the finding exists within your code.

After you run an analysis on the example code, open the results in the **Problems** pane. Use the filter to search for A16-2-3. There is one result on line five, column five.

Click the finding in the **Problems** pane to open the file that contains the finding. Each finding is noted in the file with a red underline. Hover over the underlined sections of code to see a list of each finding associated with the problem.

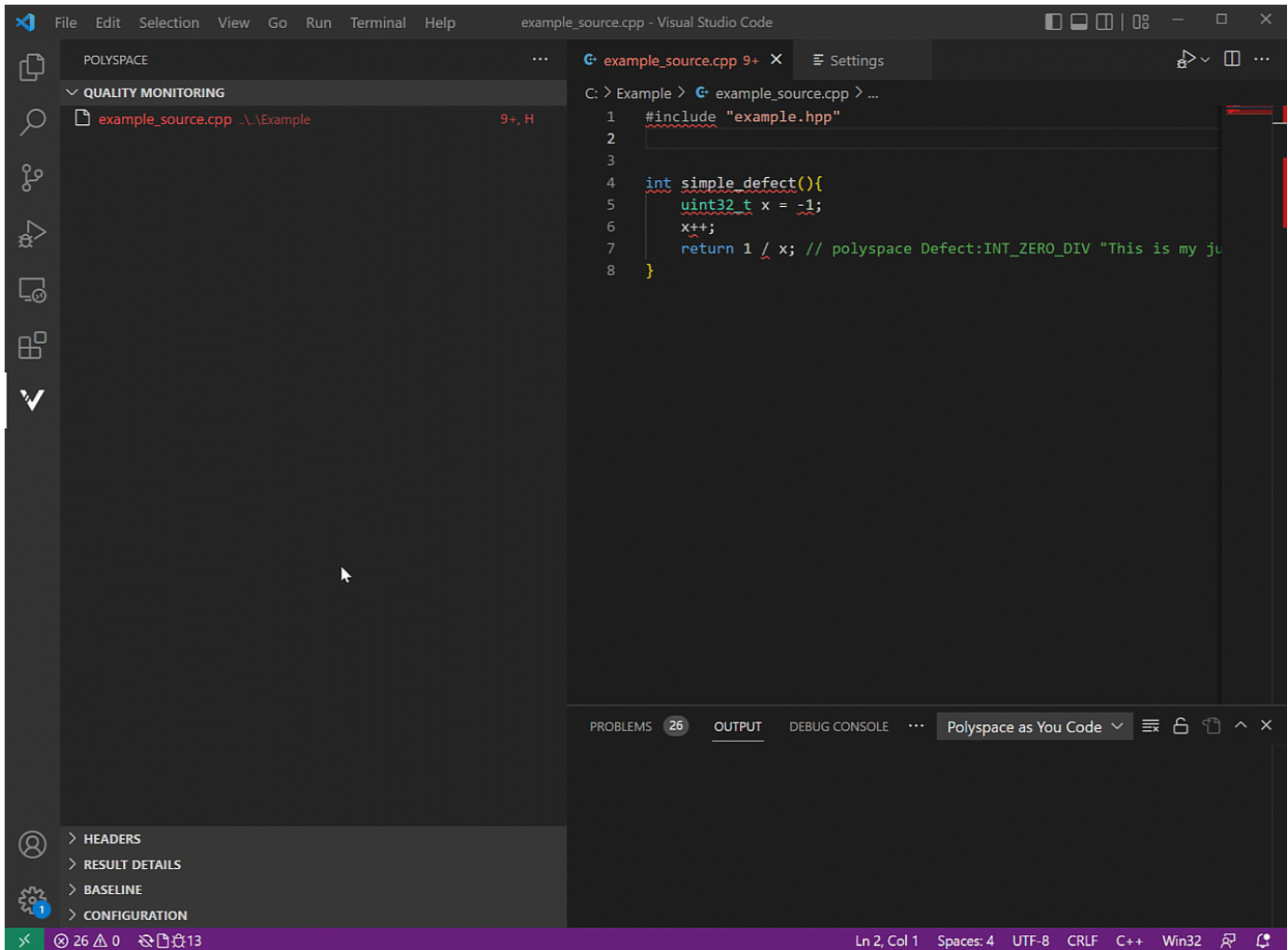
In the example code, add this include on line two:

```
#include <cstdint>
```

Save this change and run another analysis. The A16-2-3 error on line five is no longer present.



With automatic analysis active, you can find and fix findings during the code authoring process. Each time you save changes, a Polyspace analysis begins in the background and displays any new findings or removes fixed findings from the list.

## View and Fix Findings

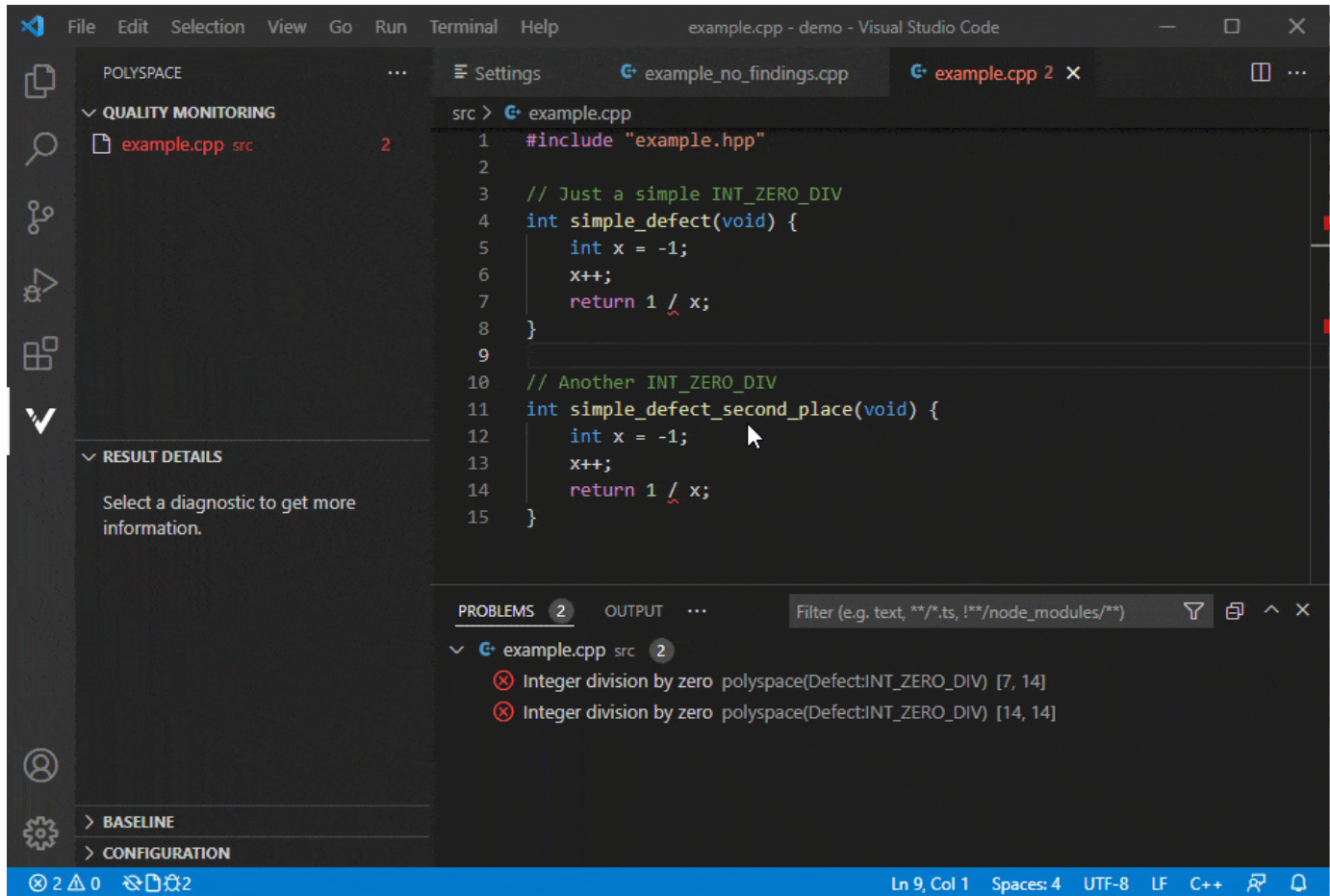


## Justify Individual Findings

You can justify a finding from the **Problems** pane or from the location of the finding within the code.

Click on a finding in the **Problems** pane to change the red error icon  to a light bulb icon . Click on the light bulb icon and select the appropriate justification from the menu. This adds a comment to your code which you can amend. Adding a justification removes the finding from the **Problems** pane. To show the finding again, remove the justification comment and perform the analysis.

## Justify Findings



## Provide Justification Catalog

You can add a catalog of predefined justifications to Polyspace as You Code. The justification catalog allows you to select a prewritten justification instead of manually typing each justification comment.

Open the Visual Studio Code **Settings** and search for `polyspace.justification.catalog`. Enter the path to your justification catalog in the text box relative to the install location. The catalog must be in JSON format. If you do not already have a justification catalog, use the example JSON file.

`example_catalog.json`

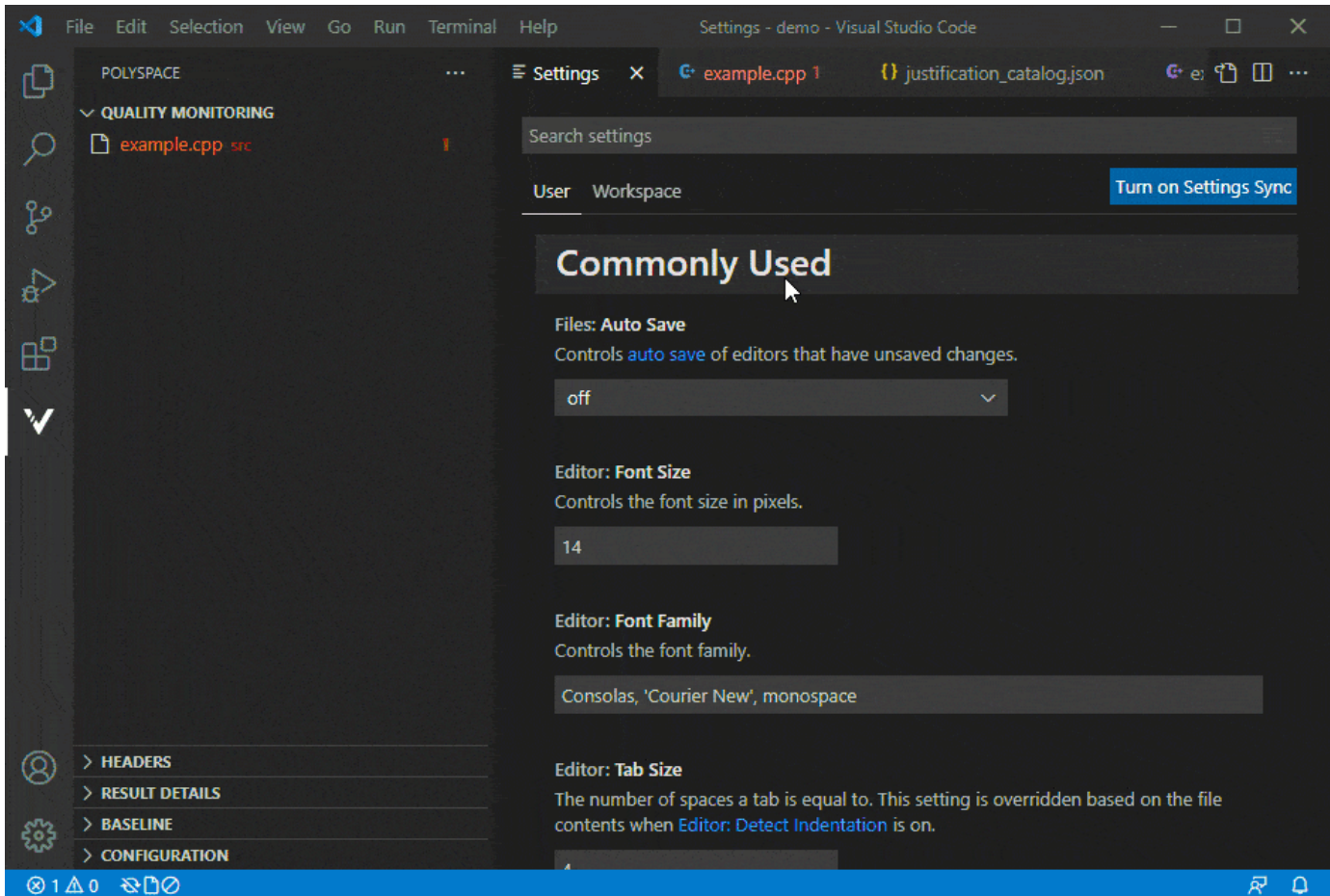
```
{
  "justifications": [
    {
      "family": "Defect",
      "acronym": "INT_ZERO_DIV",
      "comment": "This is my justification for division"
    },
    {
      "family": "Defect",
      "acronym": "INT_ZERO_DIV",
```

```

        "comment": "Alternative justification for division"
    }
}
}


```

## Provide a Justification Catalog

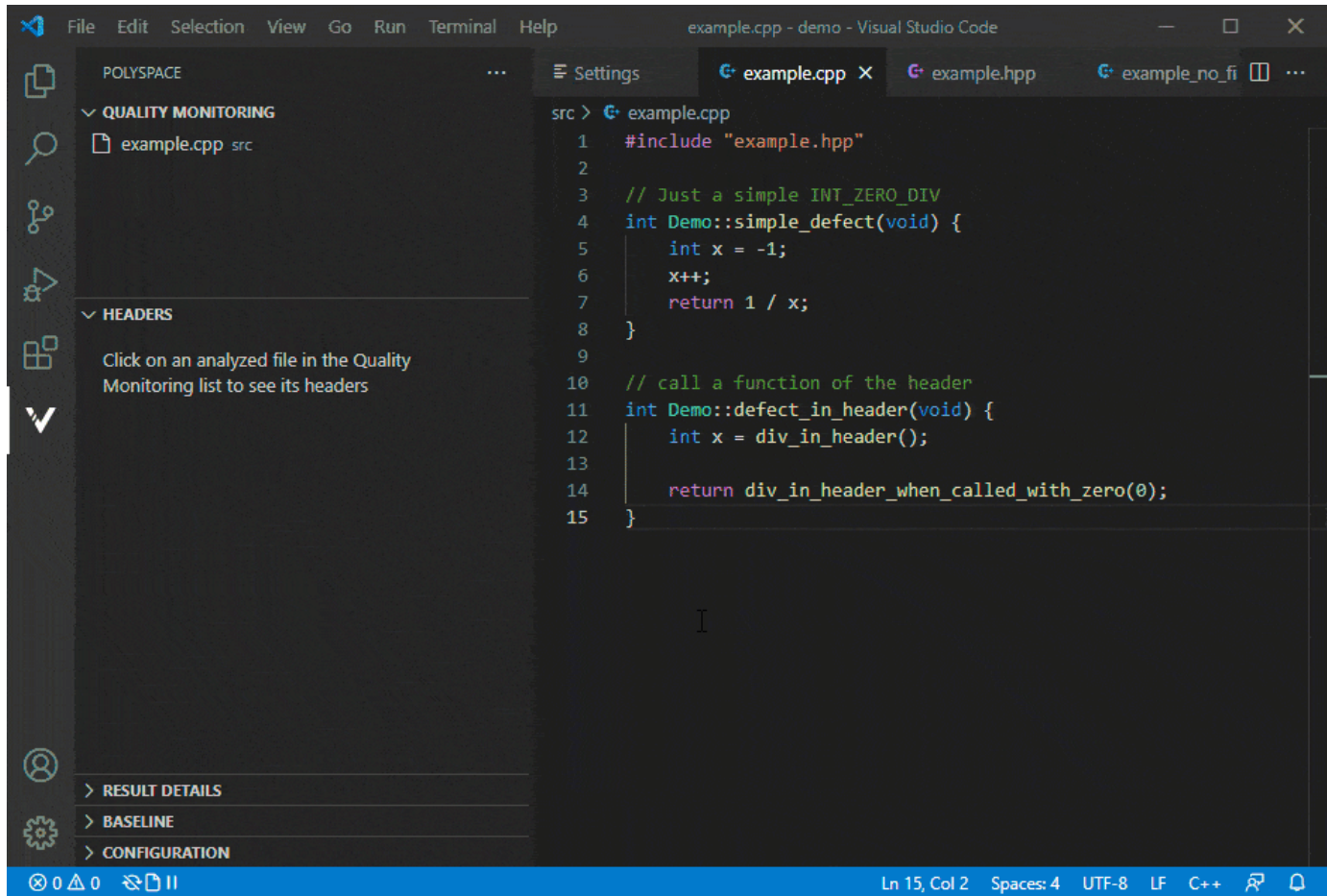


## View Header Findings

When you analyze source files, any header file findings are shown in the **Problems** pane. If a header contains a finding, it is noted with a red **H** next to the source file in the **Quality Monitoring** list.


Click a header finding in the **Problems** pane to go to the header file that contains the finding. To find which source file analysis caused the header finding, click the light bulb icon  next to the finding in the **Problems** pane and then click the option to show details about the finding.

## View Header Findings

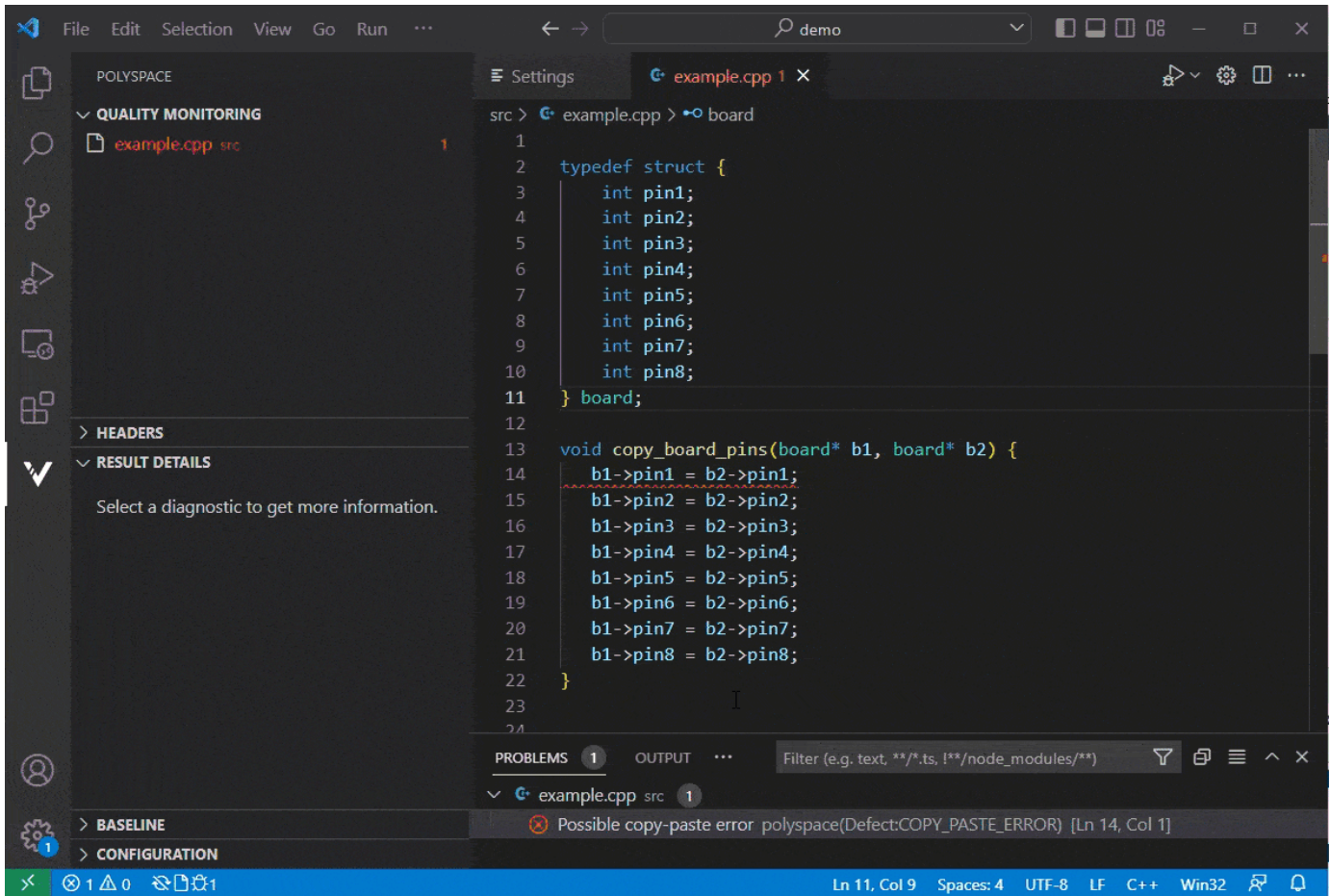


## Check for Potential Duplicate Code

Polyspace checks for potential copy and paste errors, duplicate code, and other code refactoring opportunities.

Click **Possible copy-paste error** in the **Problems** pane to view more information in the **Result Details** pane. Expand the **Traceback** node to locate the duplicate sections occur and view further details. To view the potential error in the Visual Studio Code peek window, point to **Possible copy-paste error** in the **Result Details** pane and click the **Toggle peek view in source** icon .

## Check for Potential Duplicate Code



The screenshot shows the Visual Studio Code interface with the Polyspace extension. The editor displays the following C++ code:

```
src > example.cpp > board
1
2 typedef struct {
3     int pin1;
4     int pin2;
5     int pin3;
6     int pin4;
7     int pin5;
8     int pin6;
9     int pin7;
10    int pin8;
11 } board;
12
13 void copy_board_pins(board* b1, board* b2) {
14     b1->pin1 = b2->pin1;
15     b1->pin2 = b2->pin2;
16     b1->pin3 = b2->pin3;
17     b1->pin4 = b2->pin4;
18     b1->pin5 = b2->pin5;
19     b1->pin6 = b2->pin6;
20     b1->pin7 = b2->pin7;
21     b1->pin8 = b2->pin8;
22 }
23
24
```

The PROBLEMS panel shows a diagnostic message:

```
example.cpp src 1
Possible copy-paste error polyspace(Defect:COPY_PASTE_ERROR) [Ln 14, Col 1]
```


The status bar at the bottom indicates the current position: Ln 11, Col 9, Spaces: 4, UTF-8, LF, C++, Win32.

## Configure and Download Baseline with Polyspace as You Code

For more efficient bug fixing, you can create a baseline using Polyspace Access results. Download the baseline and use it to compare your Polyspace as You Code results and focus on new or unreviewed results.

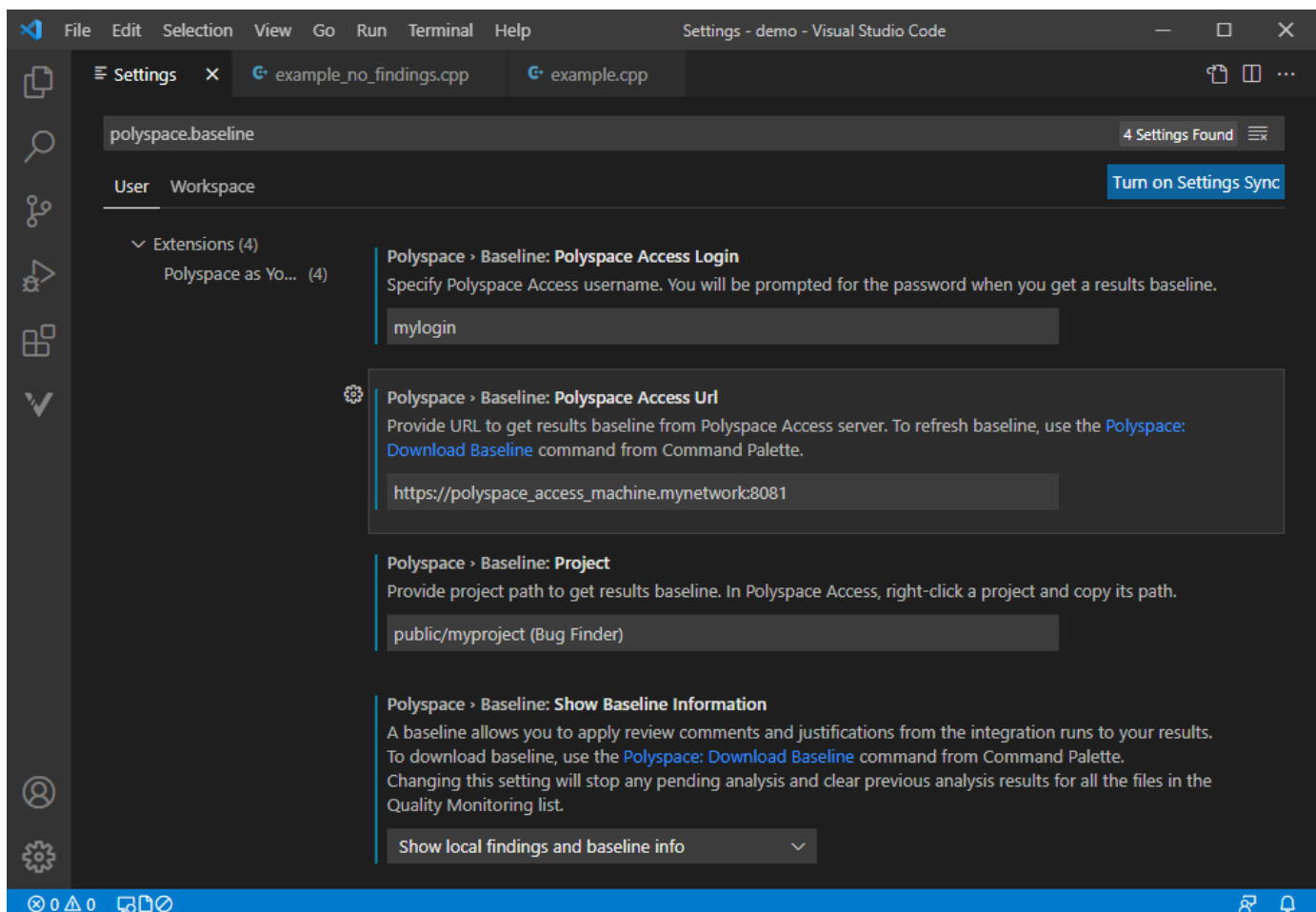
In order to configure baseline results in Polyspace as You Code, you must have a Polyspace Access server login name and password along with an uploaded project result. The project must contain results from an analysis of the same files you are analyzing in Polyspace as You Code.

### Configure Baseline

Configure a baseline using your Polyspace Access server information. Click the settings icon in the **Baseline** pane of the **Polyspace** sidebar  or go to **Settings** and search for the option `polyspace.baseline`.

Enter your login, the server URL, and the project path for the project you want to create a baseline for.


### Configure Baseline






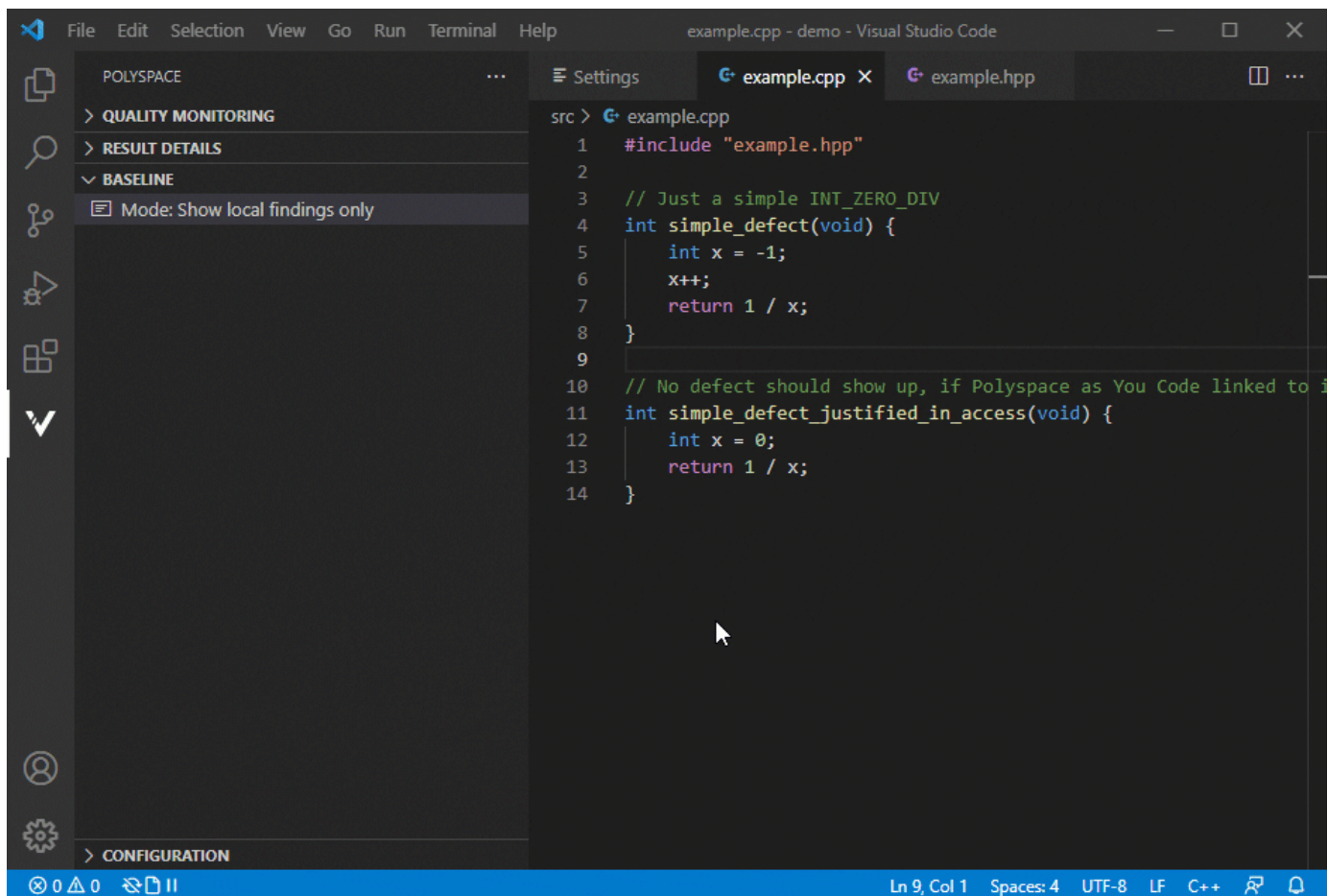
## Download Baseline

To keep using the most up-to-date baseline information, make sure that you periodically update your baseline by downloading the latest information from Polyspace Access.

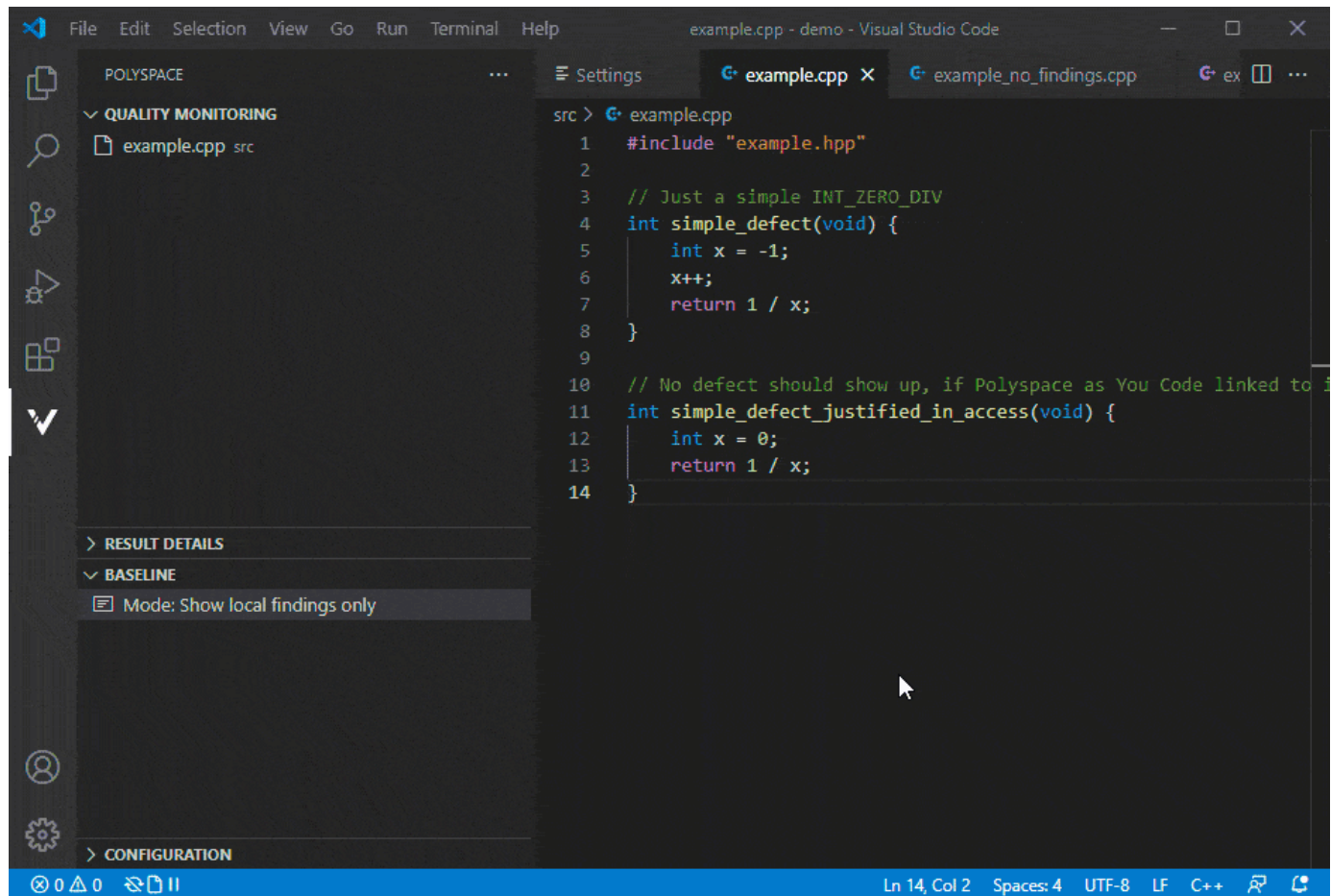
Click the settings icon in the **Baseline** pane of the **Polyspace** sidebar  or go to **Settings** and search for the option `polyspace.baseline`. Under the **Show Baseline Information** setting, select **Show local findings and baseline info** from the list.

**Baseline not downloaded** is shown in red in the **Baseline** pane of the **Polyspace** sidebar. Click the cloud download icon at the top of the **Baseline** pane  to download the baseline. Click this cloud download icon any time you want to update your baseline with the latest information.

### Download Baseline




## Update Baseline

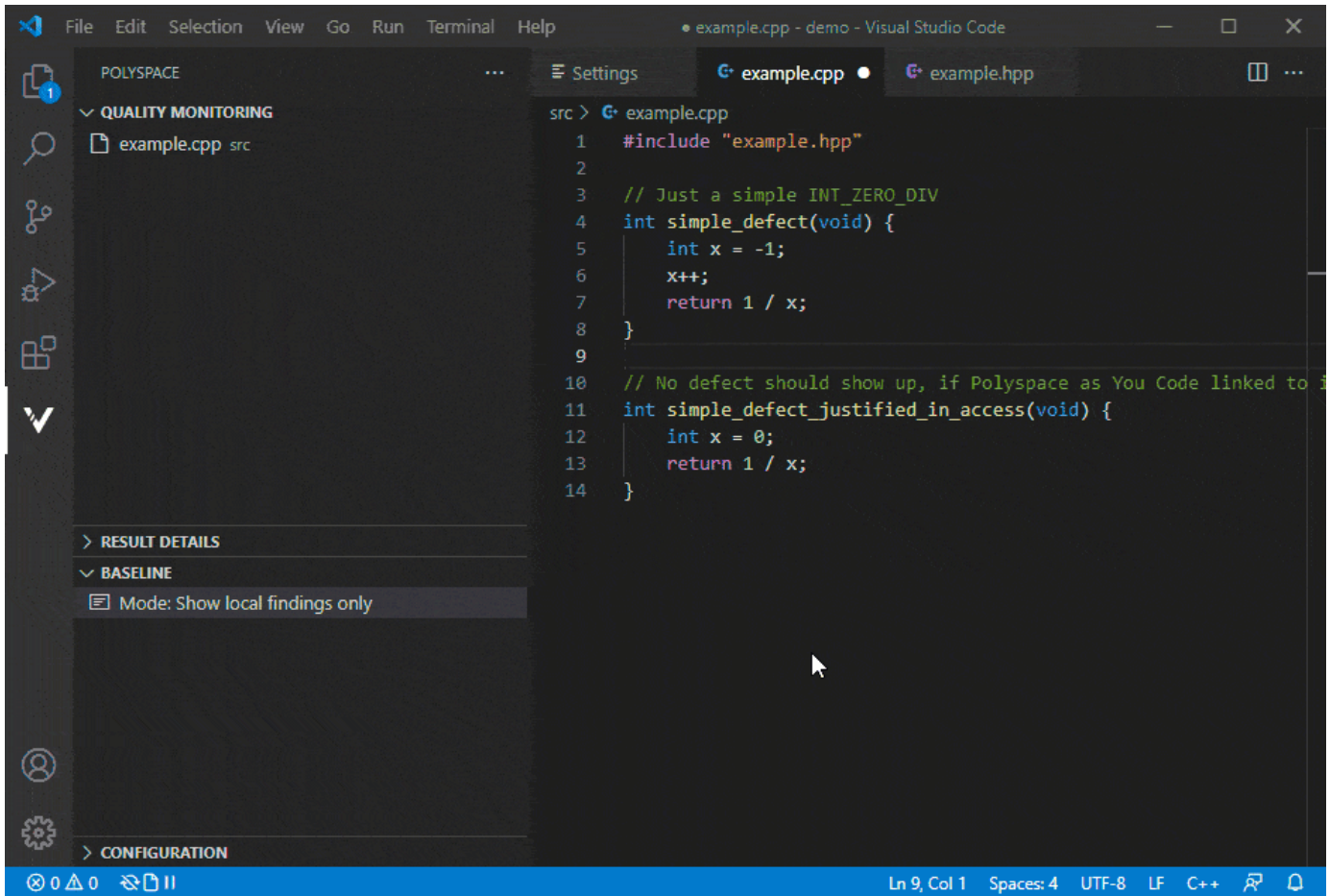


## Show New Findings and Compare Results

Sometimes it is beneficial to only view new results compared to the baseline.

To view only new findings, click the settings icon in the **Baseline** pane of the **Polyspace** sidebar  or go to **Settings** and search for the option `polyspace.baseline`. Under the **Show Baseline Information** setting, select **Show new findings only**.

## Show New Results



## Perform Polyspace as You Code Analysis in Eclipse

Polyspace as You Code helps you find defects and coding standard violations while developing in the Eclipse IDE. You can run an analysis and fix findings as you code, saving you from finding bugs late in the development cycle. When connected to the Polyspace Access central repository, Polyspace as You Code highlights new issues compared to the development baseline.

The examples in this guide show you how to:

- “Configure Polyspace as You Code in Eclipse” on page 5-25
- “Run and Review Results in Polyspace as You Code for Eclipse” on page 5-32
- “Configure and Download Baseline with Polyspace as You Code in Eclipse” on page 5-38

The examples assume you have a working knowledge of Eclipse.

Before you begin the first step:

- Confirm Polyspace as You Code is installed on your machine. See “Install Polyspace as You Code Plugin in Eclipse”.
- Use your own code or copy and paste this code into Eclipse to follow along with the guide.

example.hpp

```
#include <string>

class Demo {
    void myFunction(std::string buffer) const{

    }
};
```

example\_source.cpp

```
#include "example.hpp"

int simple_defect(){
    uint32_t x = -1;
    x++;
    return 1 / x;
}
```

For the first example, see “Configure Polyspace as You Code in Eclipse” on page 5-25.

## Configure Polyspace as You Code in Eclipse

Configure the Polyspace as You Code plugin before you begin your first analysis. Configuration of the plugin allows you to customize your workstation and analysis preferences. The plugin preserves settings between sessions.


In this tutorial, you:

- 1 Manually configure the build.
- 2 Automatically add files to the **Quality Monitoring** list.
- 3 Set up automatic analysis of files on save.
- 4 Configure Polyspace checkers.
- 5 View header file findings.
- 6 Configure an analysis script.

Open your files in Eclipse. Open the Polyspace as You Code perspective by going to **Window > Perspective > Open Perspective > Polyspace as You Code**. The perspective contains different panes such as **Quality Monitoring**, **Configuration**, and **Baseline**.

### Manually Configure Your Build

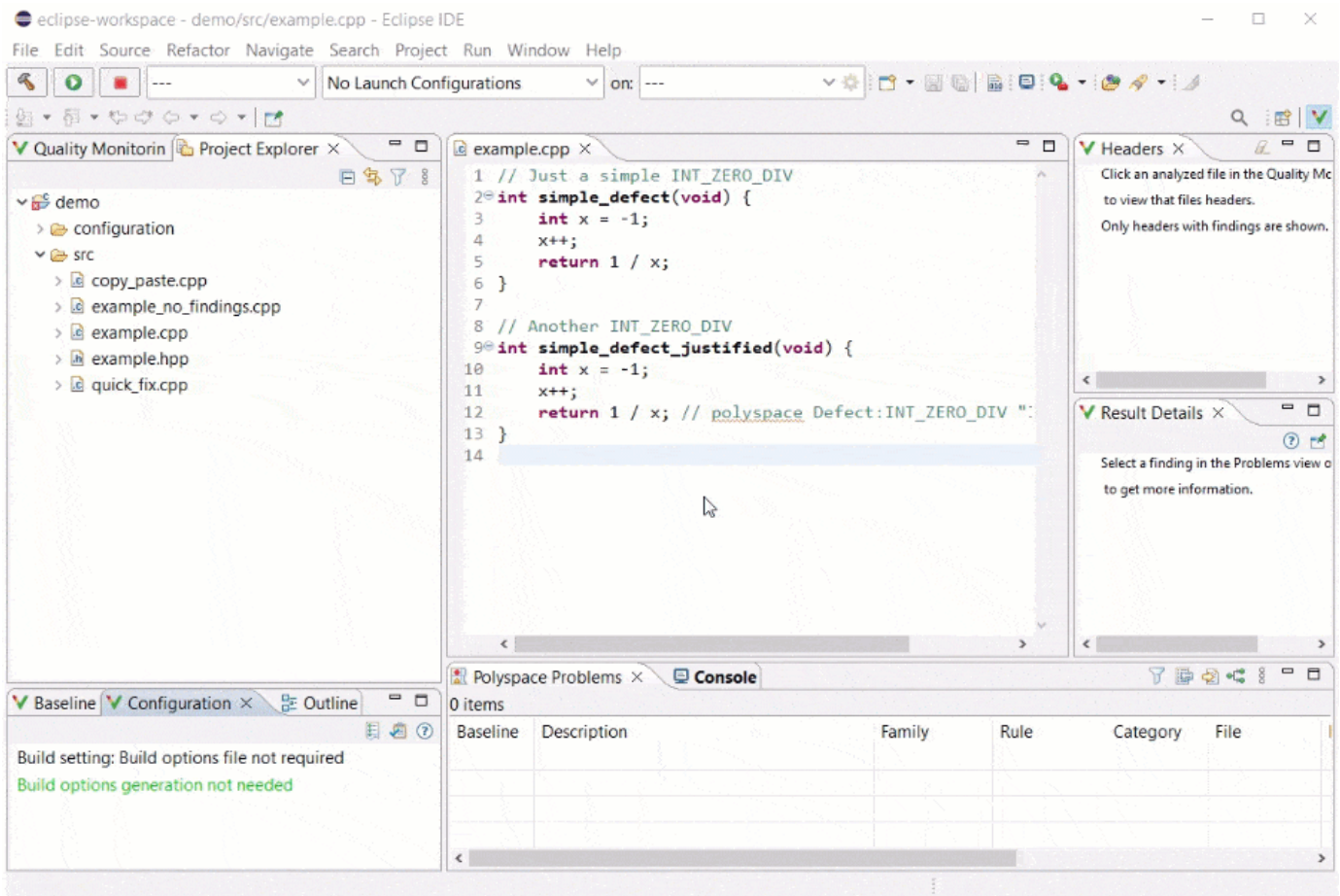
You have the option of manually configuring your build. Manual setup of the analysis involves specifying build options. You can extract build options from a build command, an Eclipse C/C++ project, of a JSON Compilation Database file or specify them in a Polyspace build options file.

- 1 Click the preferences icon  in the **Configuration** pane or go to **Window > Preferences > Polyspace as You Code > Analysis**.
- 2 Choose how to provide your options file using the **Build** setting under **Analysis Setup:: Manual setup**. To provide the path to your options file, use the relevant option of the same file type. For example, if your options file is a JSON Compilation Database file, select **Get from JSON Compilation Database file** and enter the path to the file.

For this tutorial, leave **Build** set as Build options file not required.

For more information on build options, see “Analysis Setup”.


## Manual Build Configuration



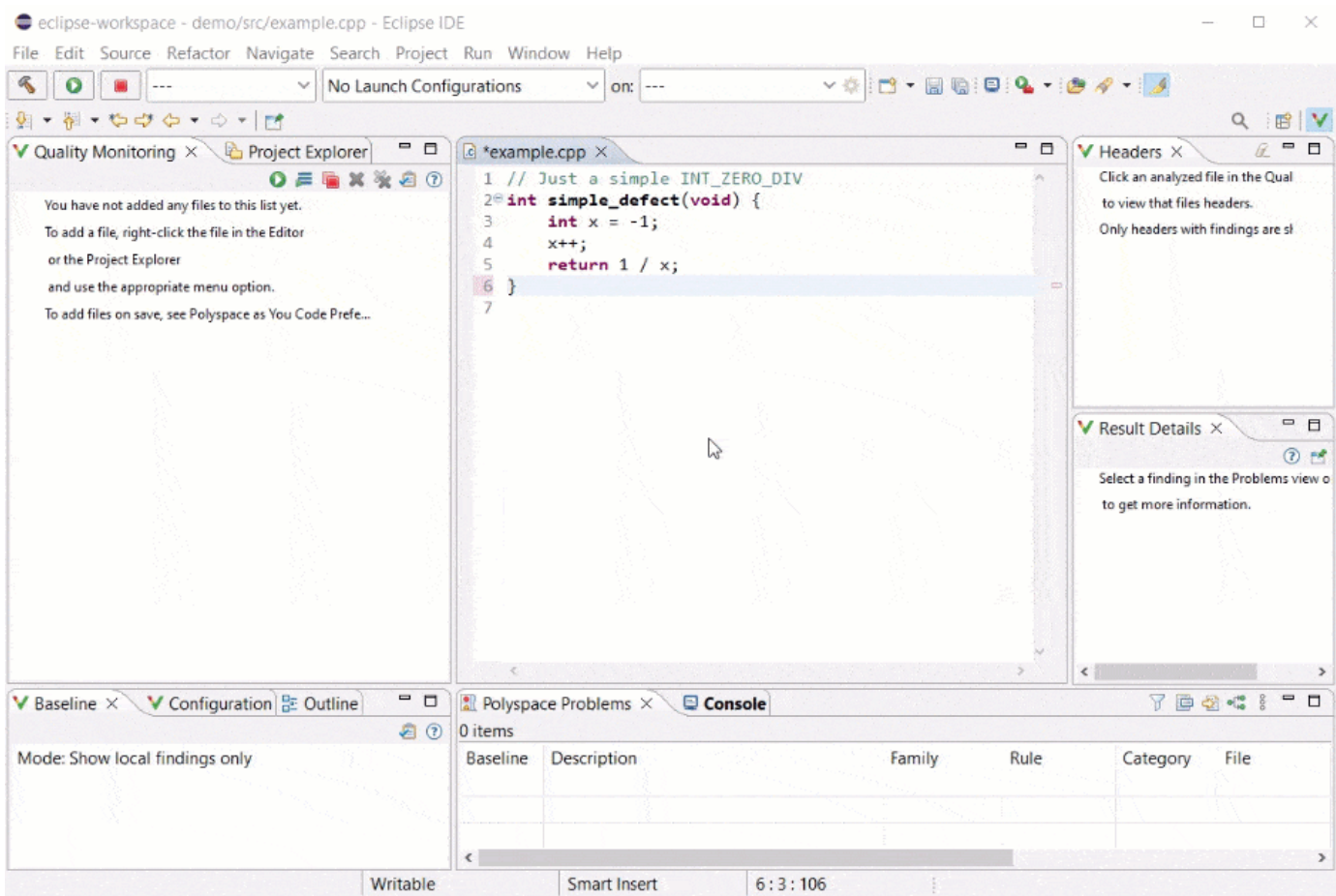
## Automatically Add Files to Quality Monitoring List

Polyspace has a **Quality Monitoring** list that keeps track of edited files. Polyspace can analyze all files in the **Quality Monitoring** list at the same time. You can choose to have Polyspace automatically add files to the **Quality Monitoring** list or manually add files.

- To automatically add files to the **Quality Monitoring** list when you save a file, select **Add to Quality Monitoring list on save**.
- To manually add files to the **Quality Monitoring** list, right-click inside the file editor and select **Add file to the Polyspace Quality Monitoring list**.

For this example, automatically add files to the quality monitoring list. Click the preferences icon  in the **Configuration** pane and select the check box for **Add to Quality Monitoring list on save**.


## Automatically Add Files to Quality Monitoring List




## Analyze Files Automatically

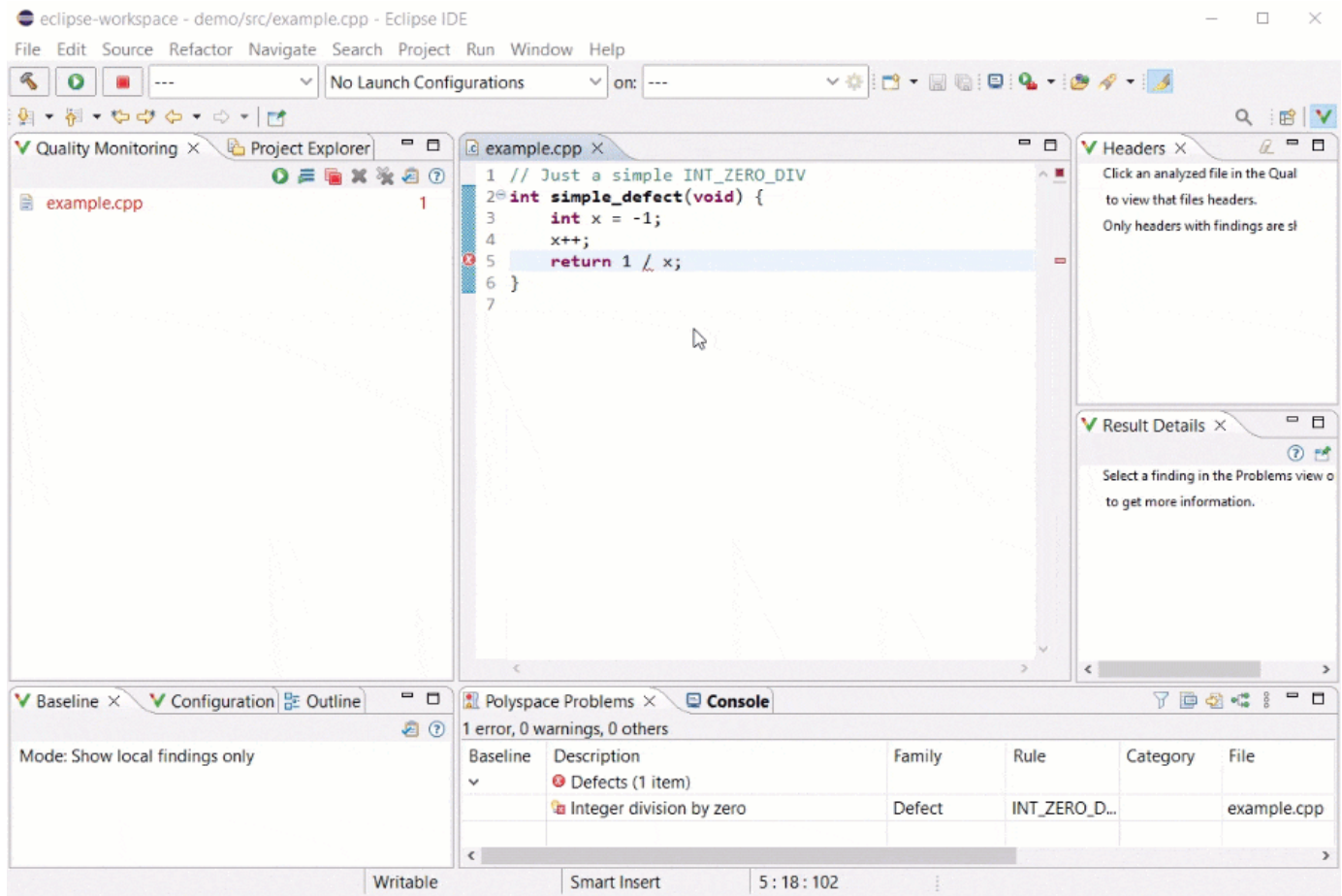
Polyspace can automatically run an analysis of any file in the **Quality Monitoring** list when you save the file. Alternatively, you can manually run analysis of individual files or a group of files.

You can manually trigger an analysis in these ways:

- Select a file in the **Quality Monitoring** list and then click the Run Polyspace Analysis button .
- Right-click inside the file editor and select **Run Polyspace Analysis**.


Turn on automatic analysis of files. Click the preferences icon  in the **Configuration** pane and select the check box for **Start analysis on save**.

## Automatically Run Analysis on Save



## Configure Polyspace Checkers

Polyspace checks for a default set of checkers. You can customize this set of standards and defects to check for specific certification standards. Use the preferences option **Checkers File** to set the checkers XML file for your project.

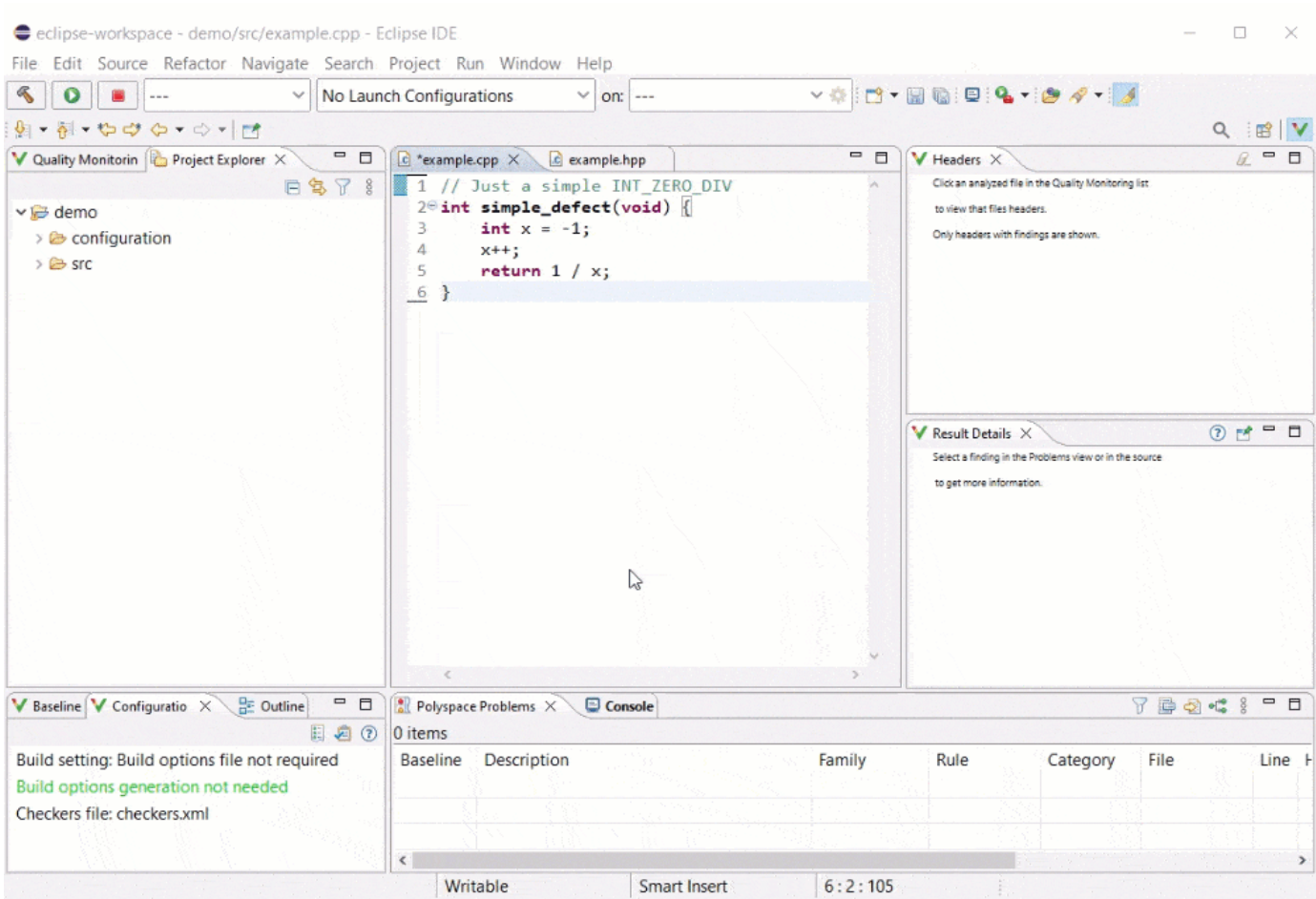
Turn on the AUTOSAR C++14 checkers for your project. In the **Configuration** pane, click the **Configure Checkers** icon  to open the **Checkers selection** window.

You can create, save, and open saved checkers selection files. For more information, see “Configure Checkers for Polyspace as You Code in Eclipse”.

Click **AUTOSAR C++14** on the left pane of the **Checkers Selection** window. In the right pane, select the **All** check box to enable all of the AUTOSAR C++14 rules. Click **Save Changes** in the top right corner to save your checker selection and close the window to return to the Eclipse application.




## Configure Polyspace Checkers

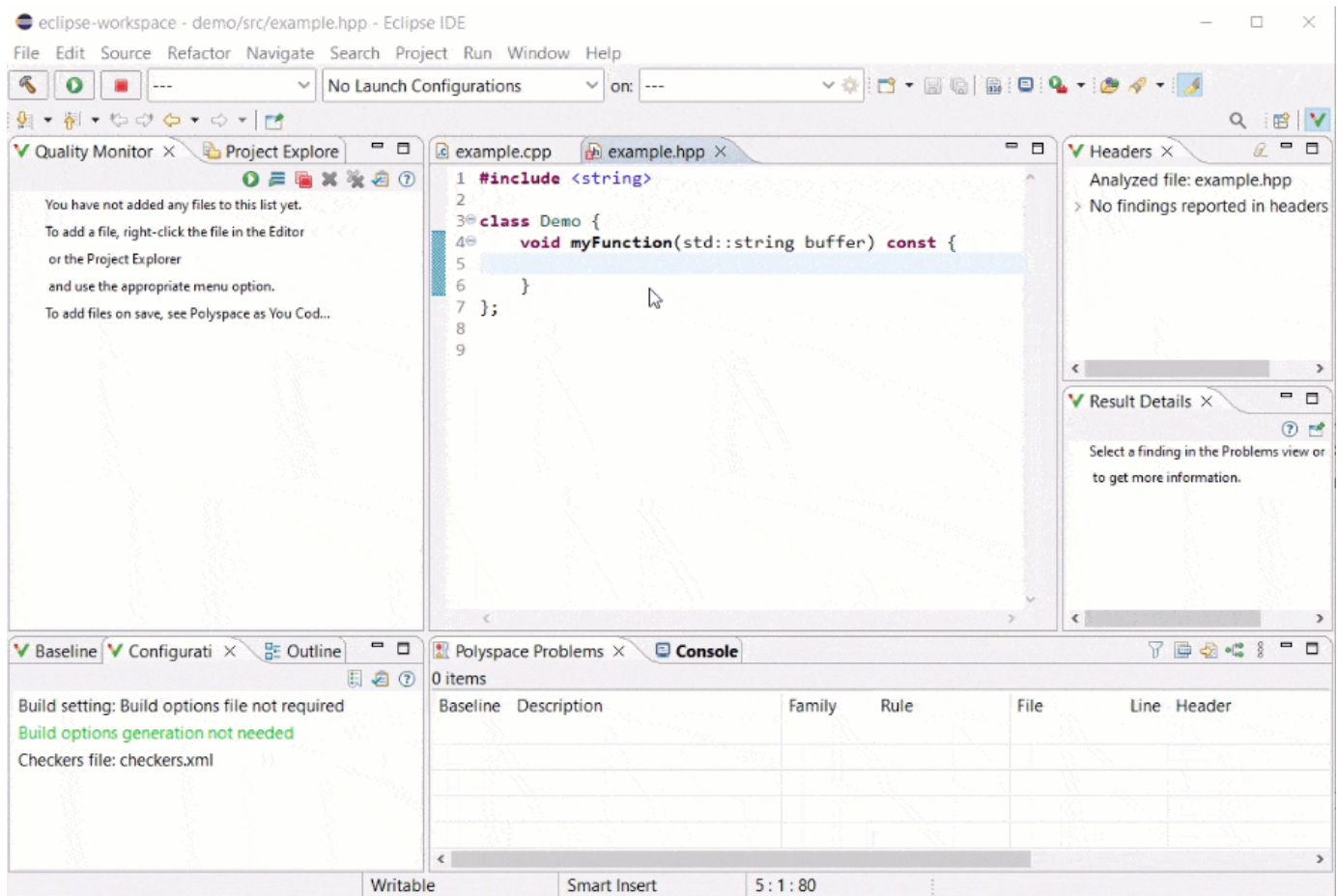


## View Header File Findings

You can run an analysis on header files as if they were source files by adding them to the **Quality Monitoring** list and running an analysis.

Manually add the `example.hpp` header file to the **Quality Monitoring** list and initiate an analysis by clicking the Run Polyspace Analysis button .


## View Header File Findings

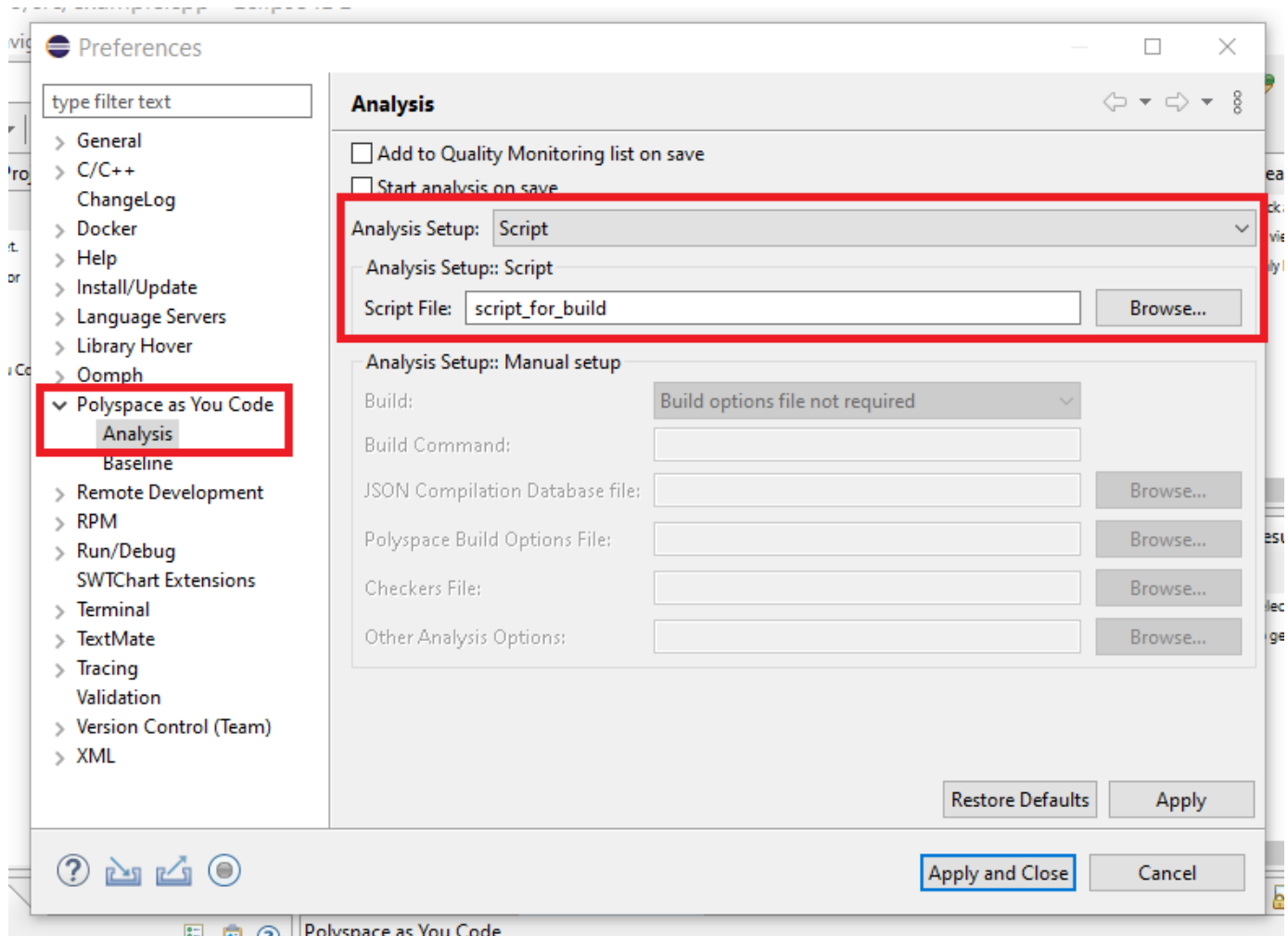


## Configure Analysis Script

You can provide a script to configure and run your Polyspace analysis in Polyspace as You Code.

For this tutorial, you do not set a script.

If you have a script, click the preferences icon  in the **Configuration** pane and select **Analysis**. For the option **Analysis Setup**, select **Script**. Provide the path to your script under the option **Script file** by clicking **Browse** and opening your script.




Once you configure the Polyspace as You Code plugin, you are ready to run an analysis and review your results.

## Run and Review Results in Polyspace as You Code for Eclipse

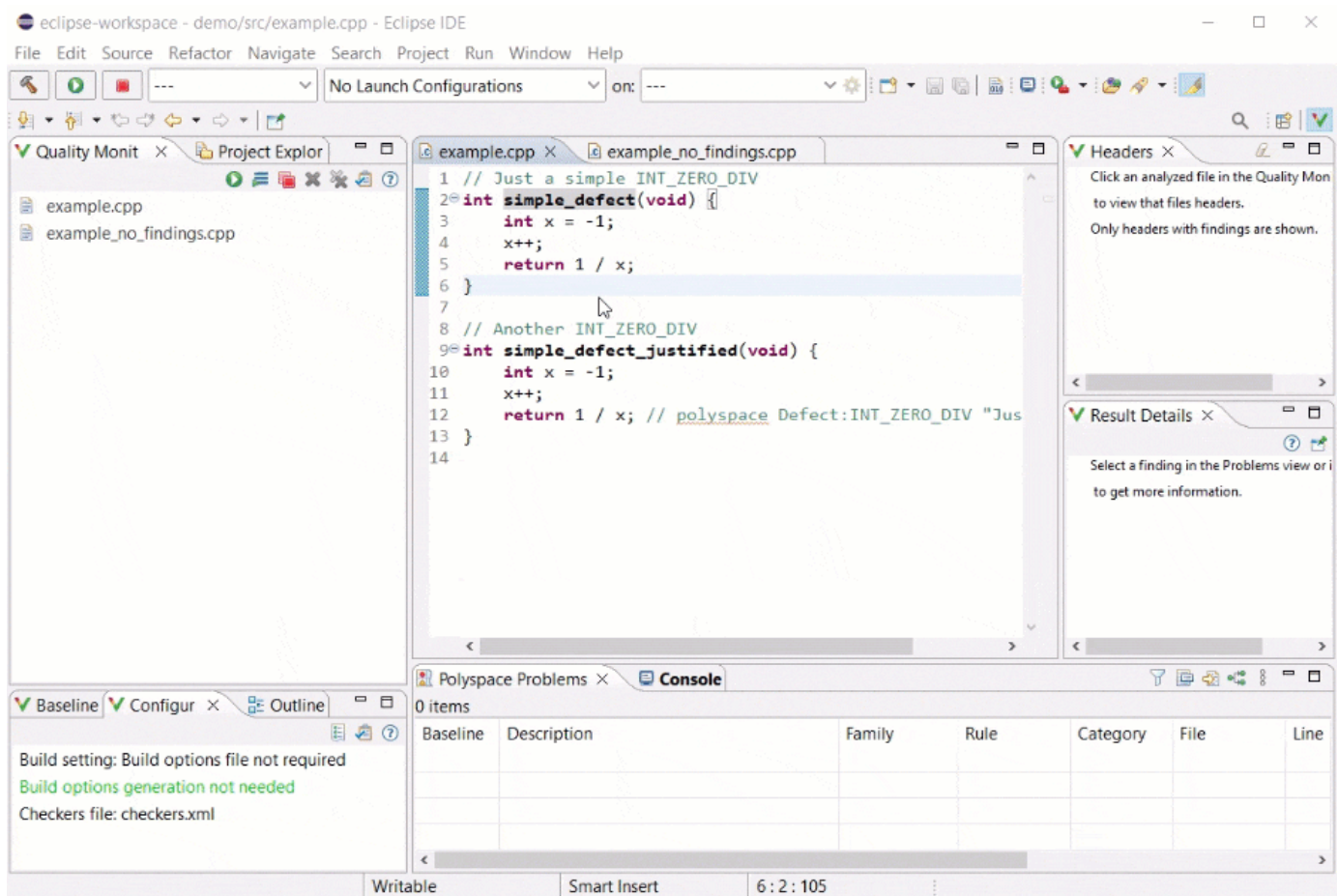
After you configure the Polyspace as You Code plugin, you are ready to run an analysis and fix or justify any findings. Before continuing this guide, confirm you have added the files `example.hpp` and `example_source.cpp` to the **Quality Monitoring** list.

### Run Polyspace as You Code Analysis in Eclipse

You can run an analysis in these ways:

- If you select **Start analysis on save**, save your edited file to run an analysis.
- Right-click in the file editor with your file open and select **Run Polyspace Analysis**.
- Manually add your file to the **Quality Monitoring** list and click the Run Polyspace Analysis button  in the **Quality Monitoring** pane to analyze your file.

#### Manually Run Analysis




## View, Fix, or Justify Findings

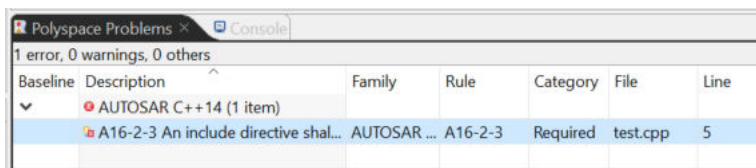
After Polyspace runs the analysis, the filename in the **Quality Monitoring** pane is green if there are no findings and red if there are findings. A number to the right of the filename indicates the number of findings.

View a list of your findings in the **Polyspace Problems** pane, which opens by default when you open the Polyspace as You Code perspective in Eclipse. If the **Polyspace Problems** pane is not open, reset the perspective by clicking **Window > Perspective > Reset Perspective**.

Use the filter in the **Polyspace Problems** pane to search for specific findings. Each finding contains a filename and line number to indicate where the finding exists in your code.

After you run an analysis on the example code:

- 1 Open the results in the **Polyspace Problems** pane.
- 2 Click the filter icon  and create a new filter to search for the text A16-2-3. There is one result on line five.



Baseline	Description	Family	Rule	Category	File	Line
	AUTOSAR C++14 (1 item)					
	A16-2-3 An include directive sha...	AUTOSAR ...	A16-2-3	Required	test.cpp	5

- 3 Click the finding in the **Polyspace Problems** pane to open the file that contains the finding.

Each finding is noted in the file with a red highlight. Click the highlighted sections of code to see a list of each finding associated with the problem in the **Result Details** pane.

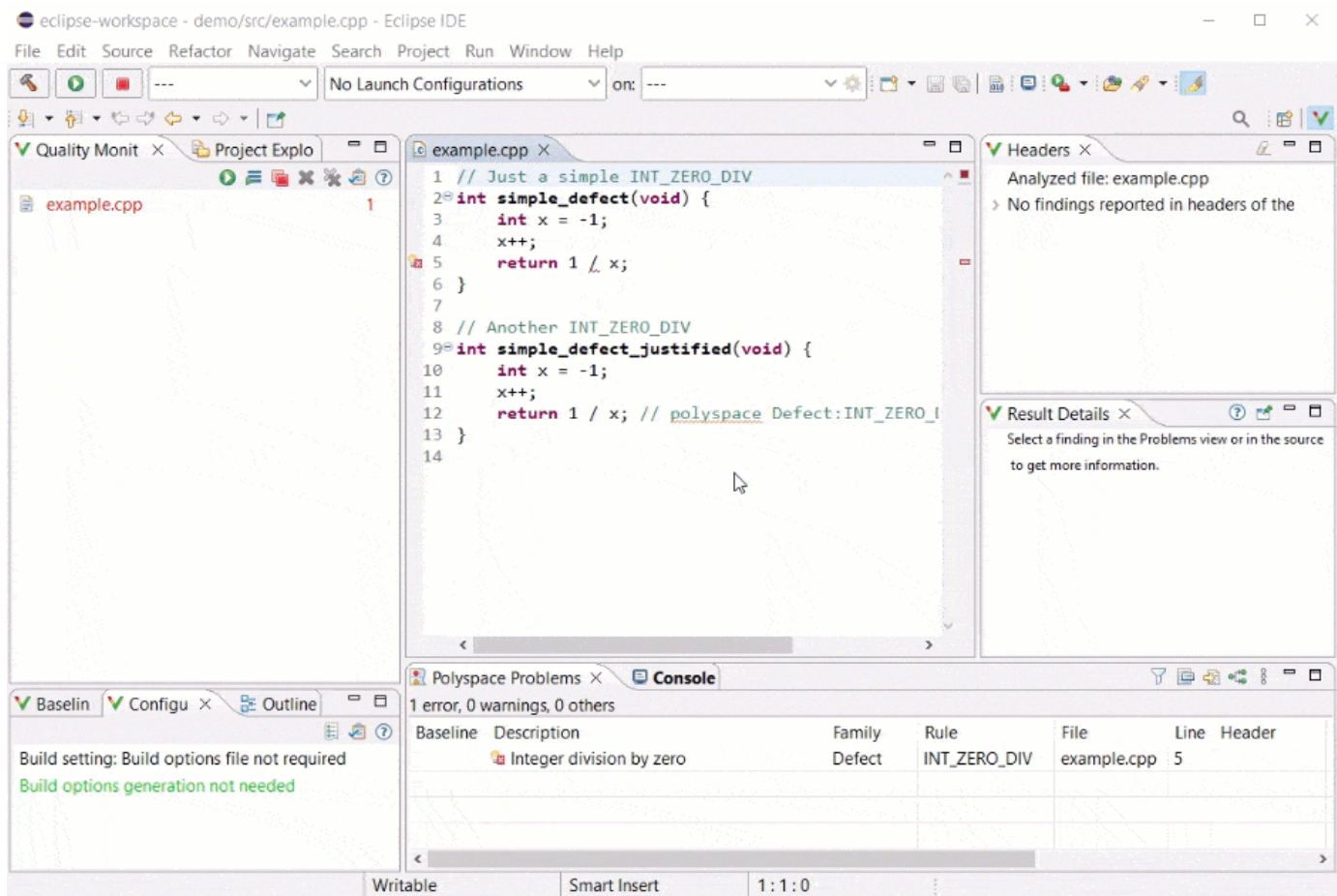
- 4 In the example code, add this `#include` statement on line two:

```
#include <cstdint>
```

- 5 Save this change and run another analysis. The A16-2-3 error on line five is no longer present.

If you set up automatic analysis, you can find and fix findings during the code authoring process. Each time you save changes, a Polyspace analysis begins in the background and displays any new findings or removes fixed findings from the list.

## View and Fix Findings

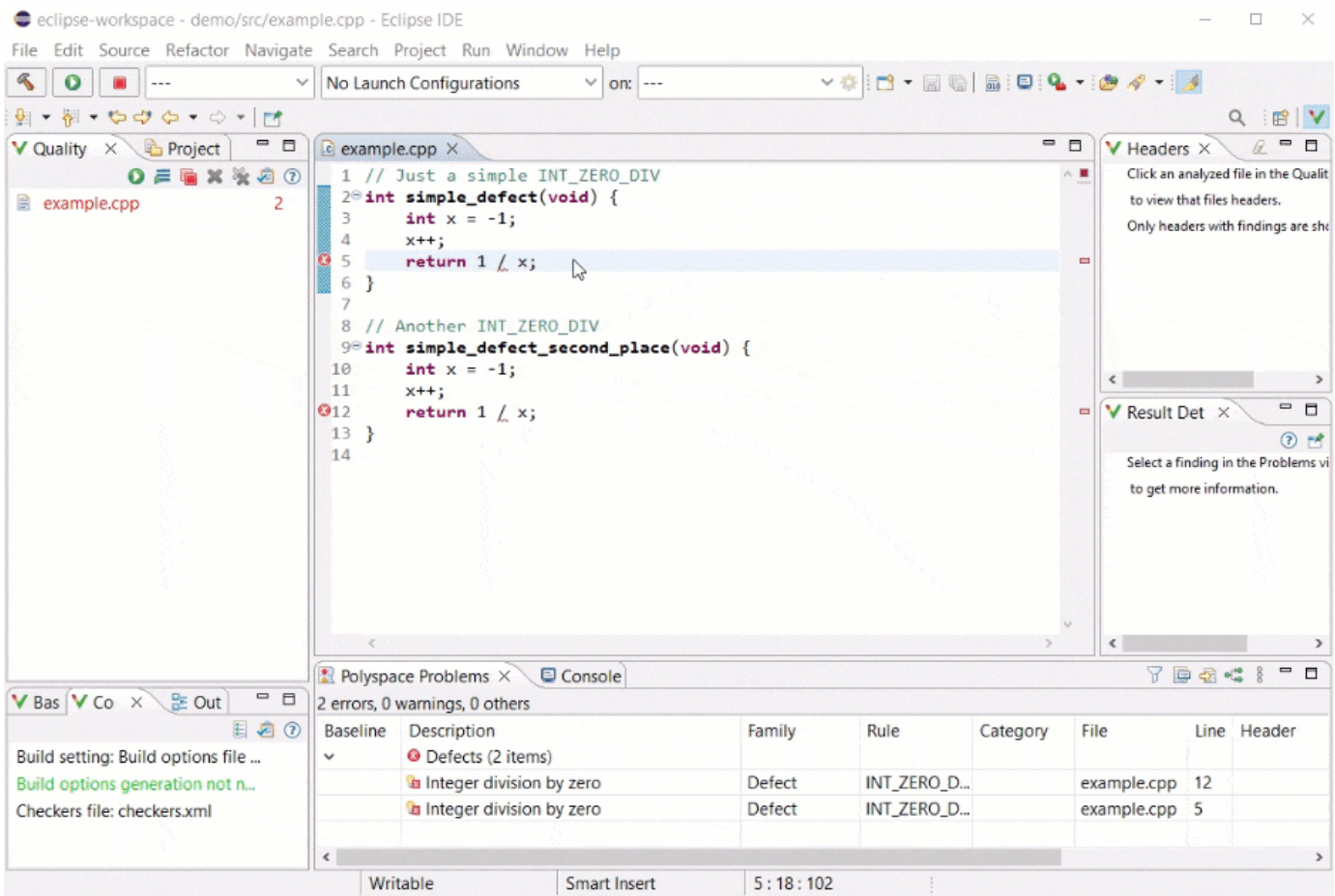


## Justify Individual Findings

You can justify a finding from the **Polyspace Problems** pane or from the location of the finding in the code.

Right-click a finding in the **Polyspace Problems** pane and select **Polyspace**. Select the appropriate justification from the menu. This adds a comment to your code, which you can amend. Adding a justification removes the finding from the **Polyspace Problems** pane. To show the finding again, remove the justification comment and perform an analysis.

## Justify Findings



## Provide Justification Catalog

You can add a catalog of predefined justifications to Polyspace as You Code. The justification catalog allows you to select a prewritten justification instead of manually entering each justification comment. The catalog must be in JSON format. If you do not already have a justification catalog, use this example JSON file.

example\_catalog.json

```

{
  "justifications": [
    {
      "family": "Defect",
      "acronym": "INT_ZERO_DIV",
      "comment": "This is my justification for division"
    },
    {
      "family": "Defect",
      "acronym": "INT_ZERO_DIV",
      "comment": "Alternative justification for division"
    }
  ]
}

```

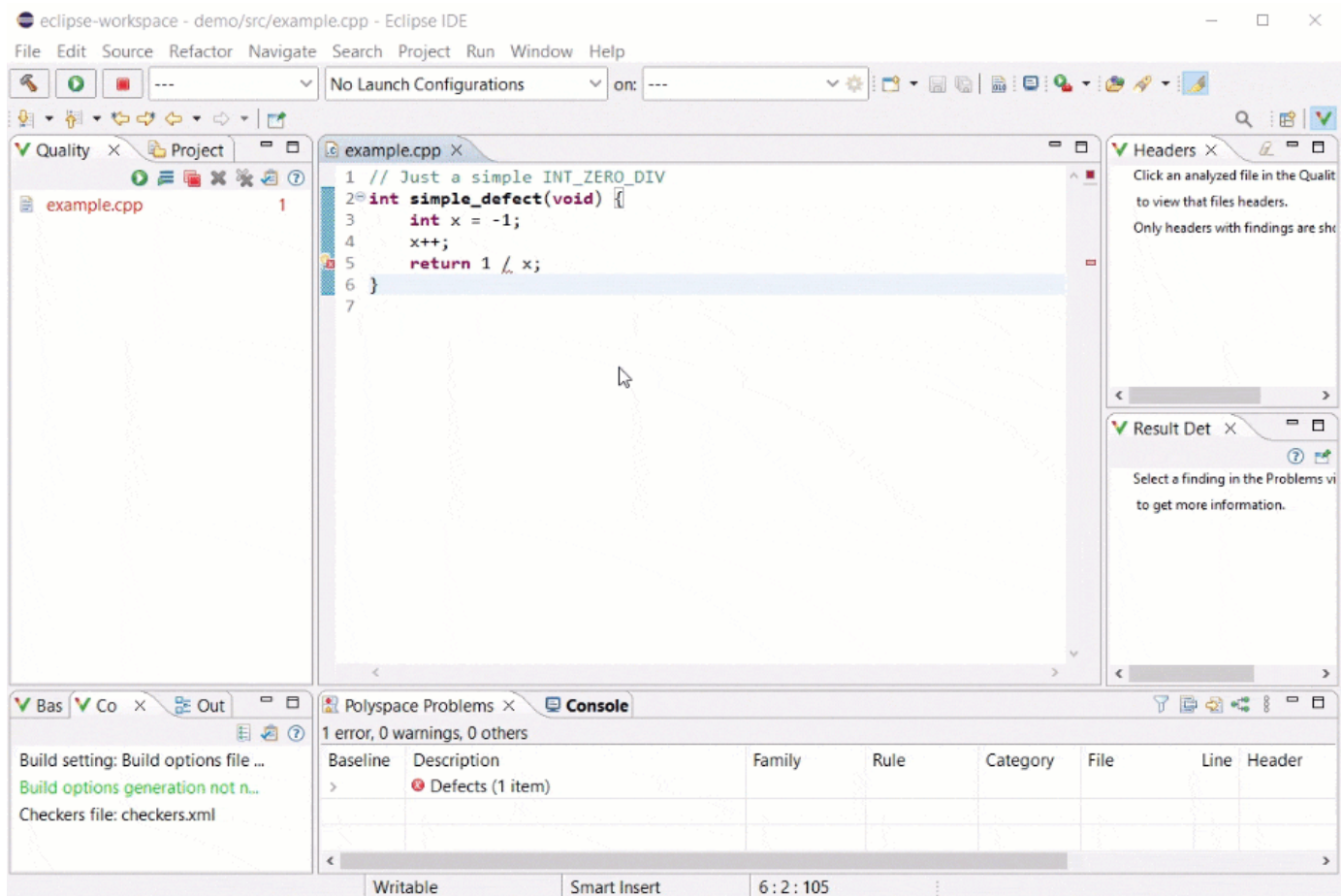
```

    ]
}

```

Open the Eclipse **Preferences** and select **Polyspace as You Code**. Next to the **Justification catalog** text box, click **Browse** and navigate to your justification catalog.

### Provide Justification Catalog



### View Header Findings

When you analyze source files, you see any header file findings in the **Polyspace Problems** pane. If a header contains any findings, the findings are noted with a red **H** next to the source file in the **Quality Monitoring** list.

Double-click a header finding in the **Polyspace Problems** pane to go to the header file that contains the finding. The source file analysis that caused the header finding is listed in the **Header** column of the **Polyspace Problems** pane.



## View Header Findings

The screenshot shows the Eclipse IDE interface with the following components:

- Main Editor:** Displays the source code for `example.cpp`. Line 7 is highlighted with a red squiggly line, indicating a defect: `return 1 / x;`.
- Polyspace Problems View:** Shows a table with one error. The table has columns: Baseline, Description, Family, Rule, File, Line, Header, and Category.

Baseline	Description	Family	Rule	File	Line	Header	Category
	Integer division by zero	Defect	INT_ZERO_DIV	example.cpp	7		


Other visible panels include: Quality Monitors, Project Explorer, Headers (with instructions to click an analyzed file), Result Details (with instructions to select a finding), and Baseline/Config/Outline (with build settings).

## Configure and Download Baseline with Polyspace as You Code in Eclipse

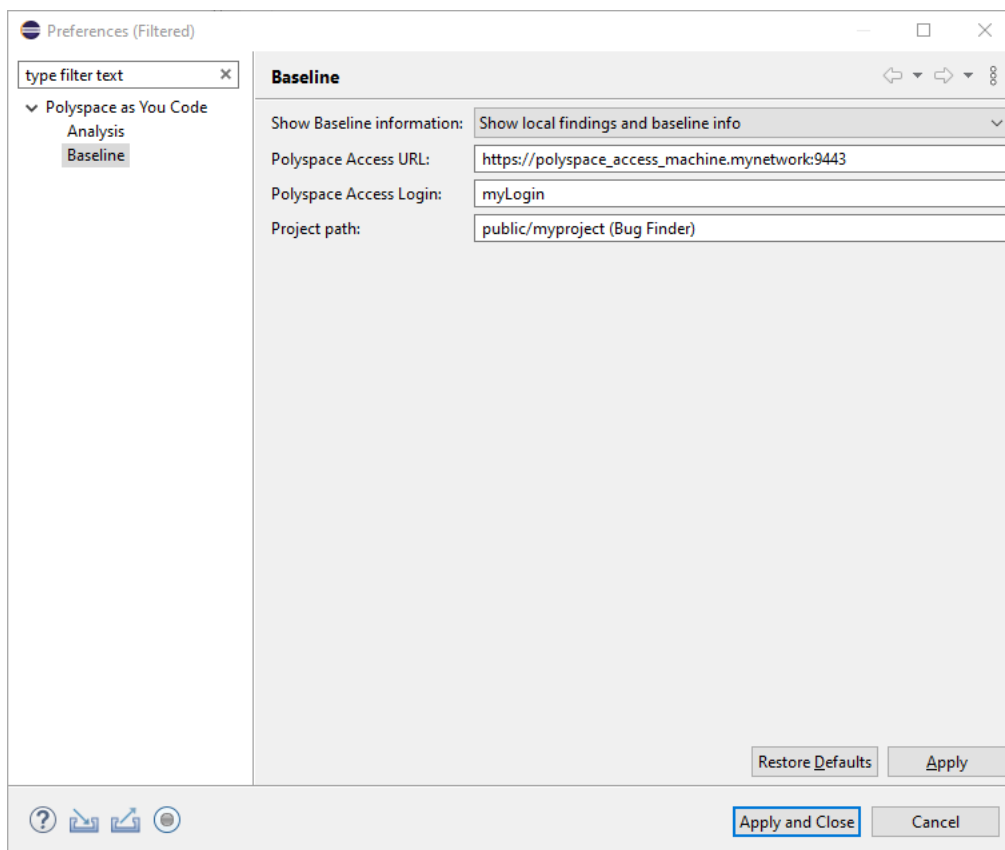
For more efficient bug fixing, you can create a baseline using Polyspace Access results. Download the baseline and use it to compare your Polyspace as You Code results and focus on new or unreviewed results.

In order to configure baseline results in Polyspace as You Code, you must have a Polyspace Access server login name and password along with an uploaded project result. The project must contain results from an analysis of the same files you are analyzing in Polyspace as You Code.

### Configure Baseline


Configure a baseline using your Polyspace Access server information. Click the preferences icon  in the **Baseline** pane to open your baseline preferences.

Select **Show local findings and baseline info** from the **Show Baseline information** menu. Then, enter your login, the server URL, and the path for the project for which you want to create a baseline.

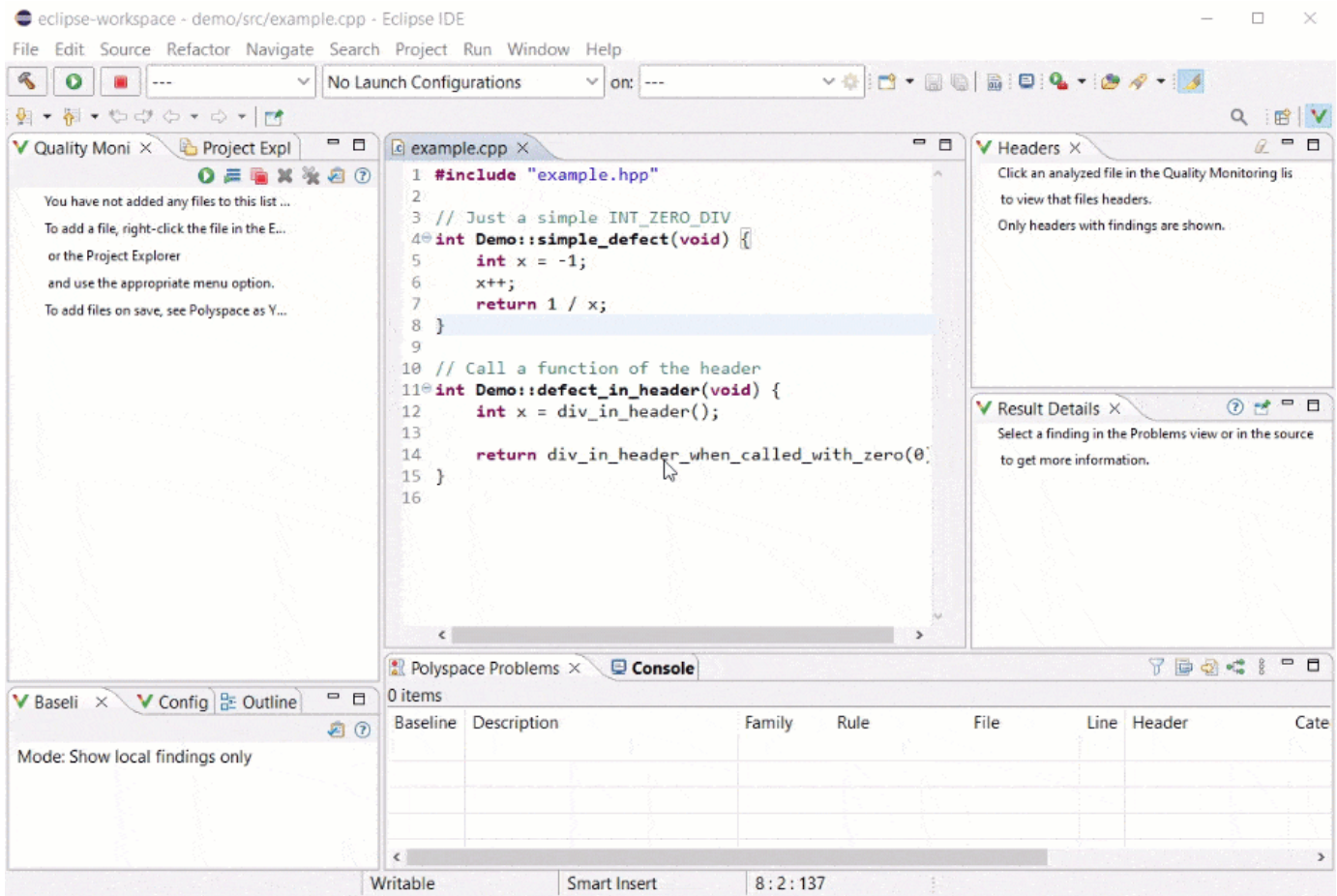


## Download Baseline

To keep using the most up-to-date baseline information, make sure that you periodically update your baseline by downloading the latest information from Polyspace Access.


After you set your baseline preferences, **Baseline not downloaded** is shown in red in the **Baseline** pane. Click the download baseline icon  to download the baseline. Click this icon any time you want to update your baseline with the latest information.

## Download Baseline

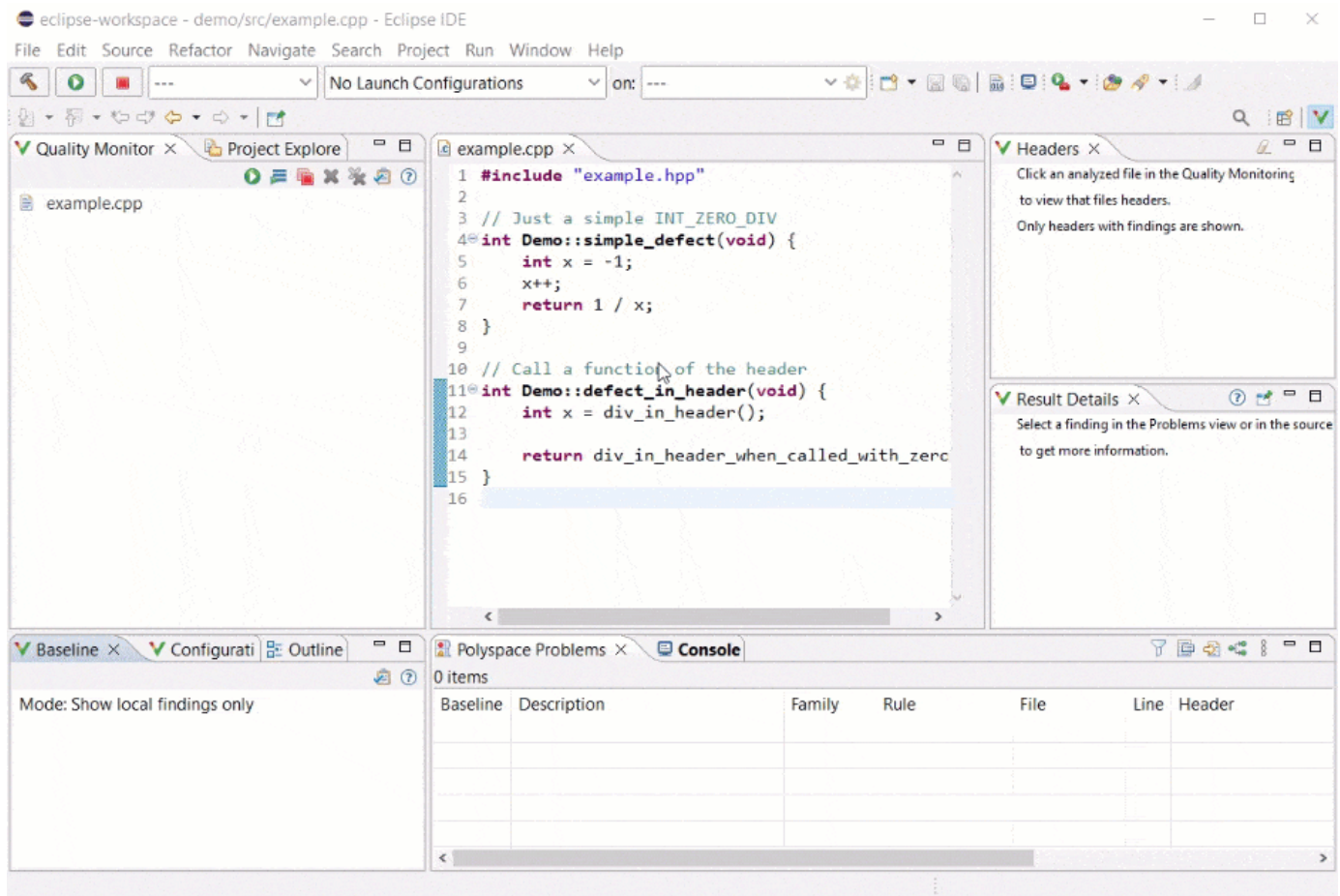


## Show New Findings and Compare Results

It can be more efficient to review only new results compared to the baseline.

To view only new findings, click the preferences icon  in the **Baseline** pane of the **Polyspace** sidebar. In the **Show Baseline Information** menu, select **Show new findings only**.

## Show New Results



# Deploy Polyspace Bug Finder

---

## Polyspace Products and Software Development Workflows

### In this section...

“Using Polyspace Products in Software Development” on page 6-2

“Coordinating Pre-Submit and Post-Submit Usage of Polyspace” on page 6-3

“Polyspace Products for Ada Code” on page 6-4

Polyspace products use static analysis to check code for run-time errors, coding standard violations, security vulnerabilities, and other issues:

- Polyspace Code Prover can cover all possible execution paths through a program and track data flow along these paths following certain mathematical rules. The exhaustive control and data flow analysis can complement dynamic testing and expose potential run-time errors that might not be otherwise found in regular robustness testing.
- Polyspace Bug Finder can scan a program for more obvious defects, security vulnerabilities, coding standard violations and other issues that potentially lead to run-time errors or unexpected results.

### Using Polyspace Products in Software Development

The Polyspace suite of products supports all phases of a software development process:

- *Prior to code submission:*

Developers can run the Polyspace desktop or IDE-focused products to check their code during development or right before submission to meet predefined quality goals.

The products can be integrated into IDEs such as Visual Studio Code, Visual Studio, or Eclipse, or run with scripts during compilation. The analysis results can be reviewed in the IDEs or in the graphical user interface of the desktop products.

Polyspace provides the following products for desktop usage. These products are meant to run on complete projects or smaller code modules (up to a single source file).

- **Polyspace Bug Finder** to check code for semantic errors that a compiler cannot detect (such as use of = instead of ==), concurrency issues, security vulnerabilities and other defects in C and C++ source code.
- **Polyspace Code Prover** to perform a much deeper check and prove absence of overflow, divide-by-zero, out-of-bounds array access and other run-time errors in C and C++ source code.
- *After code submission:*

The Polyspace server products can run automatically on newly committed code as a build step in a continuous integration process (using tools such as Jenkins). The analysis runs on a server and the results are uploaded to a web interface for collaborative review.

Polyspace provides these products for server usage:

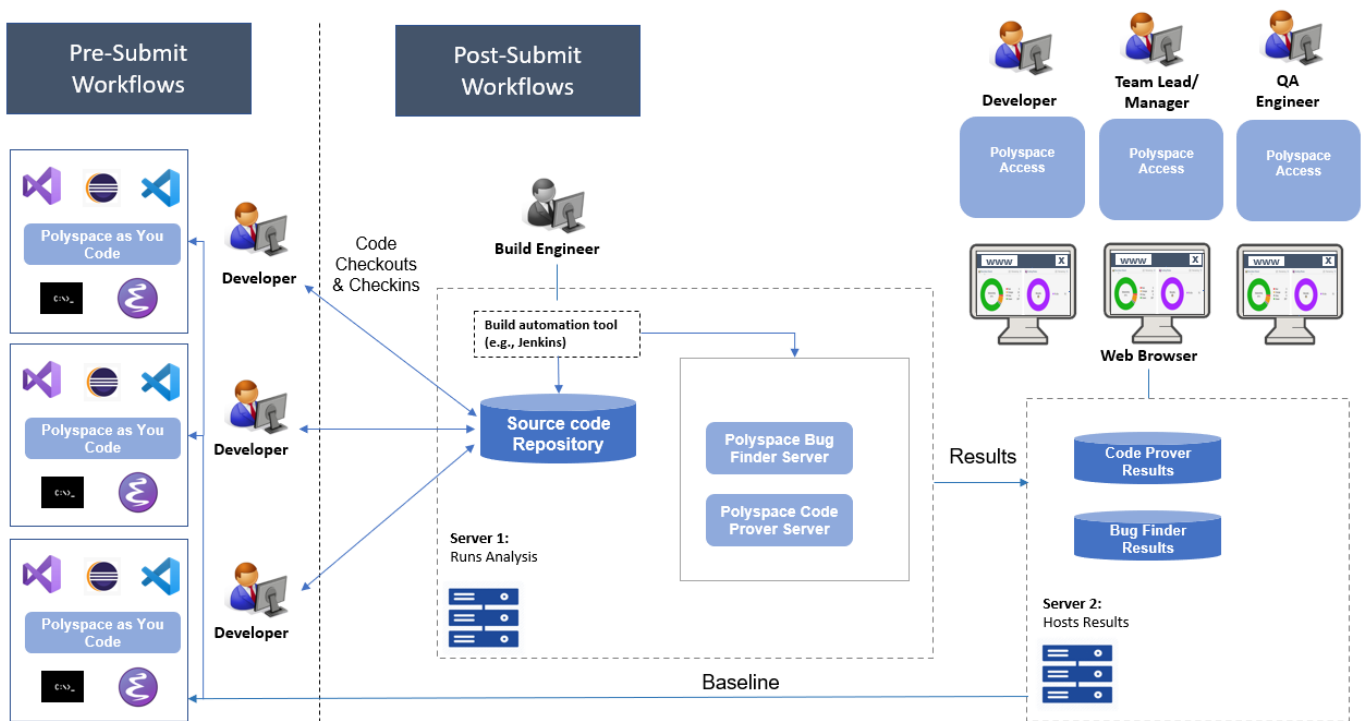
- **Polyspace Bug Finder Server** to run Bug Finder automatically on a server and upload the results to a web interface for review, and **Polyspace Access** to review the uploaded results.

- **Polyspace Code Prover Server** to run Code Prover automatically on a server and upload the results to a web interface for review, and **Polyspace Access** to review the uploaded results.

Typically, Polyspace Bug Finder Server (or Polyspace Code Prover Server) runs on a few build servers and checks newly committed code as part of software build and testing. Each reviewer (developer, quality assurance engineer or development manager) has a Polyspace Access license to review the uploaded analysis results.

In addition, if developers have access to **Polyspace Access** for web review of post-submission results, they can also install **Polyspace as You Code** in their IDEs for pre-submission analysis. When installed as an IDE extension, Polyspace as You Code performs a file-scope Bug Finder-like analysis and provides near-instant feedback to developers while coding.

This diagram shows *one possible deployment* of Polyspace products:



Note: Depending on the specifications, the same computer can serve as both Server 1 and Server 2.

When you use both the desktop and server products, your pre-submission workflow can transition smoothly to the post-submission workflow.

## Coordinating Pre-Submit and Post-Submit Usage of Polyspace

When you run more than one Polyspace products at separate stages in your software development workflow, the later runs can benefit from the earlier usage, and vice versa. In particular:

- Developers using Polyspace as You Code in their IDEs can easily fix defects and coding standard violations that can be found and resolved within a single file. A later Polyspace Bug Finder Server analysis after code submission no longer shows these issues.

- The results of a Polyspace Bug Finder Server analysis can act as a baseline for Polyspace as You Code runs. Developers using the latest Polyspace Bug Finder Server result as baseline for their IDE runs can focus only on issues that result from their code changes.

### Polyspace Products for Ada Code

Polyspace provides these products for verifying Ada code:

- **Polyspace Client™ for Ada** to check Ada code for run-time errors on a desktop.
- **Polyspace Server for Ada** to check Ada code for run-time errors on a server.

You can either use the desktop product to run the analysis on your desktop, or a combination of the desktop and server products to run the analysis on a server. The analysis results are downloaded to your desktop for review.

If you have a Polyspace Code Prover Access license and have set up the web interface of Polyspace Code Prover Access, you can upload each individual Ada result from the Ada desktop products to the web interface for collaborative review.

See also <https://www.mathworks.com/products/polyspace-ada.html>.

### See Also

#### Related Examples

- “Install Polyspace Desktop Products”
- “Install Polyspace Server and Access Products”
- “Install Polyspace Products in IDEs”
- “Differences Between Polyspace Bug Finder and Polyspace Code Prover” on page 6-5



## Differences Between Polyspace Bug Finder and Polyspace Code Prover

Polyspace Bug Finder and Polyspace Code Prover detect run-time errors through static analysis. Though the products have a similar user interface and the mathematics underlying the analysis can sometimes be the same, the goals of the two products are different.

Bug Finder (or Polyspace as You Code, which performs a single-file analysis similar to Bug Finder) quickly analyzes your code and detects many types of defects. Code Prover checks *every* operation in your code for a set of possible run-time errors and tries to prove the absence of the error for all execution paths<sup>1</sup>. For instance, for *every* division in your code, a Code Prover analysis tries to prove that the denominator cannot be zero. Bug Finder does not perform such exhaustive verification. For instance, Bug Finder also checks for a division by zero error, but it might not find all operations that can cause the error.

The two products involve differences in setup, analysis and results review, because of this difference in objectives. In the following sections, we highlight the primary differences between a Bug Finder and a Code Prover analysis (also known as verification). Depending on your requirements, you can incorporate one or both kinds of analyses at appropriate points in your software development life cycle.

### How Bug Finder and Code Prover Complement Each Other

- “Overview” on page 6-5
- “Faster Analysis with Bug Finder” on page 6-6
- “More Exhaustive Verification with Code Prover” on page 6-6
- “More Specific Defect Types with Bug Finder” on page 6-7
- “Easier Setup Process with Bug Finder” on page 6-7
- “Fewer Runs for Clean Code with Bug Finder” on page 6-8
- “Results in Real Time with Bug Finder” on page 6-8
- “More Rigorous Data and Control Flow Analysis with Code Prover” on page 6-8
- “Few False Positives with Bug Finder” on page 6-9
- “Zero False Negatives with Code Prover” on page 6-9
- “Coding Rule Support in Bug Finder” on page 6-10

#### Overview

Use both Bug Finder and Code Prover regularly in your development process. The products provide a unique set of capabilities and complement each other. For possible ways to use the products together, see “Workflow Using Both Polyspace Bug Finder and Polyspace Code Prover” on page 6-11.

This table provides an overview of how the products complement each other. For details, see the sections below.

<sup>1</sup> For each operation in your code, Code Prover considers all execution paths leading to the operation that do not have a previous error. If an execution path contains an error prior to the operation, Code Prover does not consider it. See “Code Prover Analysis Following Red and Orange Checks” (Polyspace Code Prover).

Feature	Bug Finder	Code Prover
Number of checkers	300+ checkers for defects	30 checks for critical run-time errors and 4 checks on global variable usage
Depth of analysis	Fast. For instance: <ul style="list-style-type: none"> <li>• Faster analysis.</li> <li>• Easier set up and review.</li> <li>• Fewer runs for clean code.</li> <li>• Results in real time.</li> </ul>	Exhaustive. For instance: <ul style="list-style-type: none"> <li>• All operations of a type checked for certain critical errors.</li> <li>• More rigorous data and control flow analysis.</li> </ul>
Reporting criteria	Probable defects	Proven findings
Bug finding criteria	Few false positives	Zero false negatives

### Faster Analysis with Bug Finder

How much faster the Bug Finder analysis is depends on the size of the application. The Bug Finder analysis time increases linearly with the size of the application. The Code Prover verification time increases at a rate faster than linear.

One possible workflow is to run Code Prover to analyze modules or libraries for robustness against certain errors and run Bug Finder at integration stage. Bug Finder analysis on large code bases can be completed in a much shorter time, and also find integration defects such as **Declaration mismatch** and **Data race**.

### More Exhaustive Verification with Code Prover

Code Prover tries to prove the absence of:

- **Division by Zero** error on *every* division or modulus operation
- **Out of Bounds Array Index** error on *every* array access
- **Non-initialized Variable** error on *every* variable read
- **Overflow** error on *every* operation that can overflow

and so on.

For each operation:

- If Code Prover can prove the absence of the error for all execution paths, it highlights the operation in green.
- If Code Prover can prove the presence of a definite error for all execution paths, it highlights the operation in red.
- If Code Prover cannot prove the absence of an error or presence of a definite error, it highlights the operation in orange. This small percentage of orange checks indicate operations that you must review carefully, through visual inspection or testing. The orange checks often indicate hidden vulnerabilities. For instance, the operation might be safe in the current context but fail when reused in another context.

You can use information provided in the Polyspace user interface to diagnose the checks. See “More Rigorous Data and Control Flow Analysis with Code Prover” on page 6-8. You can also

provide contextual information to reduce unproven code even further, for instance, constrain input ranges externally.

Bug Finder does not aim for exhaustive analysis. It tries to detect as many bugs as possible and reduce false positives. For critical software components, running a bug finding tool is not sufficient because despite fixing all defects found in the analysis, you can still have errors during code execution (false negatives). After running Code Prover on your code and addressing the issues found, you can expect the quality of your code to be much higher. See “Zero False Negatives with Code Prover” on page 6-9.

### **More Specific Defect Types with Bug Finder**

Code Prover checks for types of run-time errors where it is possible to mathematically prove the absence of the error. In addition to detecting errors whose absence can be mathematically proven, Bug Finder also detects other defects.

For instance, the statement `if (a=b)` is semantically correct according to the C language standard, but often indicates an unintended assignment. Bug Finder detects such unintended operations. Although Code Prover does not detect such unintended operations, it can detect if an unintended operation causes other run-time errors.

Examples of defects detected by Bug Finder but not by Code Prover include good practice defects, resource management defects, some programming defects, security defects, and defects in C++ object oriented design.

For more information, see:

- “Defects”: List of defects that Bug Finder can detect.
- “Run-Time Checks” (Polyspace Code Prover): List of run-time errors that Code Prover can detect.

### **Easier Setup Process with Bug Finder**

Even if your code builds successfully in your compilation toolchain, it can fail in the compilation phase of a Code Prover verification. The strict compilation in Code Prover is related to its ability to prove the absence of certain run-time errors.

- Code Prover strictly follows the ANSI<sup>®</sup> C99 Standard, unless you explicitly use analysis options that emulate your compiler.

To allow deviations from the ANSI C99 Standard, you must use the “Target and Compiler” options. If you create a Polyspace project from your build system, the options are automatically set.

- Code Prover does not allow linking errors that common compilers can permit.

Though your compiler permits linking errors such as mismatch in function signature between compilation units, to avoid unexpected behavior at run time, you must fix the errors.

For more information, see “Troubleshoot Compilation and Linking Errors” (Polyspace Code Prover).

Bug Finder is less strict about certain compilation errors. Linking errors, such as mismatch in function signature between different compilation units, can stop a Code Prover verification but not a Bug Finder analysis. Therefore, you can run a Bug Finder analysis with less setup effort. In Bug Finder, linking errors are often reported as a defect after the analysis is complete.

### Fewer Runs for Clean Code with Bug Finder

To guarantee absence of certain run-time errors, Code Prover follows strict rules once it detects a run-time error in an operation. Once a run-time error occurs, the state of your program is ill-defined and Code Prover cannot prove the absence of errors in subsequent code. Therefore:

- If Code Prover proves a definite error and displays a red check, it does not verify the remaining code in the same block.

Exceptions include checks such as **Overflow**, where the analysis continues with the result of overflow either truncated or wrapped around.

- If Code Prover suspects the presence of an error and displays an orange check, it eliminates the path containing the error from consideration. For instance, if Code Prover detects a **Division by Zero** error in the operation  $1/x$ , in the subsequent operation on  $x$  in that block,  $x$  cannot be zero.
- If Code Prover detects that a code block is unreachable and displays a gray check, it does not detect errors in that block.

For more information, see “Code Prover Analysis Following Red and Orange Checks” (Polyspace Code Prover).

Therefore, once you fix red and gray checks and rerun verification, you can find more issues. You need to run verification several times and fix issues each time for completely clean code. The situation is similar to dynamic testing. In dynamic testing, once you fix a failure at a certain point in the code, you can uncover a new failure in subsequent code.

Bug Finder does not stop the entire analysis in a block after it finds a defect in that block. Even with Bug Finder, you might have to run analysis several times to obtain completely clean code. However, the number of runs required is fewer than Code Prover.

### Results in Real Time with Bug Finder

Bug Finder shows some analysis results while the analysis is still running. You do not have to wait until the end of the analysis to review the results.

Code Prover shows results only after the end of the verification. Once Bug Finder finds a defect, it can display the defect. Code Prover has to prove the absence of errors on all execution paths. Therefore, it cannot display results during analysis.

### More Rigorous Data and Control Flow Analysis with Code Prover

For each operation in your code, Code Prover provides:

- Tooltips showing the range of values of each variable in the operation.

For a pointer, the tooltips show the variable that the pointer points to, along with the variable values.

- Graphical representation of the function call sequence that leads to the operation.

By using this range information and call graph, you can easily navigate the function call hierarchy and understand how a variable acquires values that lead to an error. For instance, for an **Out of Bounds Array Index** error, you can find where the index variable is first assigned values that lead to the error.

When reviewing a result in Bug Finder, you also have supporting information to understand the root cause of a defect. For instance, you have a traceback from where Bug Finder found a defect to its

root cause. However, in Code Prover, you have more complete information, because the information helps you understand all execution paths in your code.

```

167 static void Square_Root_conv(double alpha, float* beta_pt)
168 /* Perform arithmetic conversion of alpha to beta */
169 {
170     *beta_pt = (float)((1.5 + cos(alpha)) / 5.0);
171 }
172
173
174 static
175 {
176     d
177     f
178     f
179
180     Square_Root_conv(alpha, &beta);
181
182     gamma = (float)sqrt(beta - 0.75); /* always sqrt(negative number) */
183 }

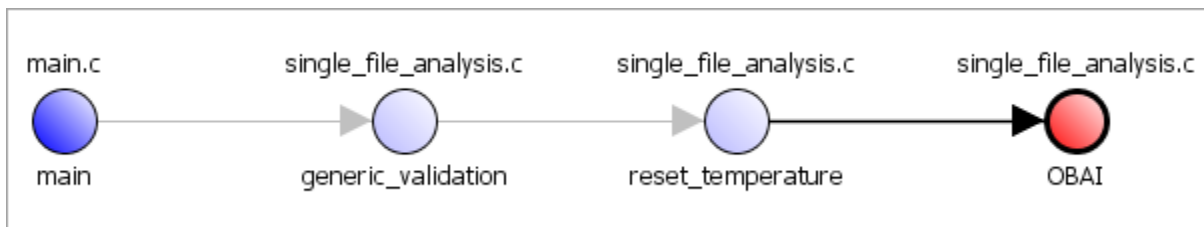
```

Dereference of parameter 'beta\_pt' (pointer to float 32, size: 32 bits):  
 Pointer is not null.  
 Points to 4 bytes at offset 0 in buffer of 4 bytes, so is within bounds (if memory is allocated).  
 Pointer may point to variable or field of variable:  
 'beta', local to function 'Square\_Root'.

Assignment to dereference of parameter 'beta\_pt' (float 32): [0.1 .. 0.5]

Press 'F2' for focus

### Data Flow Analysis in Code Prover



### Control Flow Analysis in Code Prover

#### Few False Positives with Bug Finder

Bug Finder aims for few false positives, that is, results that you are not likely to fix. By default, you are shown only the defects that are likely to be most meaningful for you.

Bug Finder also assigns an attribute called impact to the defect types based on the criticality of the defect and the rate of false positives. You can choose to analyze your code only for high-impact defects. You can also enable or disable a defect that you do not want to review<sup>2</sup>.

#### Zero False Negatives with Code Prover

Code Prover aims for an exhaustive analysis. The software checks every operation that can trigger specific types of error. If a code operation is green, it means that the operation cannot cause those run-time errors that the software checked for<sup>3</sup>. In this way, the software aims for zero false negatives.

<sup>2</sup> You can also disable certain Code Prover defects related to non-initialization.

If the software cannot prove the absence of an error, it highlights the suspect operation in red or orange and requires you to review the operation.

### **Coding Rule Support in Bug Finder**

Bug Finder supports checking for compliance with external coding standards, such as:

- AUTOSAR C++14. See “AUTOSAR C++14 Rules”.
- MISRA C:2012. See “MISRA C:2012 Directives and Rules”.
- MISRA C++:2008. See “MISRA C++:2008 Rules”.
- CERT C. See “CERT C Rules and Recommendations”.
- CERT C++. See “CERT C++ Rules”.

For the complete list of coding standards support, see “Polyspace Support for Coding Standards”.

### **See Also**

#### **More About**

- “Workflow Using Both Polyspace Bug Finder and Polyspace Code Prover” (Polyspace Code Prover)

---

3 The Code Prover result holds only if you execute your code under the same conditions that you supplied to Code Prover through the analysis options.

## Workflow Using Both Polyspace Bug Finder and Polyspace Code Prover

If you have both Bug Finder and Code Prover, based on the above differences, you can deploy the two products appropriately in your software development workflow. For instance:

- All developers in your organization can run Bug Finder on newly developed code. For maintaining standards across your organization, you can deploy a common configuration that looks only for specific defect types.

Code Prover can be deployed as part of your unit testing suite.

- You can run Code Prover only on critical components of your project, while running Bug Finder on the entire project.
- You can run Code Prover on modules of code at the unit testing level, and run Bug Finder when integrating the modules.

You can run Code Prover before unit testing. Code Prover exhaustively checks your code and tries to prove the presence or absence of errors. Instead of writing unit tests for your entire code, you can then write tests only for unproven code. Using Code Prover before unit testing reduces your testing efforts drastically.

Depending on the nature of your software development workflow and available resources, there are many other ways you can incorporate the two kinds of analysis. You can run both products on your desktop during development or as part of automated testing on a remote server. Note that it is easier to interpret and fix bugs closer to development. You benefit from using both products if you deploy them early and often in your development process.

There are two important considerations if you are running both Bug Finder and Code Prover on the same code.

- Starting in R2022a, Polyspace Bug Finder is the recommended tool for checking compliance to external coding standards such as AUTOSAR C++14 or MISRA C++:2008. Check for violation of these coding standards when you use Bug Finder on your code. You might have used Polyspace Code Prover to check for this purpose. Migrate your workflow to use Bug Finder. See “Migrate Code Prover Workflows for Checking Coding Standards and Code Metrics to Bug Finder”.
- You can use the same project for both Bug Finder and Code Prover analysis. The following set of options are common between Bug Finder and Code Prover:
  - “Target and Compiler”
  - “Macros”
  - “Environment Settings”
  - “Inputs and Stubbing”
  - “Multitasking”
  - “Coding Standards & Code Metrics”
  - “Reporting”, except Bug Finder and Code Prover report (-report-template)

You might have to change more of the default options when you run the Code Prover verification because Code Prover is stricter about compilation and linking errors.

