Intel® Parallel Inspector 2011 Getting Started Tutorials

Document Number: 323355-001US

World Wide Web: http://developer.intel.com

Legal Information

Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to:http://www.intel.com/products/processor%5Fnumber/

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Core Inside, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, the Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, InTru, the InTru logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Moblin, Pentium, Pentium Inside, skoool, the skoool logo, Sound Mark, The Journey Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, Visual Studio, Visual C++, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Microsoft product screen shot(s) reprinted with permission from Microsoft Corporation.

Contents

Chapter 3: More Resources

Introducing the Intel® Parallel Inspector

The Intel® Parallel Inspector is a serial and multithreading error checking analysis tool for Microsoft Visual Studio* C/C++ developers. The Inspector detects challenging memory leaks and corruption errors as well as threading data races and deadlock errors. This comprehensive developer productivity tool pinpoints errors and provides guidance to help ensure application reliability and quality.

The Inspector currently supports applications:

- Created with C/C++ code
- Developed in the Visual Studio* 2005/2008/2010 IDE
- Using the Intel® Threading Building Blocks, Intel® Cilk™ Plus, Windows* API, or OpenMP* API programming models
- Developed to run on the Windows* operating systems listed in *Release Notes*.

Inspector Tutorials

These tutorials tell you how to use the Inspector to find and fix memory errors in serial or parallel applications, and find and fix threading errors in parallel applications.

- Managing Memory Errors
- Managing Threading Errors

Check http://software.intel.com/en-us/articles/intel-software-product-tutorials/ for the following:

- Printable version (PDF) of all Inspector tutorials
- Show Me Video of each Inspector tutorial

See Also

- Getting Help
- More Resources

Prerequisites

You need the following tools, skills, and knowledge to effectively use these tutorials.

NOTE. The steps and screen captures in these tutorials are based on the Visual Studio* 2005 IDE. They may differ slightly for other versions of the Visual Studio* IDE.

Required Tools

You need the following tools to use these tutorials:

- Inspector
- Sample code shipped with the Inspector
- Inspector Help
- Supported compiler (see *Release Notes* for more information)

To acquire the Inspector:

If you do not already have access to the Inspector, you can download an evaluation copy from http://software.intel.com/en-us/intel-parallel-studio-home/

To install and set up Inspector sample code:

1. Copy the `tachyon_studio.zip` file from the `Samples\<locale>` folder in the Parallel Studio installation folder (the default installation folder is `C:\Program Files\Intel\Parallel Studio 2011`) to a writable directory or share on your system.
2. Extract the sample(s) from the `.zip` file.

NOTE.

- Samples are non-deterministic. Your screens may vary from the screen captures shown throughout these tutorials.
- Samples are designed only to illustrate Inspector features and do not represent best practices for creating multithreaded code.

To access Inspector Help:

See Getting Help

Required Skills and Knowledge

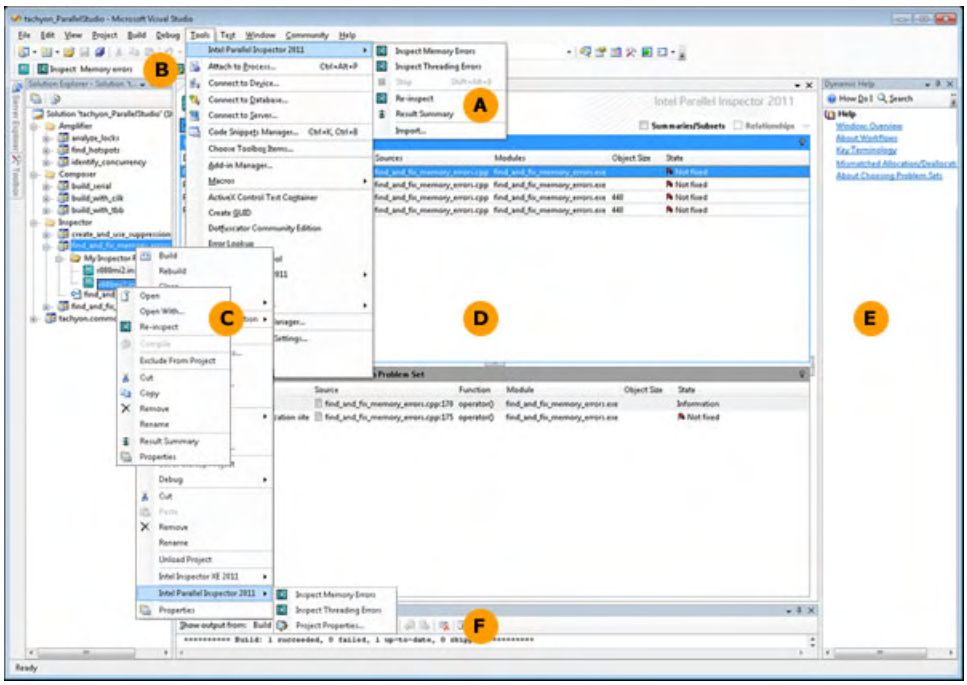
These tutorials are designed for developers with a basic understanding of the Visual Studio* 2005 development environment (IDE), including how to:

- Open a project/solution.
- Display the **Solution Explorer**.
- Compile and link an application.
- Ensure an application compiled successfully.
- Access the **Document Explorer** window.

Navigation Quick Start

Inspector/Visual Studio* 2005 Integration

The Inspector integrates into the Visual Studio* development environment (IDE) and can be accessed from the menus, toolbar, and **Solution Explorer** in the following manner:



A — Use the **Tools > Intel Parallel Inspector 2011** menu to control result collection.

B — Use the **Inspector** toolbar to control result collection.

C — **Solution Explorer** pop-up menus and folders:

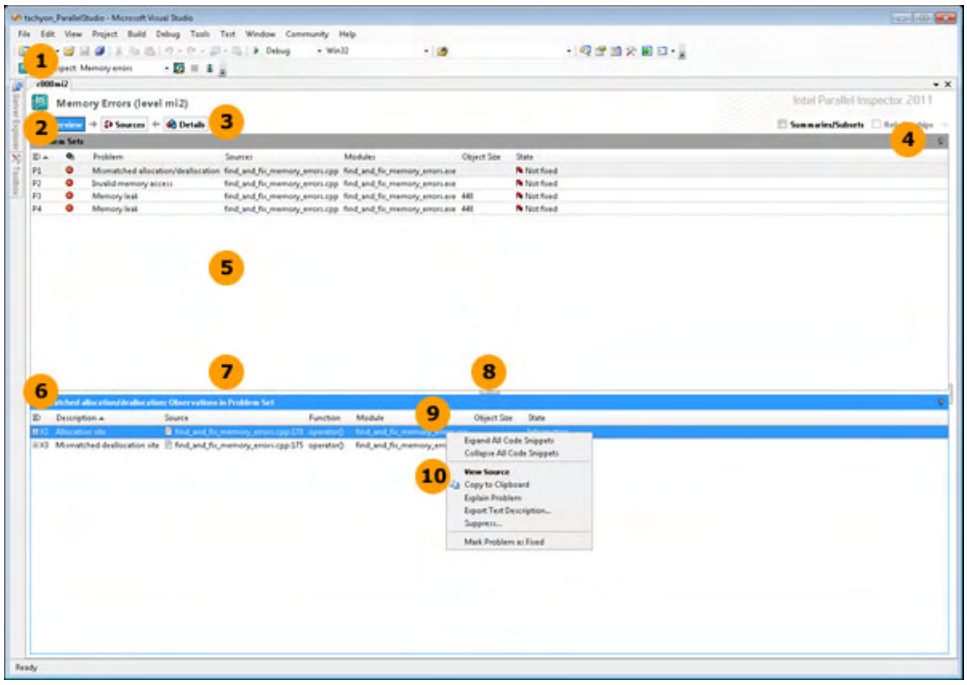
- Use the **Intel Parallel Inspector 2011** menu on the **Solution Explorer** project pop-up menu to control result collection.
- Use the pop-up menu on results in the **My Inspector Results** folder to manage previously collected results.

D Use the Inspector result tabs to manage result data.

E Use the **Dynamic Help** window to access help topics related to the current Inspector window.

F Use the **Output** window to view application execution and analysis output.

Inspector Result Tabs



1 Use result tab names to distinguish among results.

2 Click the icon to display a help snippet and provide links to more help information. (You can also press the F1 key to display more complete help topics and links.)

3 Click buttons on the navigation toolbar to change window views.

4 Click buttons to display help pages that describe how to use window panes.

5 Use window panes to manage result data.

6 Use title bars to identify window panes.

7 Drag window pane borders to resize window panes.

8 Click , , , and icons to toggle window panes on and off.

9 Data column headers - Drag to reposition the data column; drag the left or right border to resize the data column; click to sort results in ascending or descending order by column data.

10 Right-click data in window panes to display pop-up menus that provide access to key capabilities.

Tutorial: Managing Memory Errors 1

Learning Objectives

This tutorial shows how to use the Intel® Parallel Inspector to identify, analyze, and resolve memory errors.

Estimated completion time: 10 - 15 minutes.

NOTE. This tutorial does not require prior completion of other tutorials. Consequently, some terms, concepts, and workflow steps are common to all tutorials.

After you complete this tutorial, you should be able to:

- List, in order, the steps to manage memory errors.
- Define key Inspector terms, such as *analysis*, *result*, *problem set*, *problem*, and *observation.*
- Identify compiler/linker options that produce the most accurate, complete analysis results.
- Explain how data set size impacts application execution time and analysis speed.
- Run memory error analyses.
- Influence analysis scope and running time.
- Access help for the Inspector command-line interface.
- Navigate among windows in Inspector results.
- Display a prioritized *to-do* list for resolving errors.
- Access help for resolving specific errors.
- Access source code to fix errors.

Key Terms and Concepts

Key Terms

analysis: A process during which the Inspector performs collection and finalization.

collection: A process during which the Inspector executes an application, identifies issues that may need handling, and collects those issues in a result.

false positive: The Inspector detects something that is not an error.

false negative: The Inspector does not detect an error because the problem may be too complex/big or involve too much runtime/memory cost.

finalization: A process during which the Inspector uses debug information from binary files to convert symbol information into filenames and line numbers, perform duplicate elimination, and form problem sets.

observation: A fact the Inspector observes at a source code location, such as a *write* observation. A focus observation is a source code location with relationships you choose to explore. A related observation is a source code location with a relationship to a focus observation and possibly other observations.

problem: A small group of closely related observations (source code locations) that indicate an error in an application, such as a *data race* problem.

problem set: A larger group of more loosely related observations (source code locations) that could share a common solution, such as a problem set resulting from deallocating an object too early during program execution. You can view problem sets only after analysis is complete.

project: A compiled application, collection of configurable attributes surrounding the compiled application, and a container for results.

result: A collection of issues that may need handling.

target: An application you inspect for errors using the Inspector.

Key Concept: Choosing Small, Representative Data Sets

When you run a dynamic analysis, the Inspector executes an application against a data set. Data set size has a direct impact on application execution time and analysis speed.

For example, it takes longer to process a 1000x1000 pixel image than a 100x100 pixel image. A possible reason: You may have loops with an iteration space of 1...1000 for the larger image, but only 1...100 for the smaller image. The exact same code paths may be executed in both cases. The difference is the number of times these code paths are repeated.

You can control analysis *cost* without sacrificing completeness by removing this kind of redundancy from your data set.

Instead of choosing large, repetitive data sets, choose small, representative data sets. Data sets with runs in the seconds time range are ideal. You can always create additional data sets to ensure all your code is inspected.

Key Concept: Choosing Analysis Type Configurations

The Inspector offers preset memory and threading configurations to help you control analysis scope and *cost*.

Preset Configurations	Advantages	Disadvantages
Narrowest scope	• Minimizes the load on the system. • Minimizes the time required to perform the analysis. • Increases the chances the analysis will complete successfully - particularly on large applications/large data sets.	• Detects only simple errors. • Provides no details.

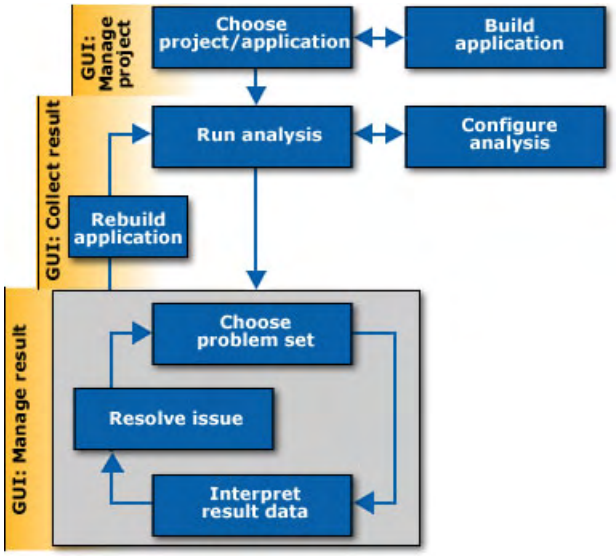
Preset Configurations	Advantages	Disadvantages
Intermediate scope(s)	• Detects important errors. • Provides some context and detail for detected errors.	• Increases the load on the system. • Increases the time required to perform the analysis. • Increases the chances the analysis will fail because the system may run out of resources.
Widest scope	• Detects important errors. • Provides context and maximum amount of detail for detected errors.	• Maximizes the load on the system. • Maximizes the time required to perform the analysis. • Maximizes the chances the analysis will fail because the system may run out of resources.

For more details, check the Inspector Help Index for *configuration*.

Workflow Steps to Identify, Analyze, and Resolve Memory Errors

You can use the Inspector to identify, analyze, and resolve memory errors in serial or parallel programs by performing a series of steps in a workflow. This tutorial guides you through these workflow steps using `tachyon_studio` sample code.

NOTE. Click Show Me for a video demonstration. Show Me Videos require Adobe* Flash* Player.



1. Choose a project.
2. Verify settings and build an application to inspect for errors.
3. Configure a memory error analysis.
4. Run the memory error analysis on the application.
5. Choose a problem set in the analysis result.
6. Interpret the result data.
7. Resolve the issue.
8. Resolve the next issue.
9. Rebuild the application and rerun the memory error analysis.

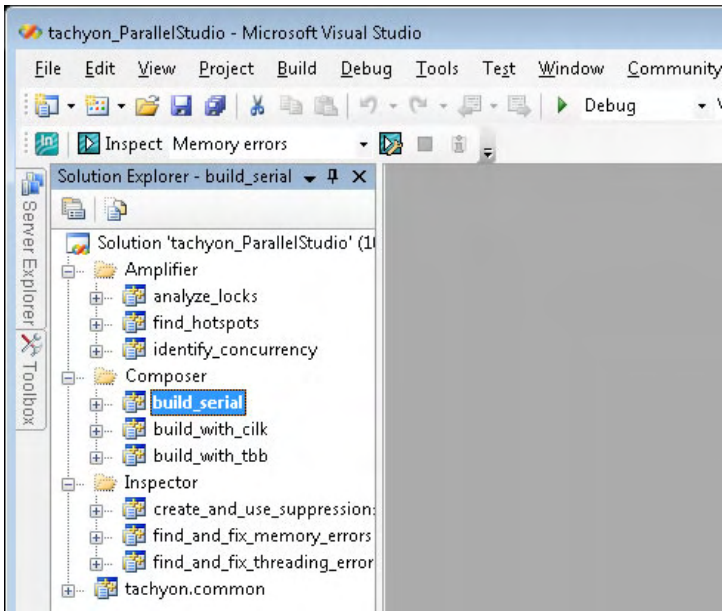
Choose Project

To enable Inspector functionality:

- Open a Visual Studio* solution.
- Set a startup project.

To open a Visual Studio* solution:

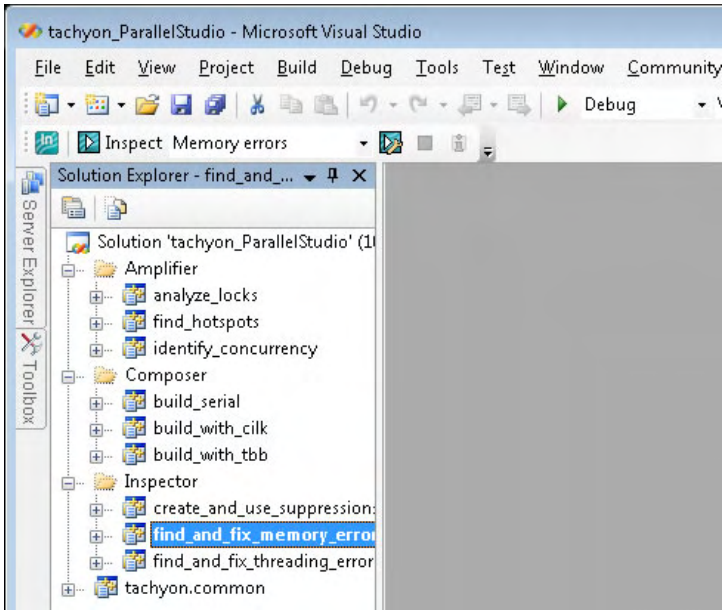
1. From the Visual Studio* menu, choose **File** > **Open** > **Project/Solution**.
2. In the **Open Project** dialog box, open the `tachyon_studio\vc8\tachyon_ParallelStudio.sln` file to display the **tachyon_ParallelStudio** solution in the **Solution Explorer**:



NOTE. See Prerequisites to locate and extract the `*.sln` file.

To choose a startup project:

1. Right-click the **find_and_fix_memory_errors** project.
2. Choose **Set as Startup Project**:



Recap

You chose the **find_and_fix_memory_errors** project.

Key Terms and Concepts

- Concept: Choosing Small, Representative Data Sets

Build Application

You can use the Inspector on both debug and release modes of binaries containing native code; however, projects compiled/linked in debug mode using the following options produce the most accurate, complete results.

Compiler/Linker Options	Correct Setting	Impact If Not Set Correctly
Debug information	Enabled (`/Zi` or `/ZI`)	Missing file/line information
Optimization	Disabled (`/Od`)	Incorrect file/line information
Dynamic runtime library	Selected (`/MD` or `/MDd`)	False positives or missing observations

To enable correct analysis of Intel® Threading Building Blocks (Intel® TBB) applications, you must also set the following required macros before compiling:

- **TBB_USE_DEBUG** (which sets **TBB_USE_THREADING_TOOLS**) if you use Intel® TBB debug libraries
- **TBB_USE_THREADING_TOOLS** if you use Intel® TBB release libraries

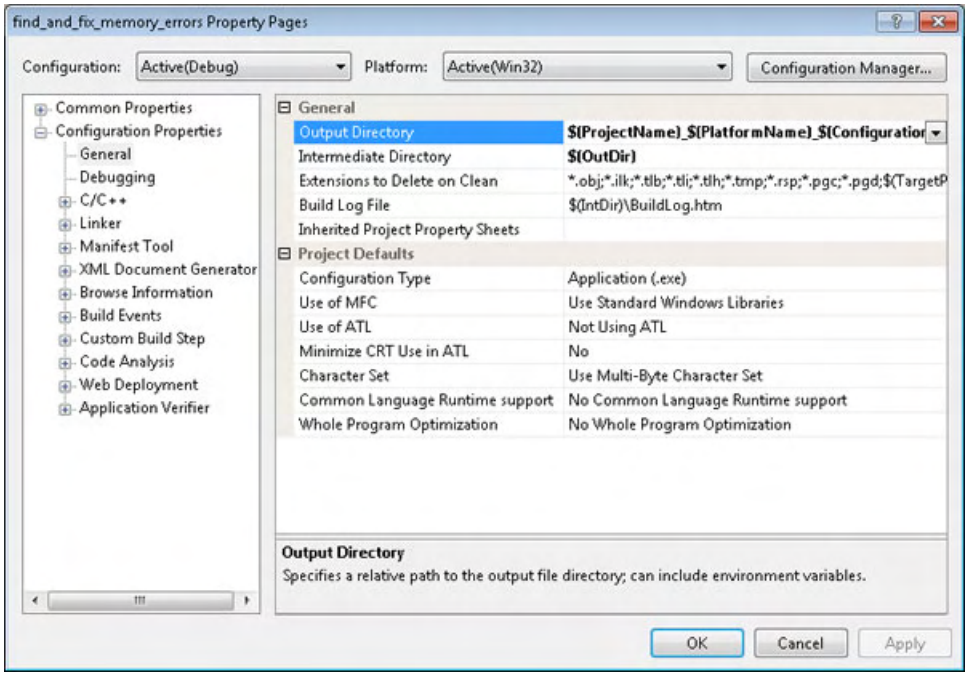
The impact if not set correctly: Intel Inspector XE may generate false positives. See Intel® TBB documentation for more information.

To create an application the Inspector can inspect for memory errors:

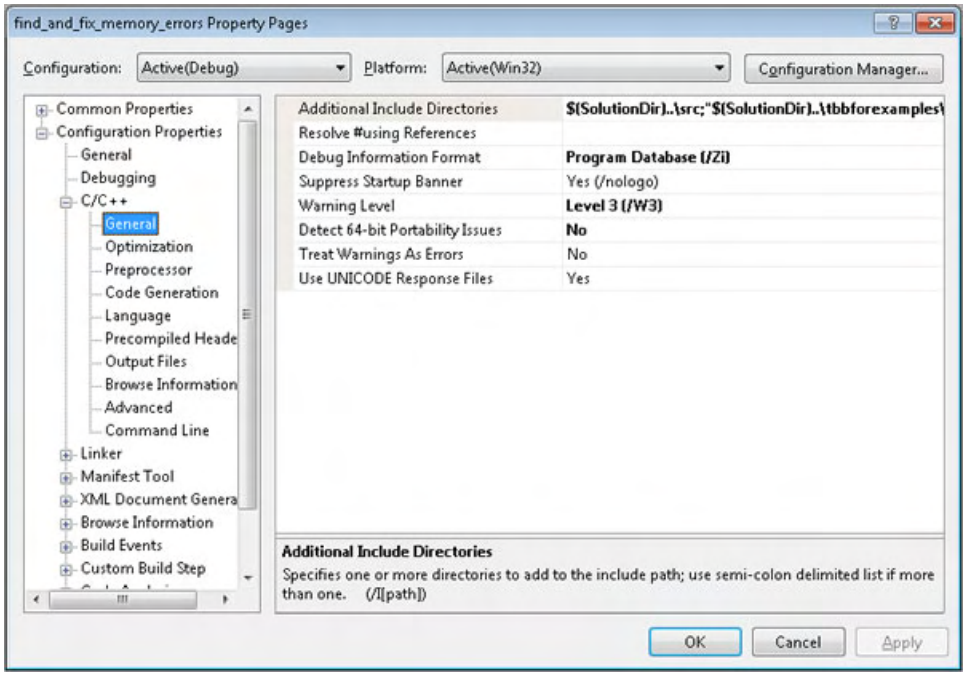
- Verify debug mode is configured to produce the best results.
- Verify the application is set to build in debug mode.
- Build the application.

To verify debug mode is configured to produce the best results:

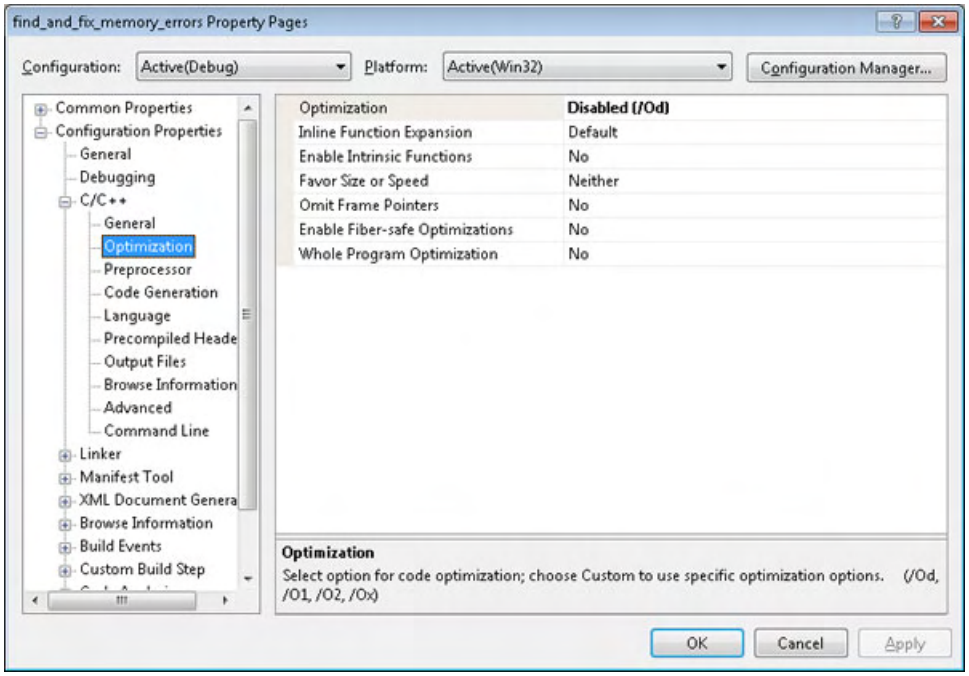
1. Right-click the **find_and_fix_memory_errors** project in the **Solution Explorer**.
2. Choose **Properties** to display the **Property Pages** dialog box.
3. Verify the **Configuration** drop-down list is set to **Debug** or **Active(Debug)**.



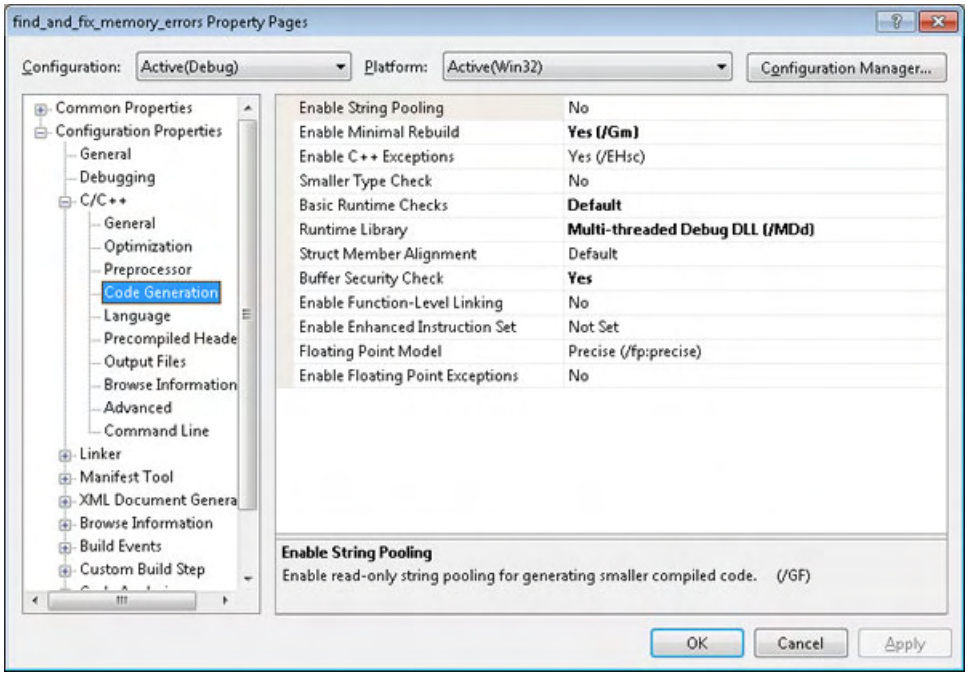
4. In the left pane, choose **Configuration Properties** > **C/C++** > **General**.
5. Verify the **Debug Information Format** is set to **Program Database (/Zi)** or **Program Database for Edit & Continue (/ZI)**.



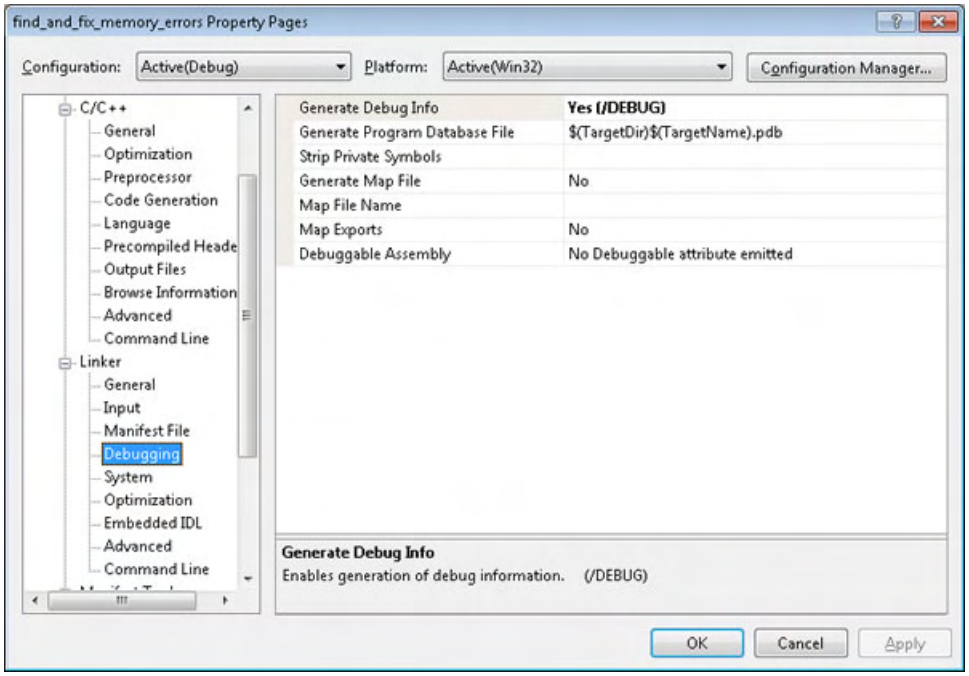
6. In the left pane, choose **Configuration Properties** > **C/C++** > **Optimization**.
7. Verify the **Optimization** field is set to **Disabled (/Od)**.



8. In the left pane, choose **Configuration Properties** > **C/C++** > **Code Generation**.
9. Verify the **Runtime Library** field is set to **Multi-threaded DLL (/MD)** or **Multi-threaded Debug DLL (/MDd)**.

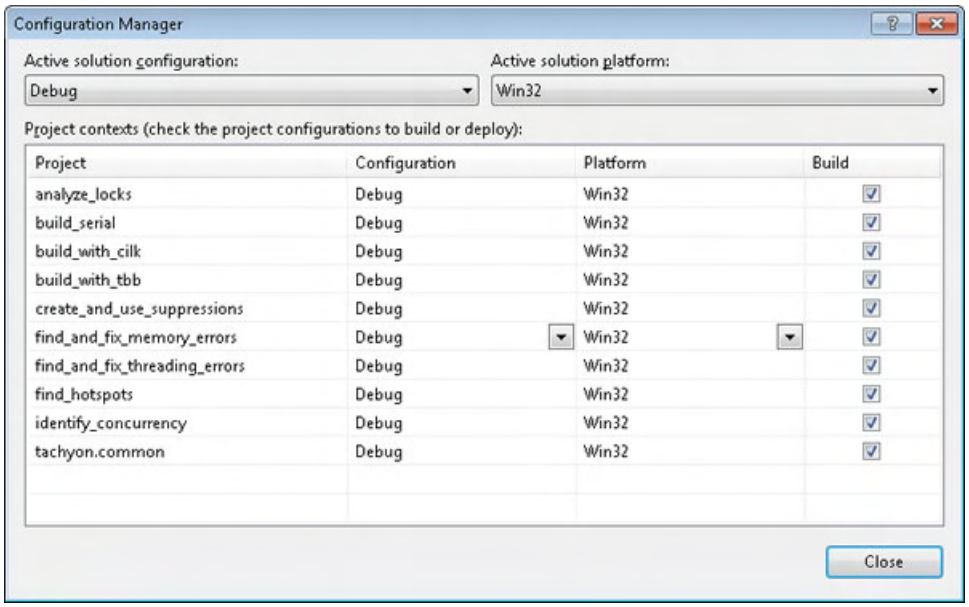


10. In the left pane, choose **Configuration Properties** > **Linker** > **Debugging**.
11. Verify the **Generate Debug Info** field is set to **Yes (/DEBUG)**.



To verify the application is set to build in debug mode:

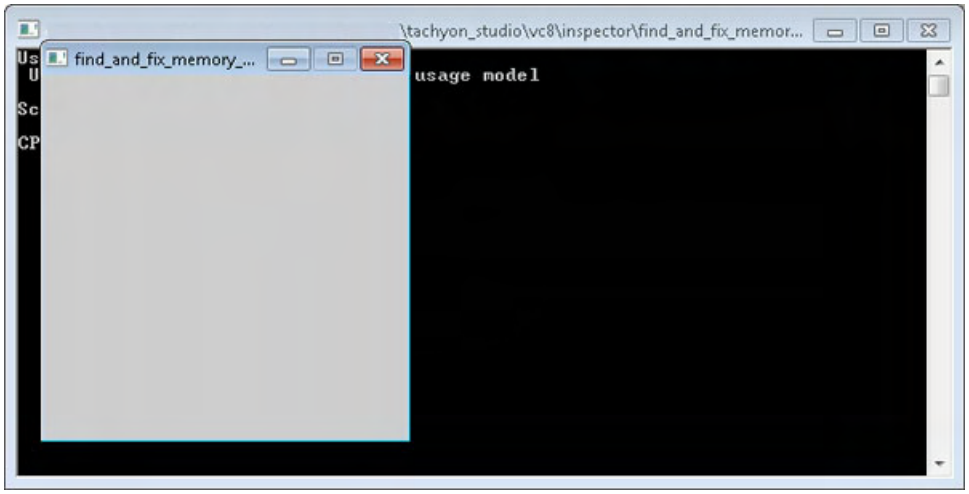
1. Click the **Configuration Manager** button.
2. Verify the **Active solution configuration** drop-down list is set to **Debug**.



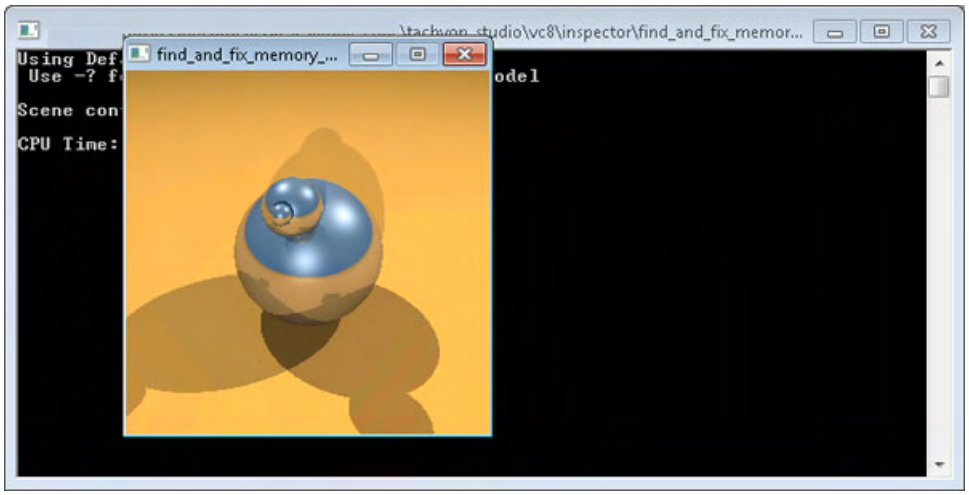
3. Click the **Close** button to close the **Configuration Manager** dialog box.
4. Click the **OK** button to close the **Property Pages** dialog box.

To build the application:

1. From the Visual Studio* menu, choose **Debug** > **Start Without Debugging**
2. If the Visual Studio* IDE responds any projects are out of date, click **Yes** to build.
3. Check for a display similar to the following:



Notice the application output window is empty. The cause: Memory errors. This is how the application output should look after various memory errors are resolved:



Recap

You verified the **find_and_fix_memory_errors** project is set to produce the most accurate, complete results; compiled and linked the application; and ensured the resulting `find_and_fix_memory_errors.exe` file runs on your system outside the Inspector.

Key Terms and Concepts

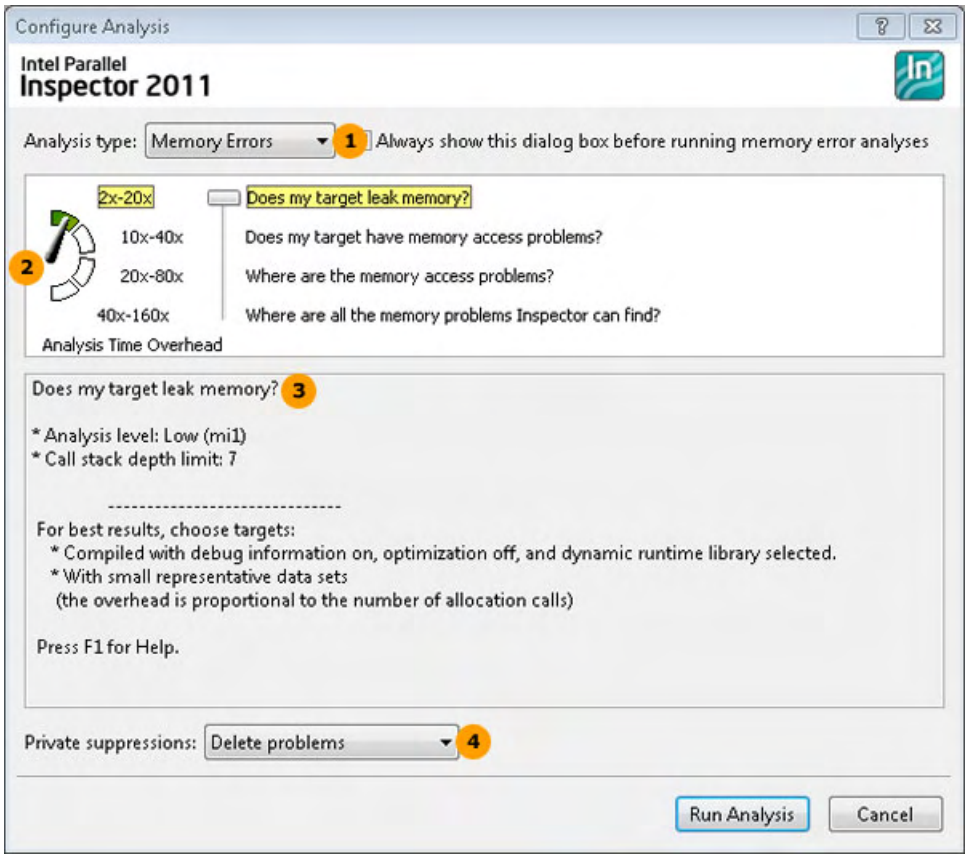
- Term: False negative
- Term: False positive

Configure Analysis

Choose a preset configuration to influence memory error analysis scope and running time.

To configure a memory error analysis:

1. From the Visual Studio* menu, choose **Tools** > **Intel Parallel Inspector 2011** > **Inspect Memory Errors** to display the **Configure Analysis** dialog box:



1 The **Analysis type** drop-down list shows Inspector dynamic analysis type offerings: Memory error analysis and threading error analysis.

This tutorial covers memory error analyses, which you can use to search for these kinds of errors: GDI resource leak, kernel resource leak, invalid memory access, memory leak, mismatched allocation/deallocation, missing allocation, and uninitialized memory access.

Use threading error analyses to search for these kinds of errors: Data race, deadlock, lock hierarchy violation, and cross-thread stack access.

2 The **Analysis Time Overhead** gauge shows the time it may take to collect a result at each preset configuration. Time is expressed in relation to normal application execution time. For example, 2x - 20x is 2 to 20 times longer than normal application execution time. If normal application execution time is 5 seconds, estimated collection time is 10 to 100 seconds.

Here, the configuration slider rests at the **Does my target leak memory?** preset configuration, which has the narrowest scope/lowest overhead. Try dragging the configuration slider to see the impact on the gauge.

3 The configuration details region shows current configuration characteristics. Try dragging the configuration slider to see the impact on the details region.

4 *Suppressing* known issues can dramatically improve your productivity. For more details, check the Inspector Help Index for *suppression rule*.

2. Drag the configuration slider to the **Does my target have memory access problems?** preset configuration:

Recap

You chose a preset configuration of intermediate scope to detect memory errors in the `find_and_fix_memory_errors.exe` file.

Key Terms and Concepts

- Term: Analysis
- Term: Target
- Concept: Choosing Analysis Type Configurations

Run Analysis

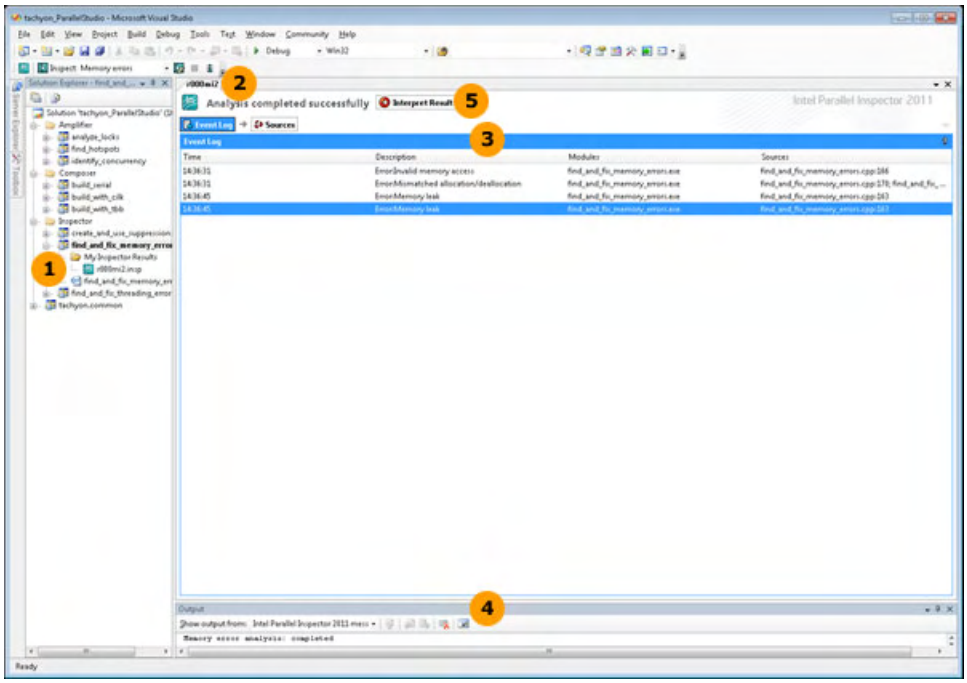
Run a memory error analysis to detect memory errors that may need handling.

To run a memory error analysis:

Click the **Run Analysis** button on the **Configure Analysis** dialog box to:

- Execute the `find_and_fix_memory_errors.exe` file.
- Identify memory errors that may need handling.
- Collect the result in a folder in the `tachyon_studio\vc8\Inspector\My Inspector Results` folder.
- Finalize the result (convert symbol information into filenames and line numbers, perform duplicate elimination, and form problem sets).

During collection, the Inspector displays a **Collection Log** window similar to the following:



1 The Visual Studio* IDE offers a pointer to the result from the **Solution Explorer** to provide easy future access. Here, the Visual Studio* IDE created a **find_and_fix_memory_errors\My Inspector Results** folder in the **Solution Explorer** with a pointer to the **r000mi2** result.

2 The result name appears in the tab. Here, the name of the result (and the name of the result folder in the `tachyon_studio\vc8\Inspector\My Inspector Results` folder) is `r000mi2`, where:

- `r` = constant
- `000` = next available number
- `mi` = memory error analysis type
- `2` = preset configuration of intermediate scope

3 The **Collection Log** pane shows detected problems in real time.

This tutorial does not cover examining and solving problems during collection. For more details, check the Inspector Help Index for *examine result during analysis*.

4 The Visual Studio* **Output** window shows Inspector analysis status messages.

If necessary, choose **Intel Parallel Inspector messages** in the **Show output from** drop-down list to see the status messages.

5 The **Interpret Result** button displays after analysis (both collection and finalization) completes successfully.

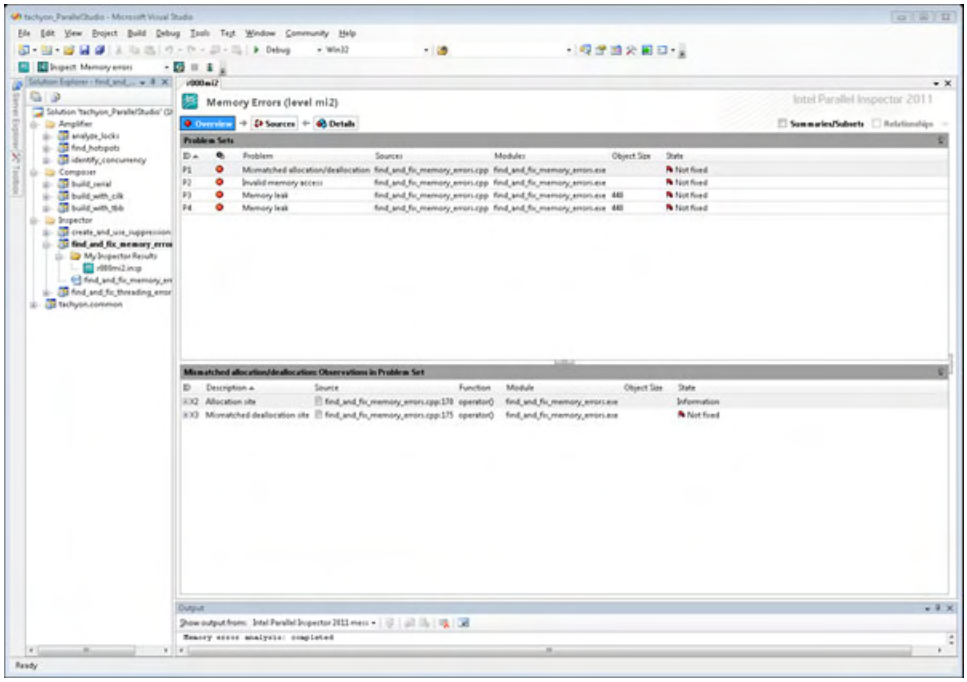
You can examine and solve problems during collection - and continue to examine and solve problems after collection and finalization are complete - but there are trade-offs involved.

TIP. Wait until analysis is complete, then click the **Interpret Result** button to use the full range of Inspector capabilities to manage result data in a more efficient, effective, and productive manner.

NOTE. This tutorial explains how to run an analysis from the Inspector graphical user interface (GUI). You can also use the Inspector command-line interface (`insp-cl` command) to run an analysis. For more details, check the Inspector Help Index for *insp-cl command*.

To start managing result data after analysis is complete:

Click the **Interpret Result** button to display the **Overview** window:



Recap

You inspected the `find_and_fix_memory_errors.exe` application for memory errors, during which:

- The Inspector ran the `find_and_fix_memory_errors.exe` file, identified memory errors that may need handling, collected a result in the `tachyon_studio\vc8\Inspector\My Inspector Results\r000mi2` folder, converted symbol information into filenames and line numbers, performed duplicate elimination, and formed problem sets.
- The Visual Studio* IDE added a pointer to the `r000mi2` result from the **Solution Explorer**.

Key Terms and Concepts

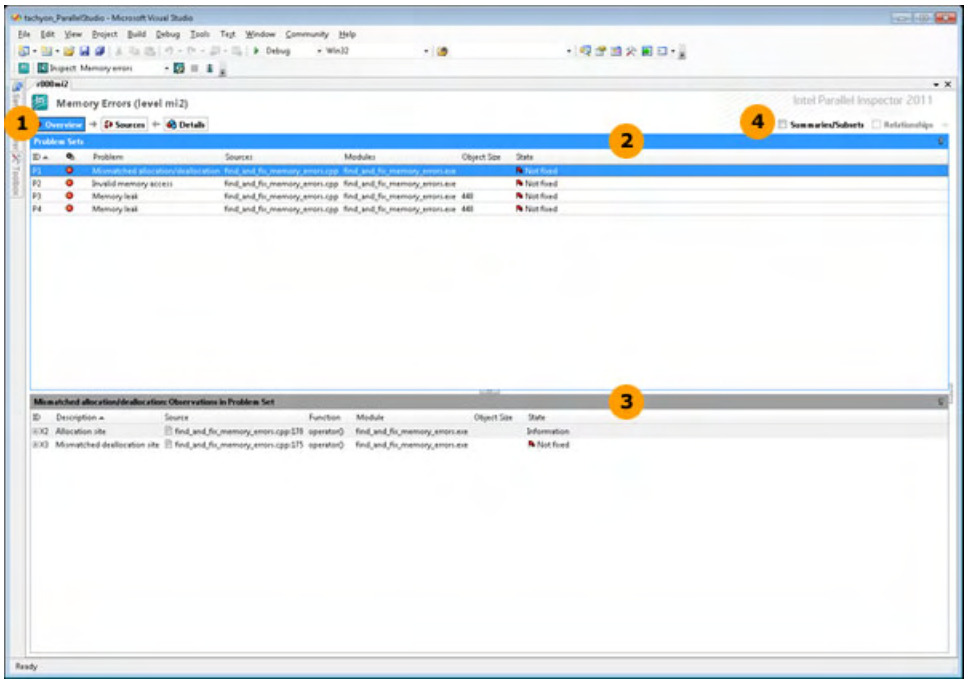
- Term: Analysis
- Term: Collection
- Term: Finalization
- Term: Problem Set
- Term: Result

Choose Problem Set

To start exploring a detected memory error:

- Interpret the window.
- Choose a problem set.

To interpret the Overview window:



1. The **Overview** window is the default starting point for managing result data. It groups observations into problem sets, and then prioritizes the problem sets by severity and size.

   The **Details** window is the advanced starting point for managing result data - it does not group observations into problem sets. This tutorial does not cover searching for observations that answer specific questions only you can formulate. For more details, check the Inspector Help Index for *select/observation search criteria*.

   The arrows in the **Navigation** toolbar indicate you can access the **Sources** window from both the **Overview** and **Details** windows.

2. Think of the **Problems** pane as a *to-do* list. Start at the top and work your way down.

3. The **Code Locations in Problem(s)** pane shows all the observations in all the problems in the selected problem set. By default, the Inspector selects the first problem set for you.

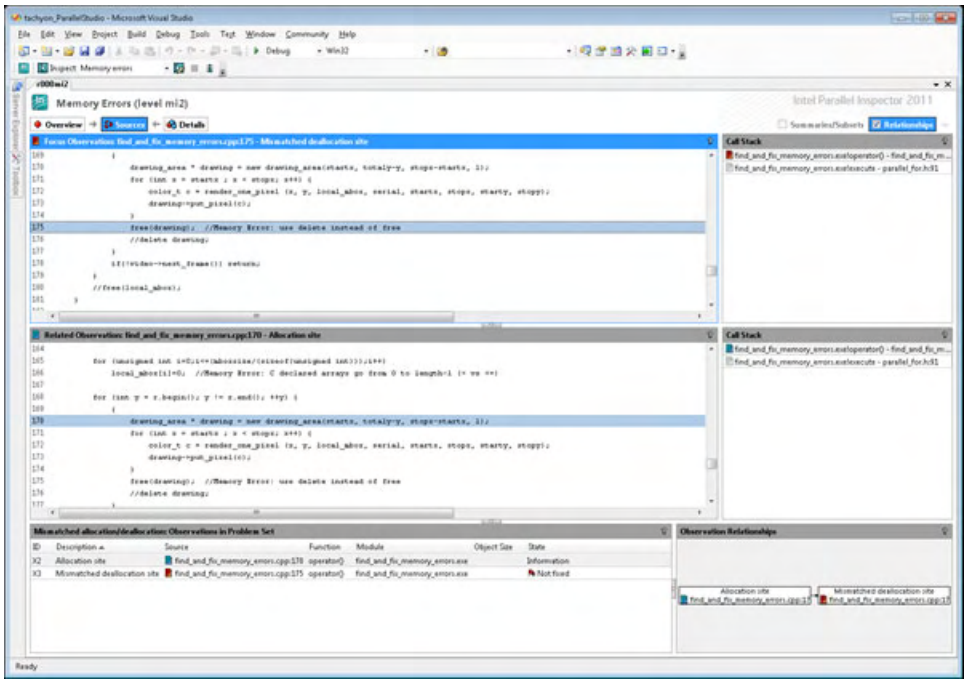
   Here, the Inspector selected the **Mismatched allocation/deallocation** problem set, which contains one problem composed of an **Allocation site** and **Mismatched deallocation site** observation in the `find_and_fix_memory_errors.cpp` source file.

4. The **Filters** pane (toggled off by default) categorizes all displayed problem sets by specific criteria. Use it to temporarily limit the list in the **Problems** pane to only those problem sets that meet specific criteria.

   This tutorial does not cover searching for specific problem sets. For more details, check the Inspector Help Index for *select/problem set search criteria*.

To choose a problem set:

Double-click the data row for the **Mismatched allocation/deallocation** problem set to display the **Sources** window, which provides more visibility into the error:



Recap

You started exploring a **Mismatched allocation/deallocation** problem set that contains one problem composed of an **Allocation site** and a **Mismatched deallocation site** observation.

Key Terms and Concepts

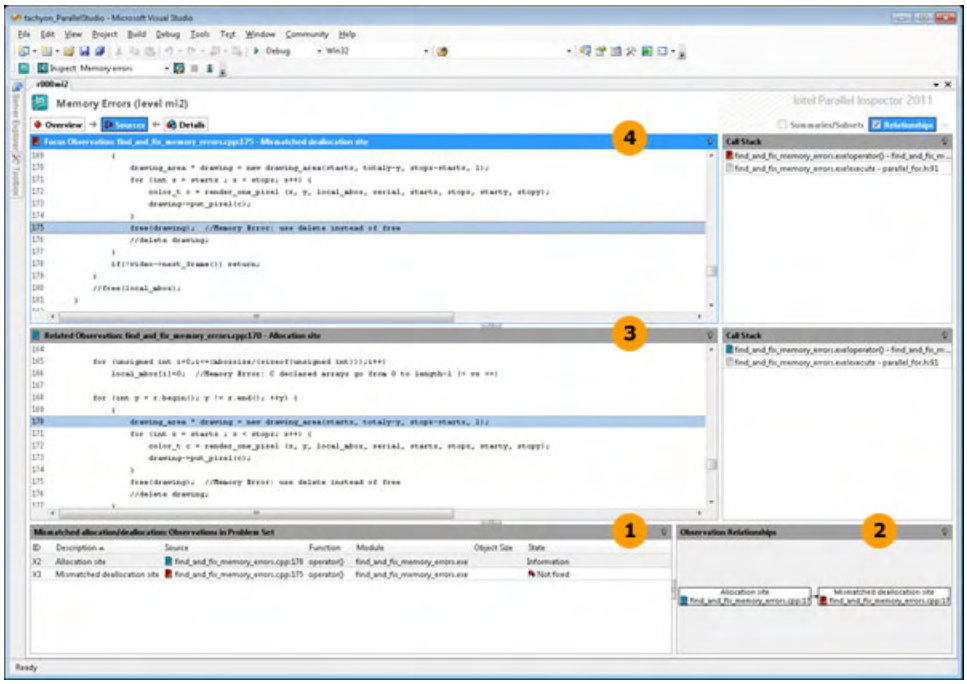
- Term: Observation
- Term: Problem
- Term: Problem set
- Term: Result

Interpret Result Data

To determine the cause of the detected memory error:

- Interpret the window.
- Interpret window icons.
- Understand the issue.
- Access more information on interpreting and resolving problems.

To interpret the Sources window:



1 Like the pane on the **Overview** window, the **Code Locations in Problem(s)** pane shows all the observations in the **Mismatched allocation/deallocation** problem in the **Mismatched allocation/deallocation** problem set.

The **Allocation site** observation represents the location and associated call stack from which the memory block was allocated. The **Mismatched deallocation site** observation represents the location and associated call stack attempting the deallocation.

2 The **Relationships** pane shows the relationship between the **Allocation site** and **Mismatched deallocation site** observations in the **Mismatched allocation/deallocation** problem. In relationship diagrams:

- Each box in a diagram represents an observation in one problem in a problem set.
- A diagram with a single box is a trivial problem with no related observation.
- Vertically stacked boxes indicate concurrent observations.
- Boxes arranged left-to-right with connecting arrows indicate a time ordering. Here, the **Allocation site** observation occurs before the **Mismatched deallocation site** observation.
- Boxes with connecting lines indicate association.

3 The **Related Code Location** pane shows the source code in the `find_and_fix_memory_errors.cpp` source file surrounding the **Allocation site** observation. (Notice the icon in the pane title matches the icon on the **Allocation site** observation data row in the **Code Locations in Problem(s)** pane.) The source code corresponding to the **Allocation site** observation is highlighted.

4 The **Focus Code Location** pane shows the source code in the `find_and_fix_memory_errors.cpp` source file surrounding the **Mismatched deallocation site** observation. (Notice the icon in the pane title matches the icon on the **Mismatched deallocation site** observation data row in the **Code Locations in Problem(s)** pane.) The source code corresponding to the **Mismatched deallocation site** observation is highlighted.

To interpret window icons:

Icon	Meaning	
	This observation is the focus observation. The Inspector chose it for you when you double-clicked the **Mismatched allocation/deallocation** problem set on the **Overview** window. Its source code is currently displayed in the **Focus Code Location** pane.	Observation source code is available for viewing in the Inspector and editing in the Visual Studio* editor.
	This observation is related to the focus observation. Its source code is currently displayed in the **Related Code Location** pane.	

To understand the issue:

Look at the code in the **Focus Code Location** pane and the **Related Code Location** pane.

The code in the **Allocation site** observation in the **Related Code Location** pane contains a new allocator, while the code in the **Mismatched deallocation site** observation in the **Focus Code Location** pane contains a `free()` deallocator.

A **Mismatched allocation/deallocation** problem occurs when you attempt a deallocation with a function that is not the logical reflection of the allocator. In the C++ programming language, the following are matched reflections:

- `new` and `delete`
- `new[]` and `delete[]`
- `malloc()` and `free()`

Only the matching deallocation technique is uniquely aware of all the memory allocation techniques and internal data storage used by the allocation technique. Using the wrong deallocation technique will almost certainly corrupt memory reclamation, its sole job.

NOTE. A **Mismatched allocation/deallocation** problem does not always cause an application crash; however, if it does cause a crash, the crash may occur later at a seemingly unrelated location.

To access more information on interpreting and resolving problems:

1. Right-click any observation in the **Code Locations in Problem(s)** pane or box in the **Relationships** pane.
2. Choose **Explain Problem** to display Inspector Help information for the **Mismatched allocation/deallocation** problem type.

Recap

You determined the cause of the **Mismatched allocation/deallocation** problem set in the `find_and_fix_memory_errors.cpp` source file: `new` and `free` are not matching techniques.

Key Terms and Concepts

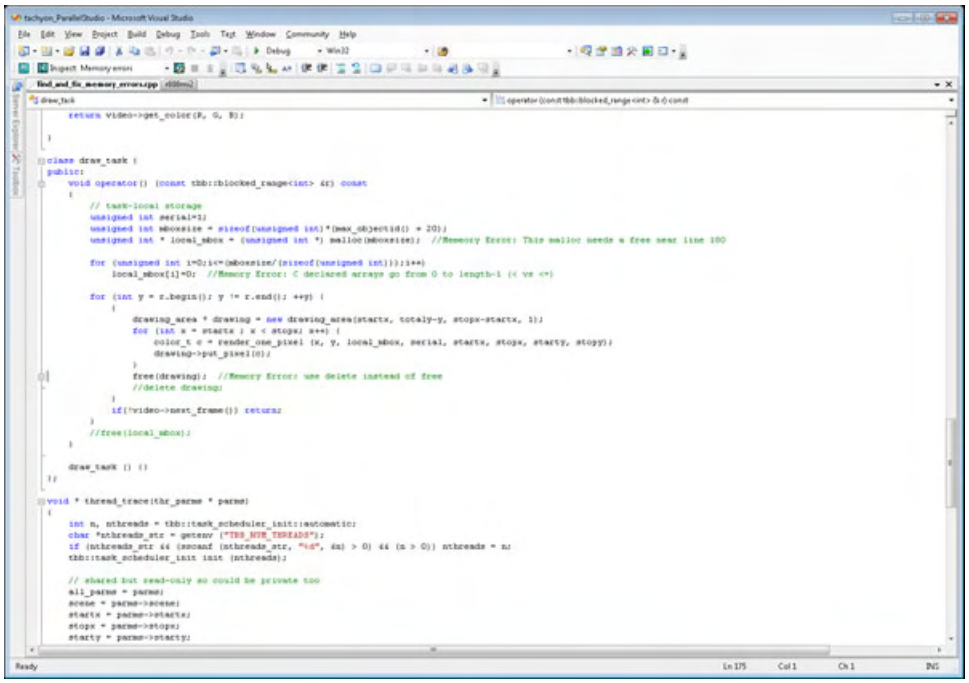
- Term: Observation
- Term: Problem
- Term: Problem set
- Term: Related observation
- Term: Result

Resolve Issue

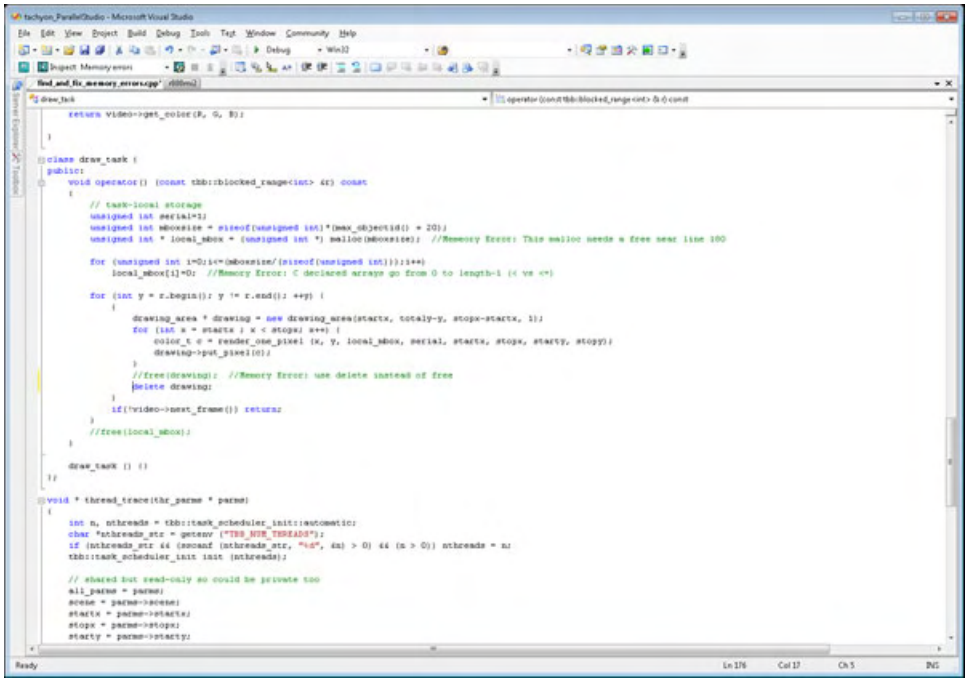
Access the Visual Studio* editor to fix the memory error.

To resolve the issue:

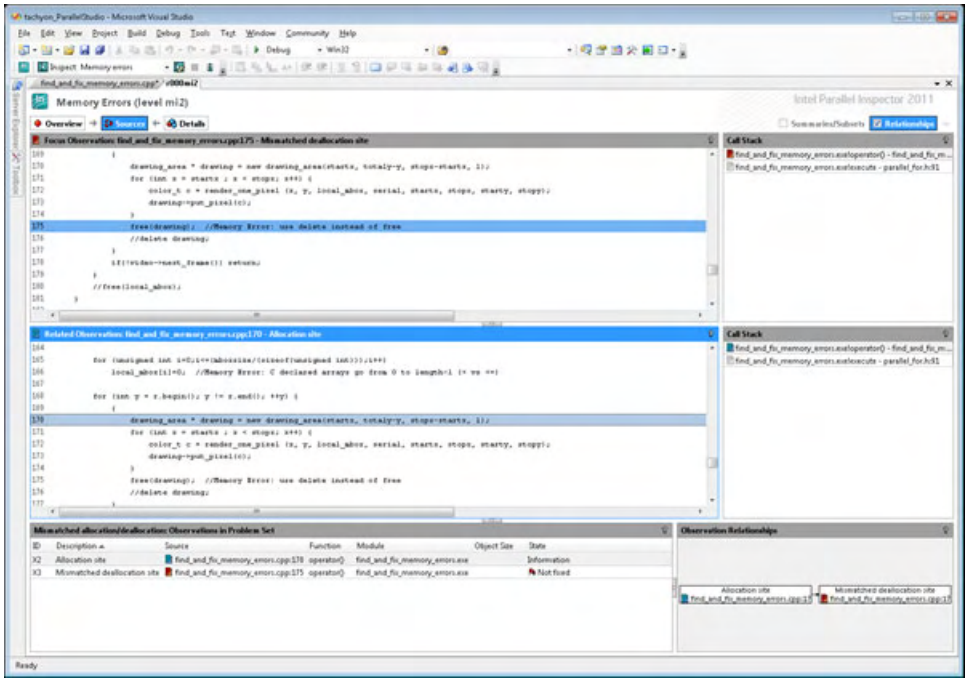
1. Double-click the highlighted code in the **Focus Code Location** pane to open the `find_and_fix_memory_errors.cpp` source file in a separate tab where you can edit it with the Visual Studio* editor:



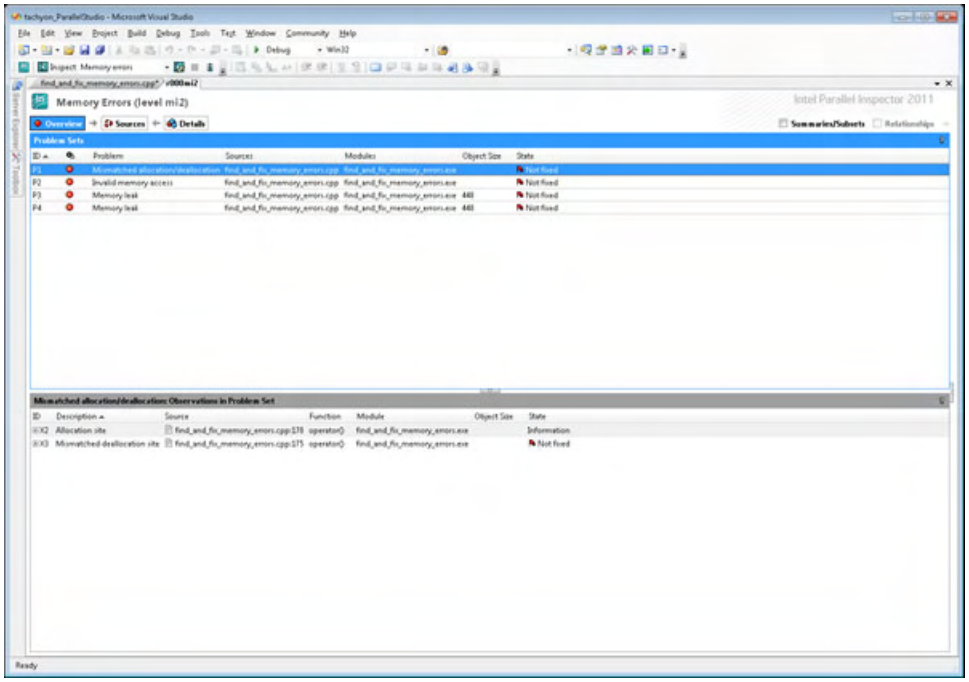
2. Comment `free(drawing);` and uncomment `//delete drawing;`:



3. Click the **r000mi2** tab to redisplay the **Sources** window:



4. Click the **Overview** button to redisplay the **Overview** window:



Recap

You edited the `find_and_fix_memory_errors.cpp` source file to fix the **Mismatched allocation/deallocation** problem.

Key Terms and Concepts

- Term: Observation
- Term: Problem
- Term: Problem set
- Term: Result

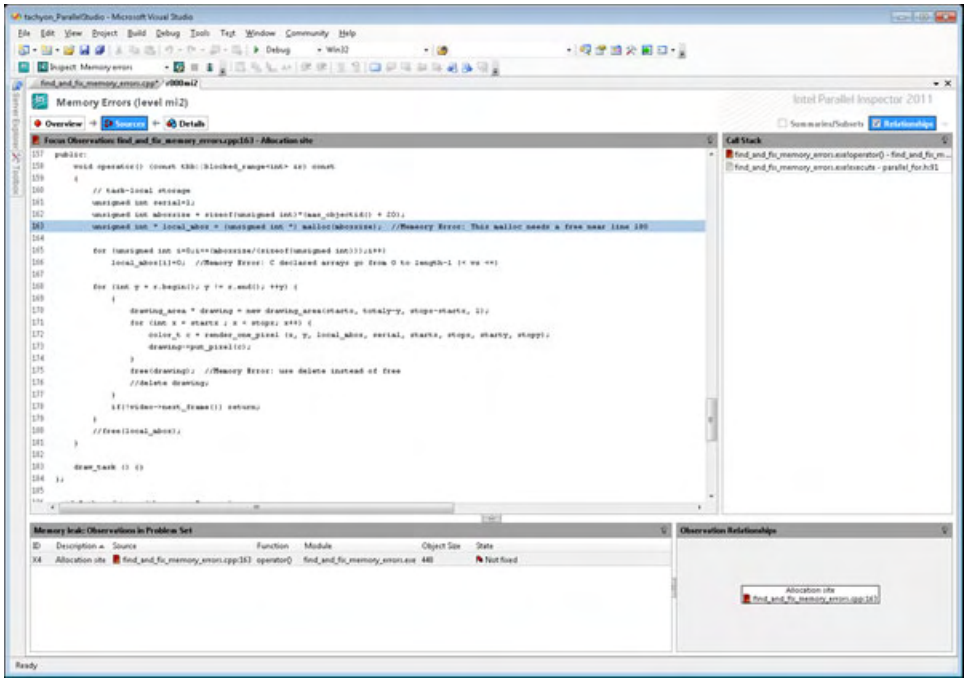
Resolve Next Issue

To resolve another detected memory error:

- Choose another problem set.
- Access more information on interpreting and resolving problems.
- Interpret the result data.
- Resolve the issue.

To choose another problem set:

In the **Problems** pane on the **Overview** window, double-click the data row for a **Memory Leak** problem set to display the **Sources** window:



To access more information on interpreting and resolving problems:

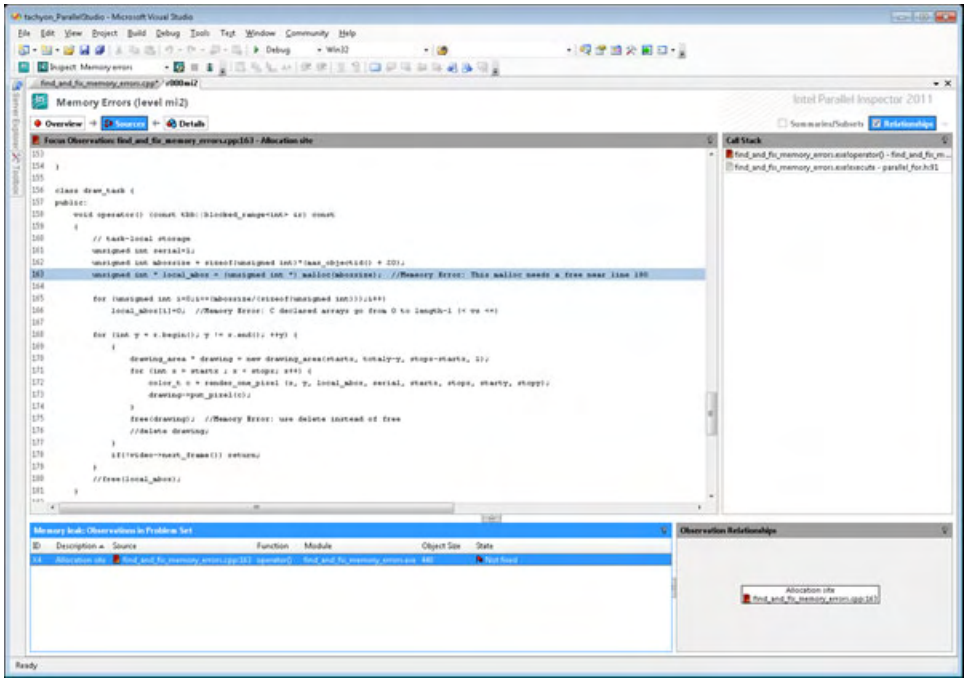
1. Right-click the **Allocation site** observation in the **Code Locations in Problem(s)** pane.
2. Choose **Explain Problem** to display Inspector Help information for the **Memory Leak** problem type.

To interpret result data:

A **Memory leak** problem occurs when a block of memory is allocated and never released.

The **Allocation site** observation represents the location from which the memory block was allocated. Here, the source code shows a `malloc()` allocator.

Scroll the source code to near line 180:



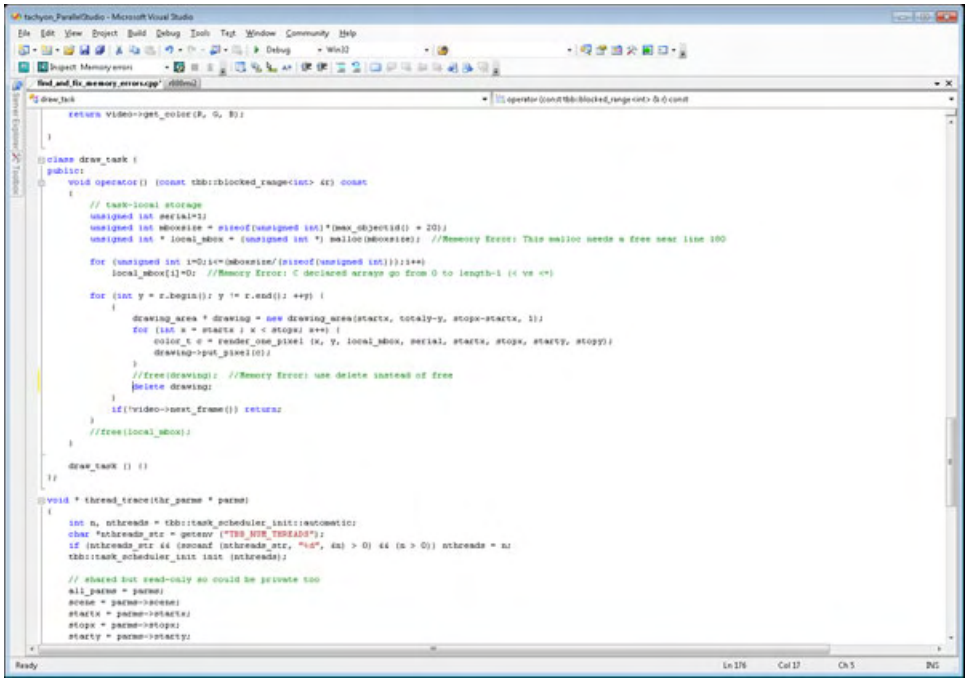
The issue is obvious: The `free()` call that releases the memory allocated by the `malloc()` call is commented out.

To resolve memory leak problems (while avoiding related mismatched allocation/deallocation problems): Match any `malloc()` call with a `free()` call, a new with a `delete`, and a `new[]` with a `delete[]` as soon as all use of the allocated space has occurred, but not sooner. Never mix and match. Be aware:

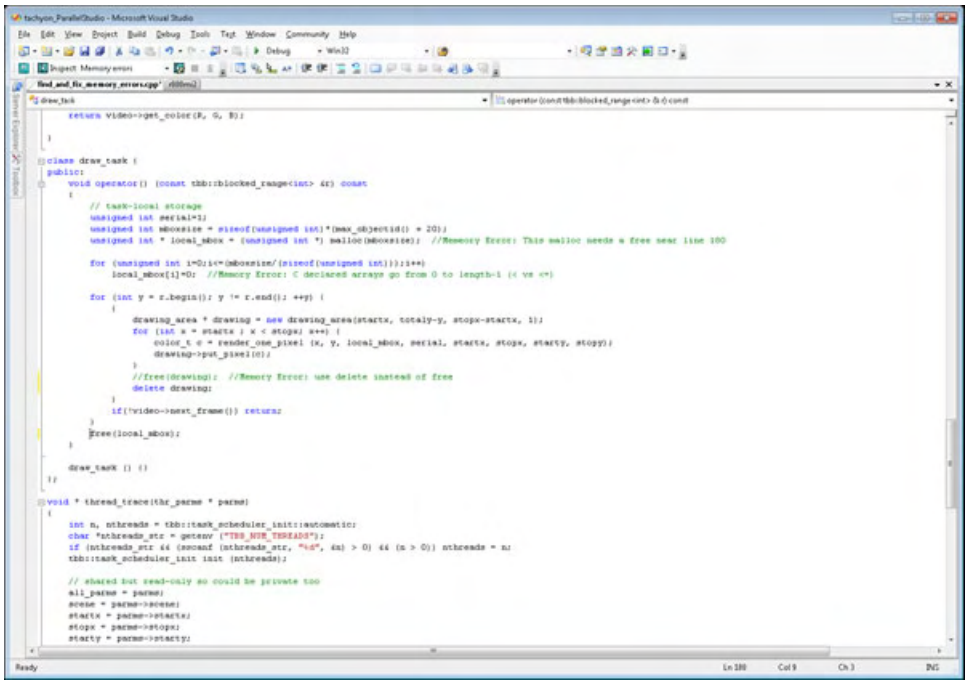
- If you do not deallocate soon enough, you waste space the program might need.
- If you deallocate too soon, you may do one of the following:
  - Corrupt your memory allocator or the space it manages when you later write to that space.
  - End up with bogus data if you later read that space when you previously returned it to the memory allocator.
- Make sure you free any space allocated within a function prior to exiting that function, or else make sure the function caller does the deallocation for the called function.

To resolve the issue:

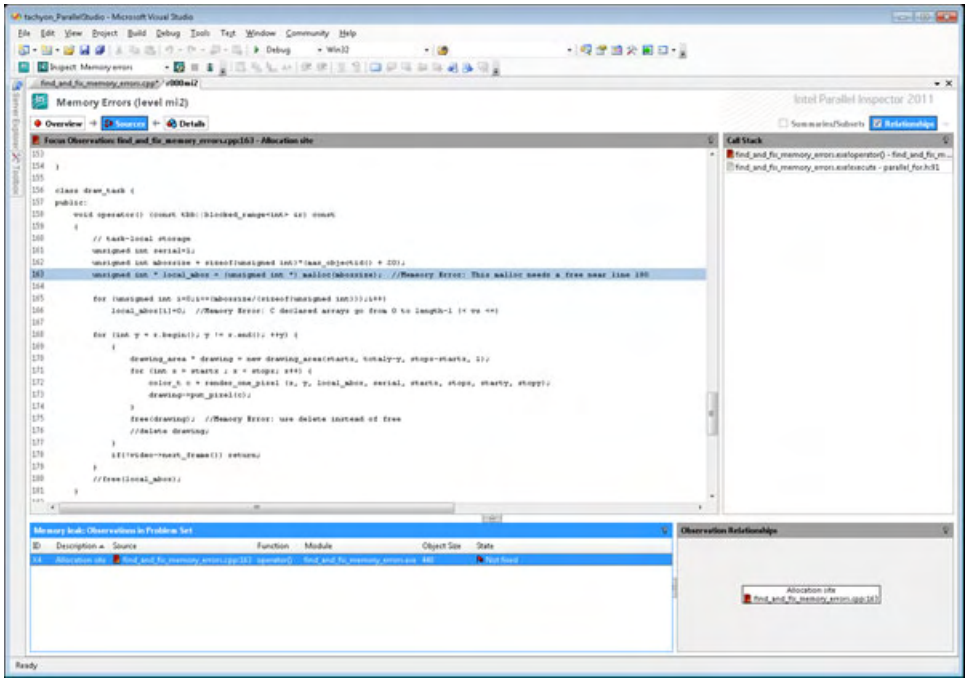
1. Click the **find_and_fix_memory_errors.cpp** tab to redisplay the `find_and_fix_memory_errors.cpp` source file in the Visual Studio* editor:



2. Uncomment `//free(local_mbox);:`



3. Click the **r000mi2** tab to redisplay the **Sources** window:



Recap

You:

1. Explored the memory leak problem detected in the `find_and_fix_memory_errors.cpp` source file.
2. Determined its cause: A `malloc()` call lacking a matching `free()` call.
3. Edited the source code to fix the problem.

Key Terms and Concepts

- Term: Observation
- Term: Problem
- Term: Problem set
- Term: Result

Rebuild and Rerun Analysis

To see if your edits resolved the memory errors:

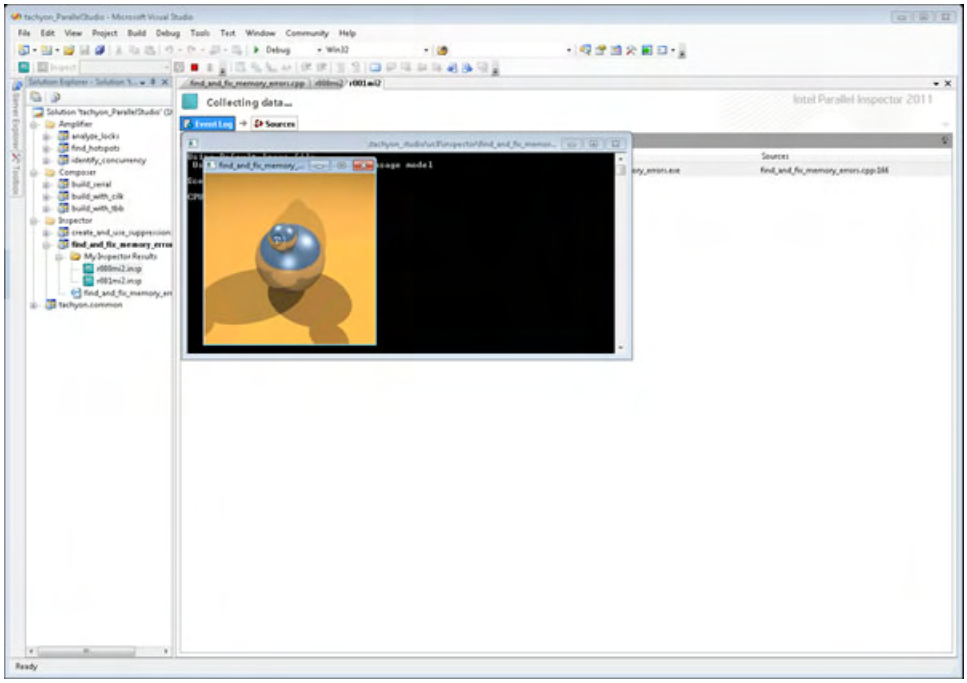
- Rebuild the application with your edited source code.
- Run another analysis.
- Start managing result data after collection.

To rebuild the application:

1. Right-click the **find_and_fix_memory_errors** project in the **Solution Explorer**.
2. Choose **Build**.

To run another analysis:

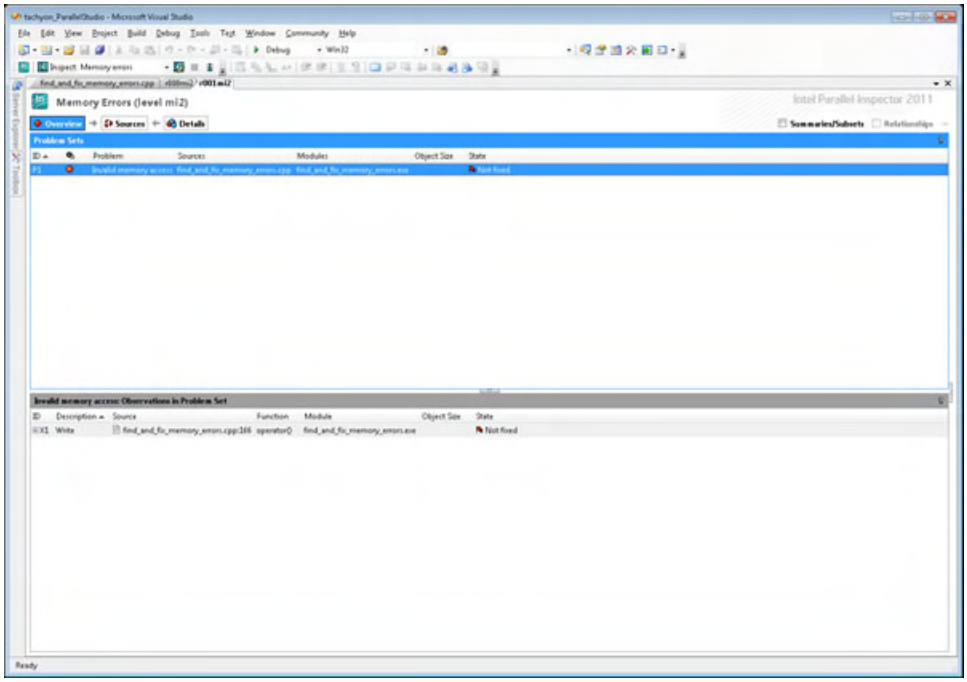
From the Visual Studio* menu, choose **Tools** > **Intel Parallel Inspector 2011** > **Re-inspect** to run another analysis of the same analysis type:



Notice the image now displays.

To start managing result data after collection:

Click the **Interpret Result** button to display the **Overview** window:



Notice the Inspector :

- Created a new result tab.
- No longer detected **Mismatched allocation/deallocation** and **Memory leak** problems.

Try fixing the remaining problem set(s) yourself.

Recap

You:

1. Recompiled and linked the **find_and_fix_memory_errors** application, saving your edited source code.
2. Reinspected the `find_and_fix_memory_errors.exe` file to verify you successfully resolved previously detected memory errors.

Key Terms and Concepts

- Term: Analysis

Summary

Here are some important things to remember when using the Inspector to manage memory errors:

Step 1. Choose Project and Build Application.

- Applications compiled/linked in debug mode using the following options produce the most accurate, complete results: `/Zi` or `/ZI`, `/Od`, and `/MD` or `/MDd`.
- You can control analysis cost without sacrificing completeness by using small, representative data sets. Data sets with runs in the seconds time range are ideal. You can always create additional data sets to ensure all your code is inspected.

Step 2. Collect Result.

Configure, then run an analysis:

- The Inspector offers preset memory analysis configurations to help you control analysis scope and cost. Widening analysis scope maximizes the load on the system, the time required to perform the analysis, and the chances the analysis will fail because the system may run out of resources.
- You can run memory error analyses from the **Tools** menu or toolbar. You can also use the `insp-cl` command.

Step 3. Manage Result.

Choose a problem set, interpret the result data, then resolve the issue:

- An observation is a fact the Inspector observes at a source code location. A problem is a small group of closely related observations (source code locations) that indicate an error in the target. A problem set is a larger group of more loosely related observations (source code locations) that could share a common solution.
- Think of the **Problems** pane on the **Overview** window as a *to-do* list: Start at the top and work your way down.
- Double-click an observation or problem set on the **Overview** window to navigate to the **Sources** window. Click the **Overview** button on the **Sources** window to return to the **Overview** window.
- Right-click an observation or problem set to display a pop-up menu, then choose **Explain Problem** to access more information on interpreting and resolving the problem.
- Double-click an observation on the **Sources** window to open the Visual Studio* editor.

Tutorial: Managing Threading Errors

2

Learning Objectives

This tutorial shows how to use the Intel® Parallel Inspector to identify, analyze, and resolve threading errors.

Estimated completion time: 10 - 15 minutes.

NOTE. This tutorial does not require prior completion of other tutorials. Consequently, some terms, concepts, and workflow steps are common to all tutorials.

After you complete this tutorial, you should be able to:

- List, in order, the steps to manage threading errors.
- Define key Inspector terms, such as *analysis*, *result*, *problem set*, *problem*, and *observation.*
- Identify compiler/linker options that produce the most accurate, complete analysis results.
- Explain how data set size and workload impact application execution time and analysis speed.
- Run threading error analyses.
- Influence analysis scope and running time.
- Access help for the Inspector command-line interface.
- Navigate among windows in Inspector results.
- Display a prioritized *to-do* list for resolving errors.
- Access help for resolving specific errors.
- Access source code to fix errors.

Key Terms and Concepts

Key Terms

analysis: A process during which the Inspector performs collection and finalization.

collection: A process during which the Inspector executes an application, identifies issues that may need handling, and collects those issues in a result.

false negative: The Inspector does not detect an error because the problem may be too complex/big or involve too much runtime/memory cost.

false positive: The Inspector detects something that is not an error.

finalization: A process during which the Inspector uses debug information from binary files to convert symbol information into filenames and line numbers, perform duplicate elimination, and form problem sets.

observation: A fact the Inspector observes at a source code location, such as a *write* observation. A focus observation is a source code location with relationships you choose to explore. A related observation is a source code location with a relationship to a focus observation and possibly other observations.

problem: A small group of closely related observations (source code locations) that indicate an error in an application, such as a *data race* problem.

problem set: A larger group of more loosely related observations (source code locations) that could share a common solution, such as a problem set resulting from deallocating an object too early during program execution. You can view problem sets only after analysis is complete.

project: A compiled application, collection of configurable attributes surrounding the compiled application, and a container for results.

result: A collection of issues that may need handling.

target: An application you inspect for errors using the Inspector.

Key Concept: Choosing Small, Representative Data Sets

When you run a dynamic analysis, the Inspector executes an application against a data set. Data set size and workload have a direct impact on application execution time and analysis speed.

For example, it takes longer to process a 1000x1000 pixel image than a 100x100 pixel image. A possible reason: You may have loops with an iteration space of 1...1000 for the larger image, but only 1...100 for the smaller image. The exact same code paths may be executed in both cases. The difference is the number of times these code paths are repeated.

You can control analysis *cost* without sacrificing completeness by removing this kind of redundancy from your data set.

Instead of choosing large, repetitive data sets, choose small, representative data sets that fully create threads with minimal to moderate work per thread. *Minimal to moderate* means just enough work to demonstrate all the different behaviors a thread can perform. Data sets with runs in the seconds time range are ideal. You can always create additional data sets to ensure all your code is inspected.

Key Concept: Ensuring Applications Contain More Than One Thread

If you plan to run the Inspector on systems with only one processor, you may need to take special steps to ensure applications create more than one thread:

Programming Model	Possible Strategies
Intel® Threading Building Blocks	When initializing an object of class `tbb::task_scheduler_init`, use the following function to force the creation of two or more threads: `initialize (int)`, where `int` = 2 or higher. For more information, see http://www.threadingbuildingblocks.org/

Programming Model	Possible Strategies
Windows* API (Win32)	Use the following function to create as many threads as you need: `CreateThread` - Creates a thread to execute within the virtual address space of the calling process. For more information, see http://msdn.microsoft.com/en-us/library/
OpenMP* API	Use the following to force the creation of two or more threads: • `OMP_NUM_THREADS` runtime environment variable - Sets the maximum number of threads in the parallel region, unless overridden by an `omp_set_num_threads` function or `num_threads` clause. • `omp_set_num_threads` function - Sets the number of threads in subsequent parallel regions, unless overridden by a `num_threads` clause. • `num_threads` clause - Sets the number of threads in a thread team. For more information, see http://msdn.microsoft.com/en-us/library/

Key Concept: Choosing Analysis Type Configurations

The Inspector offers preset memory and threading configurations to help you control analysis scope and *cost*.

Preset Configurations	Advantages	Disadvantages
Narrowest scope	• Minimizes the load on the system. • Minimizes the time required to perform the analysis. • Increases the chances the analysis will complete successfully - particularly on large applications/large data sets.	• Detects only simple errors. • Provides no details.
Intermediate scope(s)	• Detects important errors. • Provides some context and detail for detected errors.	• Increases the load on the system. • Increases the time required to perform the analysis. • Increases the chances the analysis will fail because the system may run out of resources.
Widest scope	• Detects important errors. • Provides context and maximum amount of detail for detected errors.	• Maximizes the load on the system. • Maximizes the time required to perform the analysis.

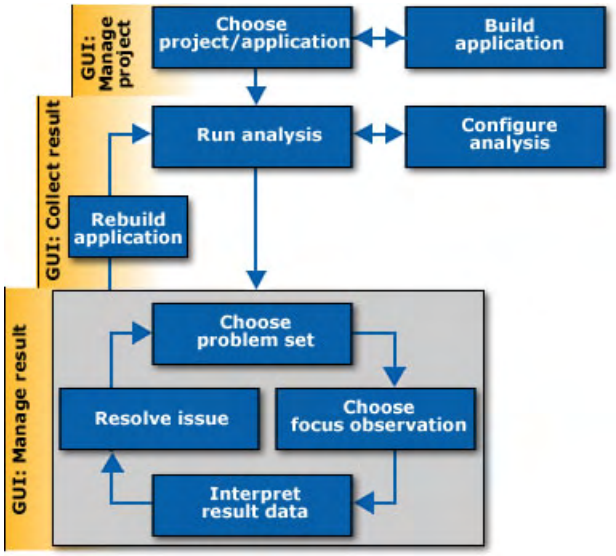
Preset Configurations	Advantages	Disadvantages
		• Maximizes the chances the analysis will fail because the system may run out of resources.

For more details, check the Inspector Help Index for *configuration*.

Workflow Steps to Identify, Analyze, and Resolve Threading Errors

You can use the Inspector to identify, analyze, and resolve threading errors in parallel programs by performing a series of steps in a workflow. This tutorial guides you through these workflow steps workflow steps using `tachyon_studio` sample code.

NOTE. Click Show Me for a video demonstration. Show Me Videos require Adobe* Flash* Player.



1. Choose a project.
2. Verify settings and build an application to inspect for errors.
3. Configure a threading error analysis.
4. Run the threading error analysis on the application.
5. Choose a problem set in the analysis result.
6. Choose a focus observation in the problem set.
7. Interpret the result data.

8. Resolve the issue.
9. Rebuild the application and rerun the threading error analysis.

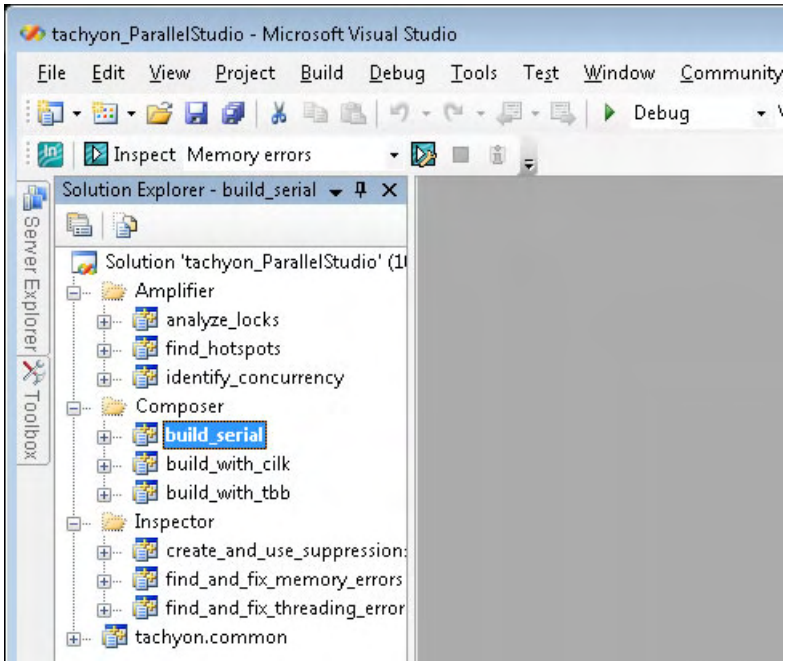
Choose Project

To enable Inspector functionality:

- Open a Visual Studio* solution.
- Set a startup project.

To open a Visual Studio* solution:

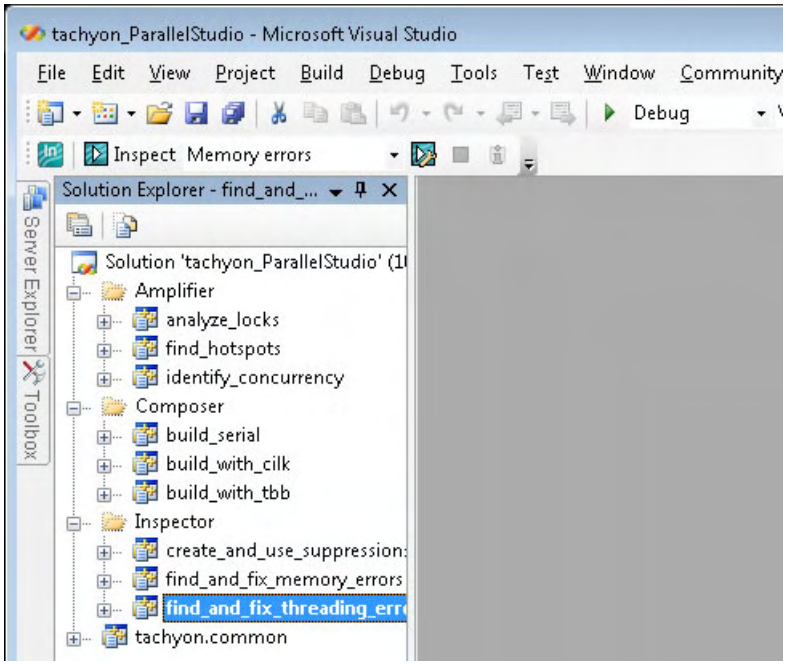
1. From the Visual Studio* menu, choose **File** > **Open** > **Project/Solution**.
2. In the **Open Project** dialog box, open the `tachyon_studio\vc8\tachyon_ParallelStudio.sln` file to display the **tachyon_ParallelStudio** solution in the **Solution Explorer**:



NOTE. See Prerequisites to locate and extract the `*.sln` file.

To set a startup project:

1. Right-click the **find_and_fix_threading_errors** project.
2. Choose **Set as Startup Project**:



Recap

You chose the **find_and_fix_threading_errors** project.

Key Terms and Concepts

- Concept: Choosing Small, Representative Data Sets
- Concept: Ensuring Applications Contain More Than One Thread

Build Application

You can use the Inspector on both debug and release modes of binaries containing native code; however, projects compiled/linked in debug mode using the following options produce the most accurate, complete results.

Compiler/Linker Options	Correct Setting	Impact If Not Set Correctly
Debug information	Enabled (`/Zi` or `/ZI`)	Missing file/line information
Optimization	Disabled (`/Od`)	Incorrect file/line information
Dynamic runtime library	Selected (`/MD` or `/MDd`)	False positives or missing observations

To enable correct analysis of Intel® Threading Building Blocks (Intel® TBB) applications, you must also set the following required macros before compiling:

- **TBB_USE_DEBUG** (which sets **TBB_USE_THREADING_TOOLS**) if you use Intel® TBB debug libraries

- **TBB_USE_THREADING_TOOLS** if you use Intel® TBB release libraries

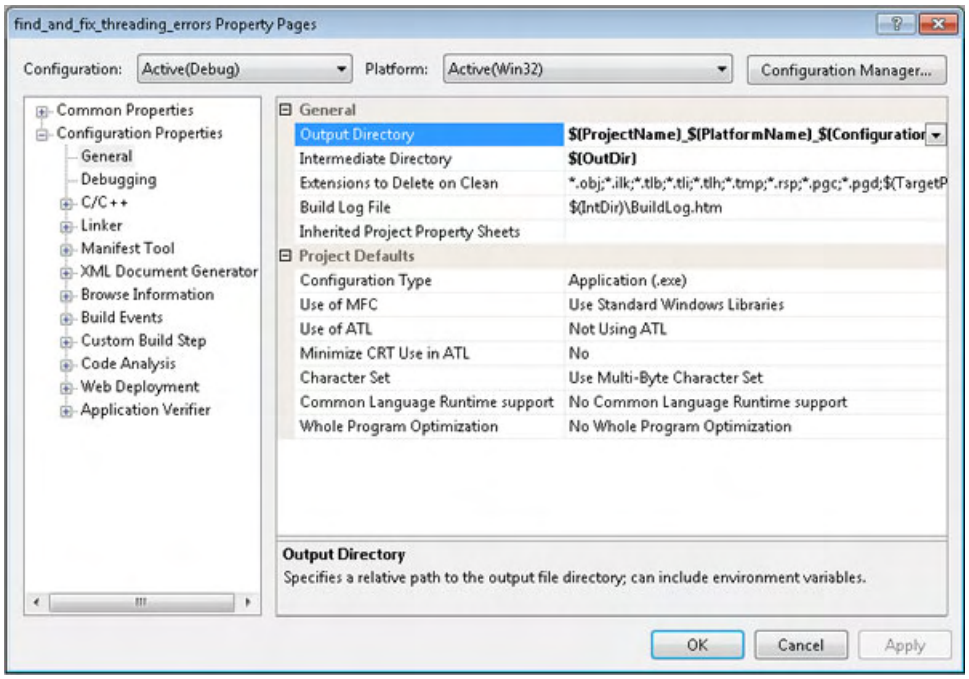
The impact if not set correctly: Intel Inspector XE may generate false positives. See Intel® TBB documentation for more information.

To create an application the Inspector can inspect for threading errors:

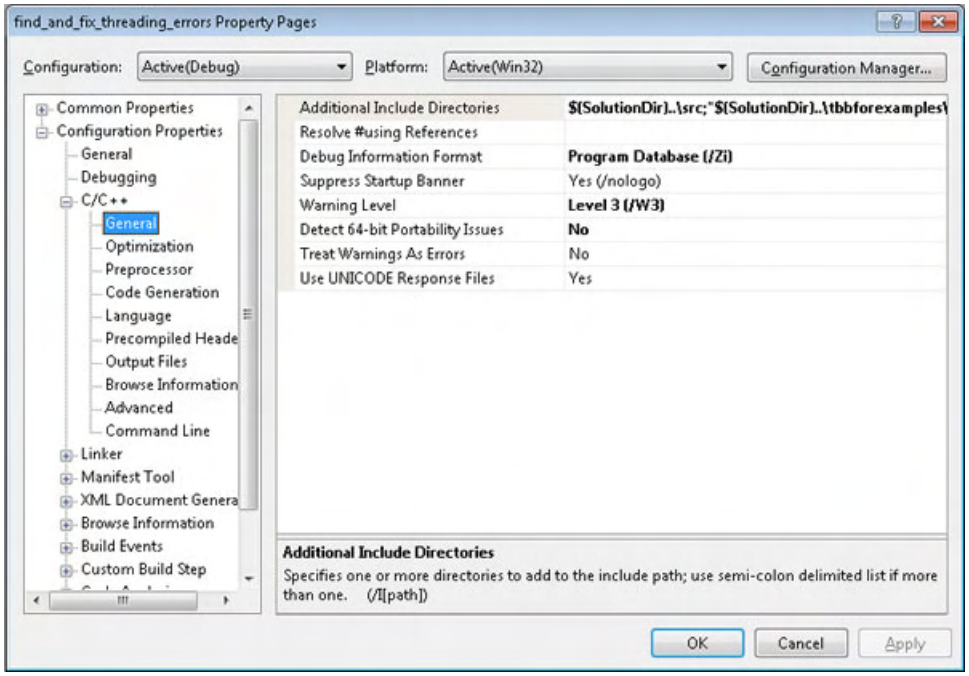
- Verify debug mode is configured to produce the best results.
- Verify the application is set to build in debug mode.
- Build the application.

To verify debug mode is configured to produce the best results:

1. Right-click the **find_and_fix_threading_errors** project in the **Solution Explorer**.
2. Choose **Properties** to display the **Property Pages** dialog box.
3. Verify the **Configuration** drop-down list is set to **Debug** or **Active(Debug)**.

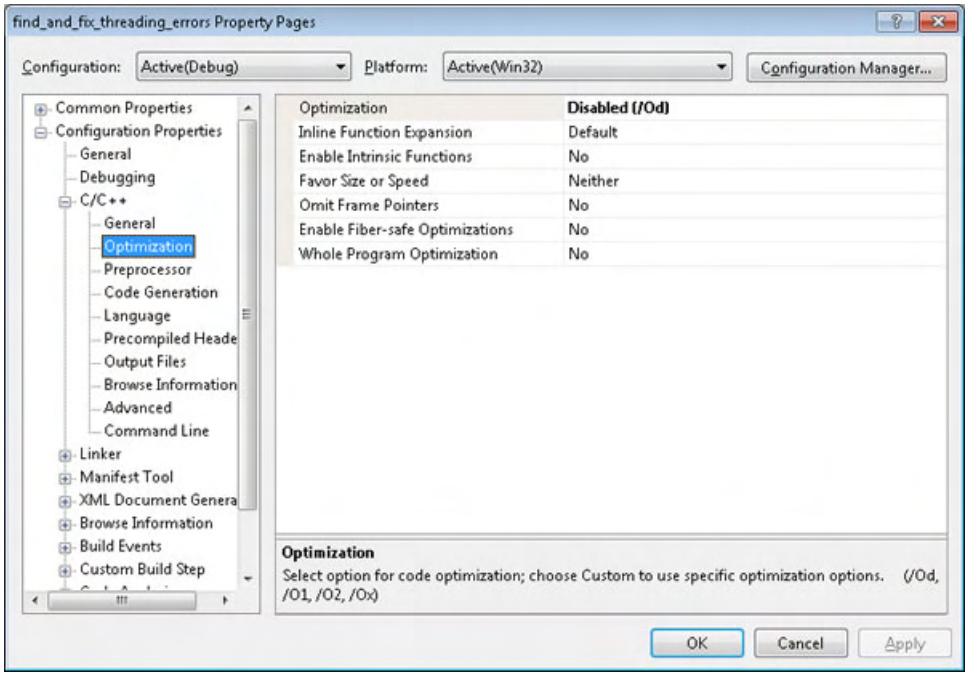


4. In the left pane, choose **Configuration Properties** > **C/C++** > **General**.
5. Verify the **Debug Information Format** is set to **Program Database (/Zi)** or **Program Database for Edit & Continue (/ZI)**.



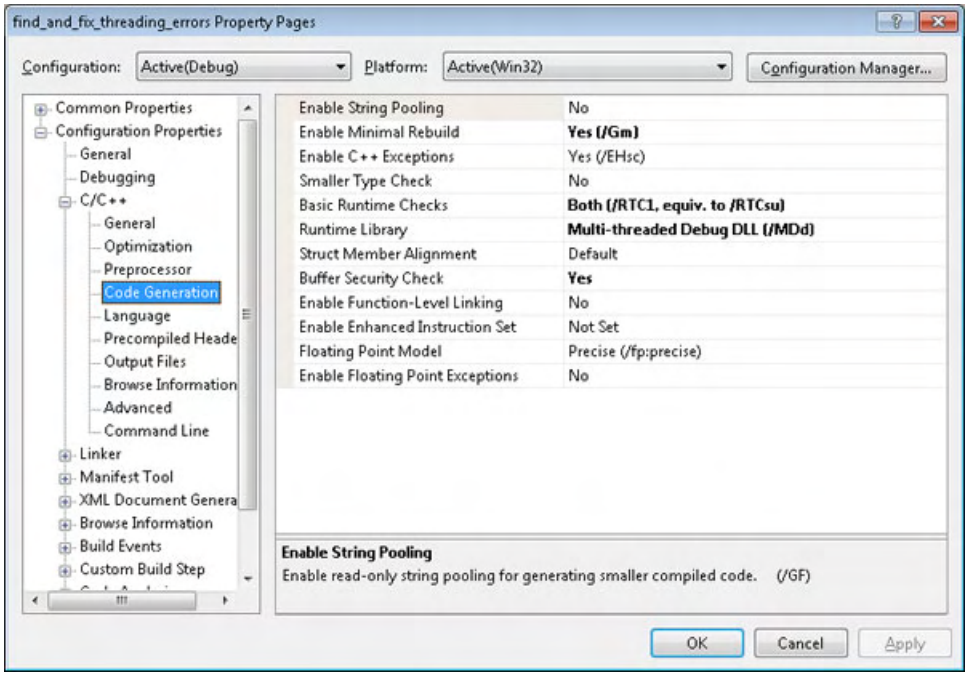
6. In the left pane, choose **Configuration Properties** > **C/C++** > **Optimization**.

7. Verify the **Optimization** field is set to **Disabled (/Od)**.

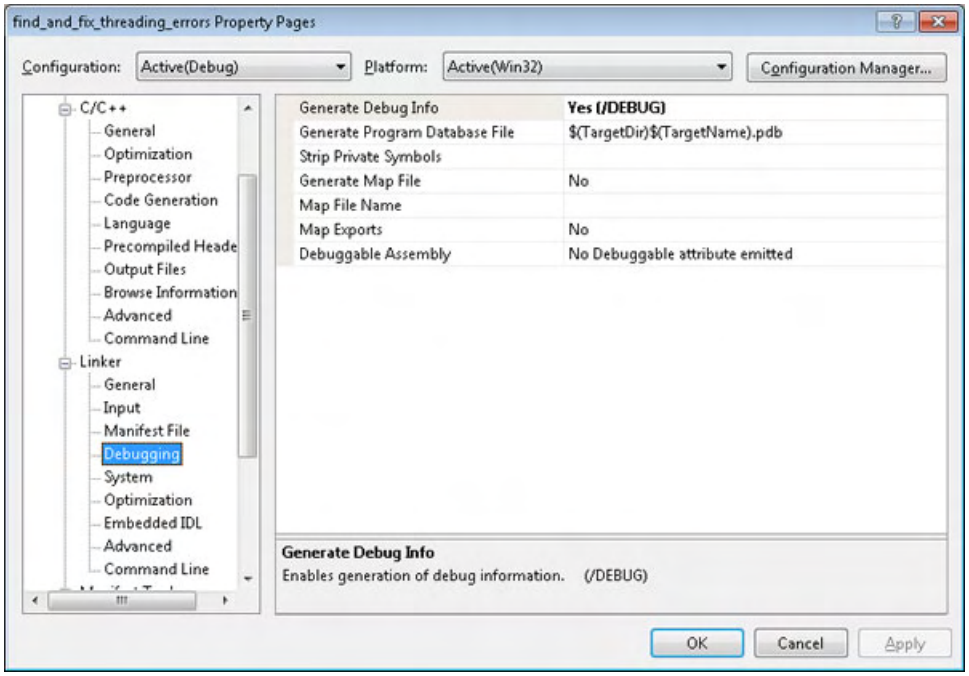


8. In the left pane, choose **Configuration Properties** > **C/C++** > **Code Generation**.

9. Verify the **Runtime Library** field is set to **Multi-threaded DLL (/MD)** or **Multi-threaded Debug DLL (/MDd)**.



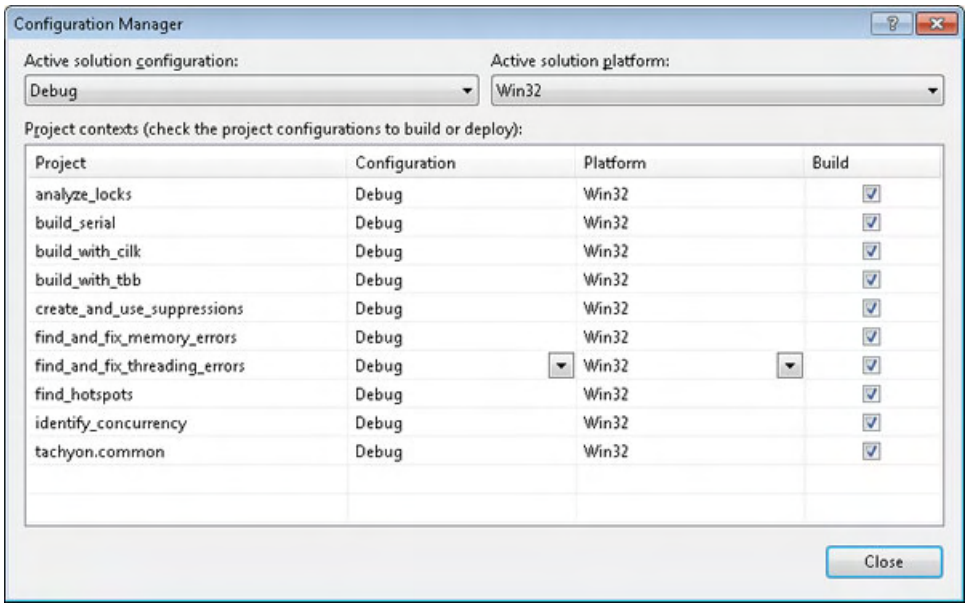
10. In the left pane, choose **Configuration Properties** > **Linker** > **Debugging**.
11. Verify the **Generate Debug Info** field is set to **Yes (/DEBUG)**.



To verify the application is set to build in debug mode:

1. Click the **Configuration Manager** button.

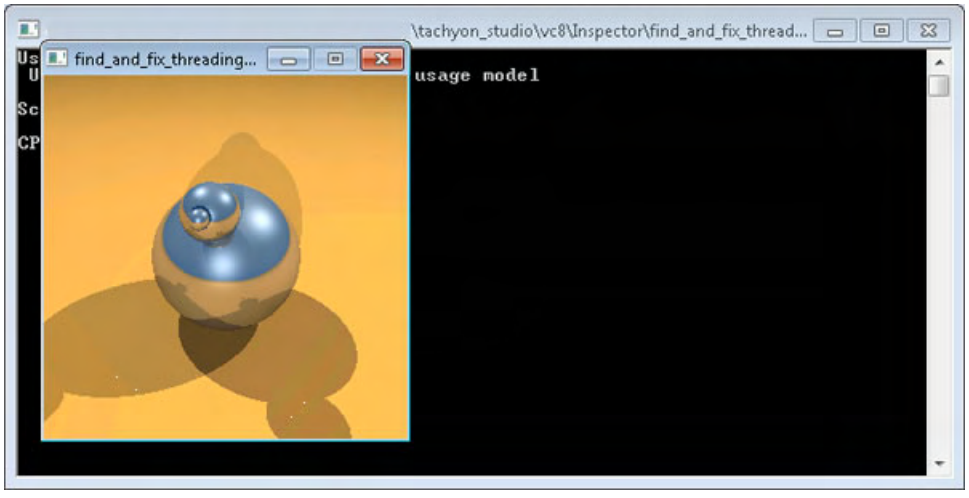
2. Verify the **Active solution configuration** drop-down list is set to **Debug**.



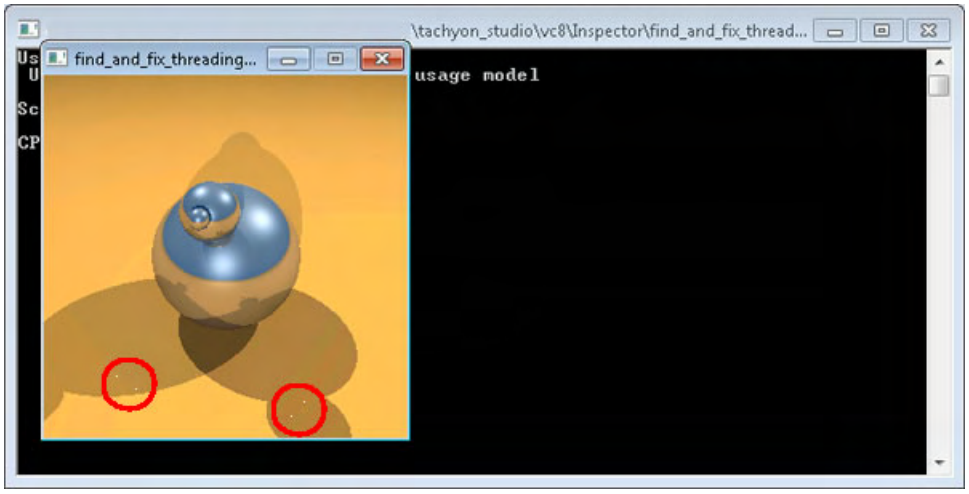
3. Click the **Close** button to close the **Configuration Manager** dialog box.
4. Click the **OK** button to close the **Property Pages** dialog box.

To build the application:

1. From the Visual Studio* menu, choose **Debug** > **Start Without Debugging**.
2. If the Visual Studio* IDE responds any projects are out of date, click **Yes** to build.
3. Check for a display similar to the following:



Notice the white dots in the the image. The cause: Threading errors.



Recap

You verified the **find_and_fix_threading_errors** project is set to produce the most accurate, complete results; compiled and linked the application; and ensured the resulting `find_and_fix_threading_errors.exe` file runs on your system outside the Inspector.

Key Terms and Concepts

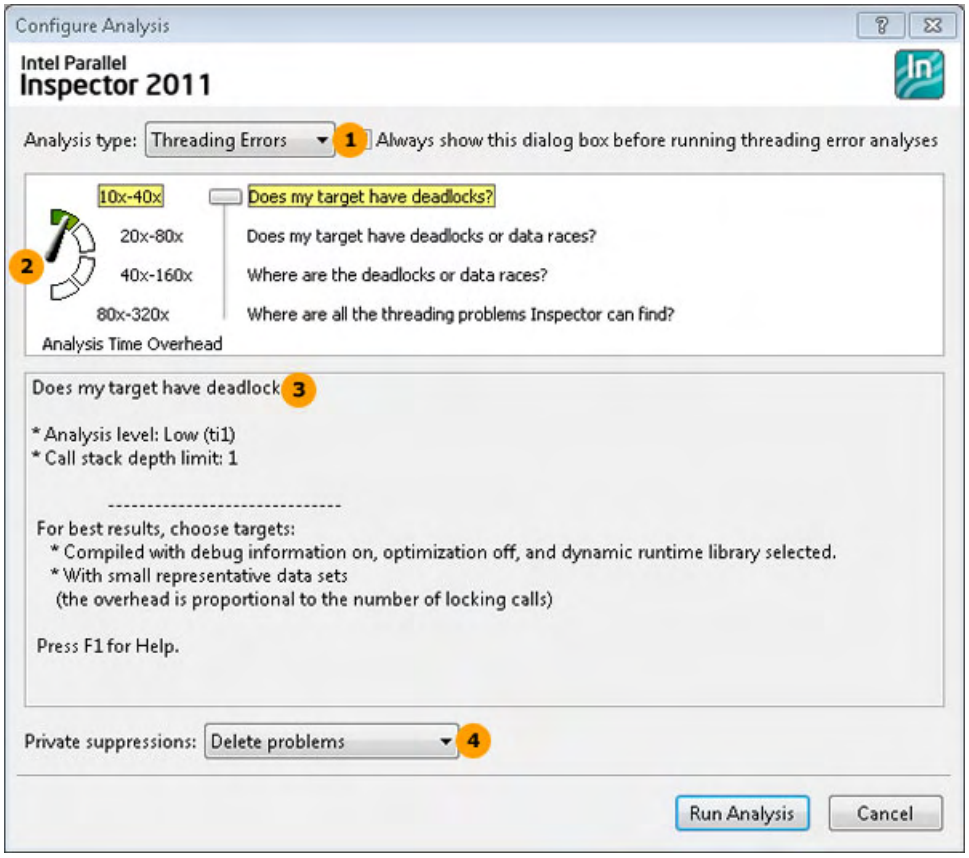
- Term: False negative
- Term: False positive

Configure Analysis

Choose a preset configuration to influence threading error analysis scope and running time.

To configure a threading error analysis:

1. From the Visual Studio* menu, choose **Tools** > **Intel Parallel Inspector 2011** > **Inspect Threading Errors** to display the **Configure Analysis** dialog box:



1. The **Analysis type** drop-down list shows Inspector dynamic analysis type offerings: Memory error analysis and threading error analysis.

   This tutorial covers threading error analyses, which you can use to search for these kinds of errors: Data race, deadlock, lock hierarchy violation, and cross-stack thread access.

   Use memory error analyses to search for these kinds of errors: GDI resource leak, kernel resource leak, invalid memory access, memory leak, mismatched allocation/deallocation, missing allocation, and uninitialized memory access.

2. The **Analysis Time Overhead** gauge shows the time it may take to collect a result at each preset configuration. Time is expressed in relation to normal application execution time. For example, 2x - 20x is 2 to 20 times longer than normal application execution time. If normal application execution time is 5 seconds, estimated collection time is 10 to 100 seconds.

   Here, the configuration slider rests at the **Does my target have deadlocks?** preset configuration, which has the narrowest scope/lowest overhead. Try dragging the configuration slider to see the impact on the gauge.

3. The configuration details region shows current configuration characteristics. Try dragging the configuration slider to see the impact on the details region.

4. *Suppressing* known issues can dramatically improve your productivity. For more details, check the Inspector Help Index for *suppression rule*.

2. Drag the configuration slider to the **Does my target have deadlocks or data races?** preset configuration:

Recap

You chose a preset configuration of intermediate scope to detect threading errors in the `find_and_fix_memory_errors.exe` file.

Key Terms and Concepts

- Term: Analysis
- Term: Target
- Concept: Choosing Analysis Type Configurations

Run Analysis

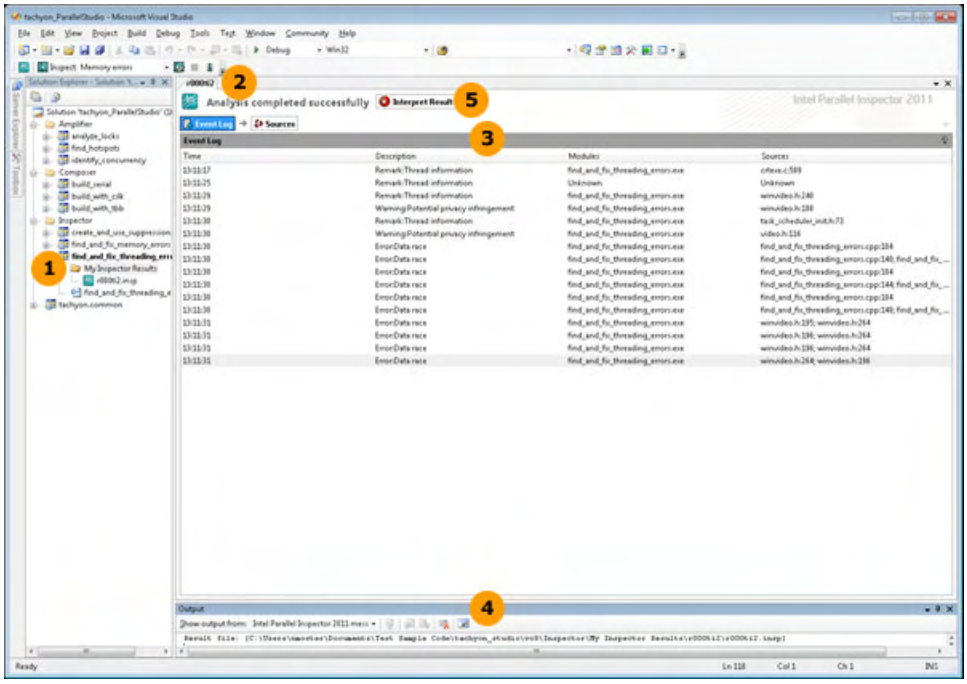
Run a threading error analysis to detect threading errors that may need handling.

To run a threading error analysis:

Click the **Run Analysis** button on the **Configure Analysis** dialog box to:

- Execute the `find_and_fix_threading_errors.exe` file.
- Identify threading errors that may need handling.
- Collect the result in a folder in the `tachyon_studio\vc8\Inspector\My Inspector Results` folder.
- Finalize the result (convert symbol information into filenames and line numbers, perform duplicate elimination, and form problem sets).

During collection, the Inspector displays a **Collection Log** window similar to the following:



1. The Visual Studio* IDE offers a pointer to the result from the **Solution Explorer** to provide easy future access. Here, the Visual Studio* IDE created a **find_and_fix_threading_errors\My Inspector Results** folder in the **Solution Explorer** with a pointer to the **r000ti2** result.

2. The result name appears in the tab. Here, the name of the result (and the name of the result folder in the `tachyon_studio\vc8\Inspector\My Inspector Results` folder) is `r000ti2`, where:
   - `r` = constant
   - `000` = next available number
   - `ti` = threading error analysis type
   - `2` = preset configuration of intermediate scope

3. The **Collection Log** pane shows detected problems in real time.

   This tutorial does not cover examining and solving problems during collection. For more details, check the Inspector Help Index for *examine result during analysis*.

4 The Visual Studio* **Output** window shows Inspector analysis status messages.

If necessary, choose **Intel Parallel Inspector messages** in the **Show output from** drop-down list to see the status messages.

5 The **Interpret Result** button displays after analysis (both collection and finalization) completes successfully.

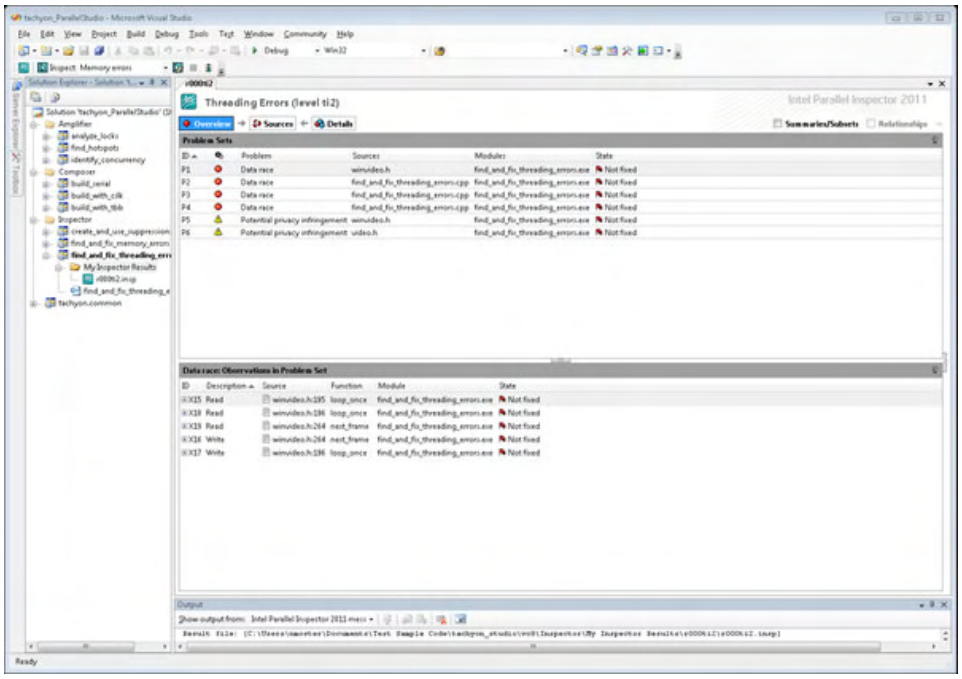
You can examine and solve problems during collection - and continue to examine and solve problems after collection and finalization are complete - but there are trade-offs involved.

TIP. Wait until analysis is complete, then click the **Interpret Result** button to use the full range of Inspector capabilities to manage result data in a more efficient, effective, and productive manner.

NOTE. This tutorial explains how to run an analysis from the Inspector graphical user interface (GUI). You can also use the Inspector command-line interface (`insp-cl` command) to run an analysis. For more details, check the Inspector Help Index for *insp-cl command*.

To start managing result data after analysis is complete:

Click the **Interpret Result** button to display the **Overview** window:



Recap

You inspected the `find_and_fix_threading_errors.exe` application for threading errors, during which:

- The Inspector ran the `find_and_fix_threading_errors.exe` file, identified threading errors that may need handling, collected a result in the `tachyon_studio\vc8\Inspector\My Inspector Results\r000ti2` folder, converted symbol information into filenames and line numbers, performed duplicate elimination, and formed problem sets.
- The Visual Studio* IDE added a pointer to the `r000ti2` result from the **Solution Explorer**.

Key Terms and Concepts

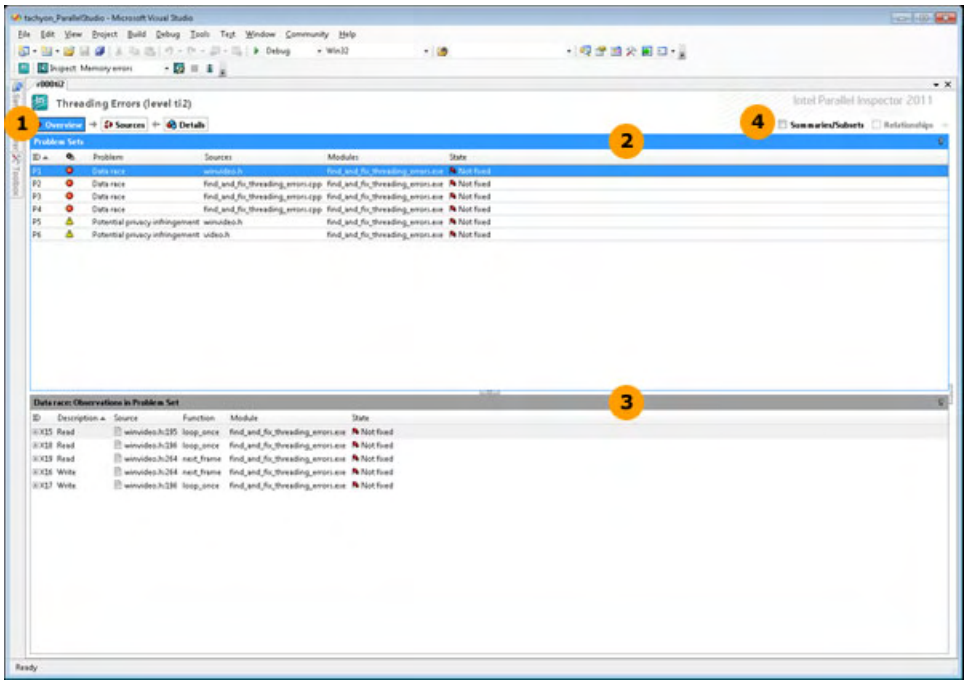
- Term: Analysis
- Term: Collection
- Term: Finalization
- Term: Problem Set
- Term: Result

Choose Problem Set and Focus Observation

To start exploring a detected threading error:

- Interpret the window.
- Choose a problem set.
- Choose a focus observation.

To interpret the Overview window:



1. The **Overview** window is the default starting point for managing result data. It groups observations into problem sets, and then prioritizes the problem sets by severity and size.

   The **Details** window is the advanced starting point for managing result data - it does not group observations into problem sets. This tutorial does not cover searching for observations that answer specific questions only you can formulate. For more details, check the Inspector Help Index for *select/observation search criteria*.

   The arrows in the **Navigation** toolbar indicate you can access the **Sources** window from both the **Overview** and **Details** windows.

2. Think of the **Problems** pane as a *to-do* list. Start at the top and work your way down.

3. The **Code Locations in Problem(s)** pane shows all the observations in all the problems in the selected problem set. By default, the Inspector selects the first problem set for you.

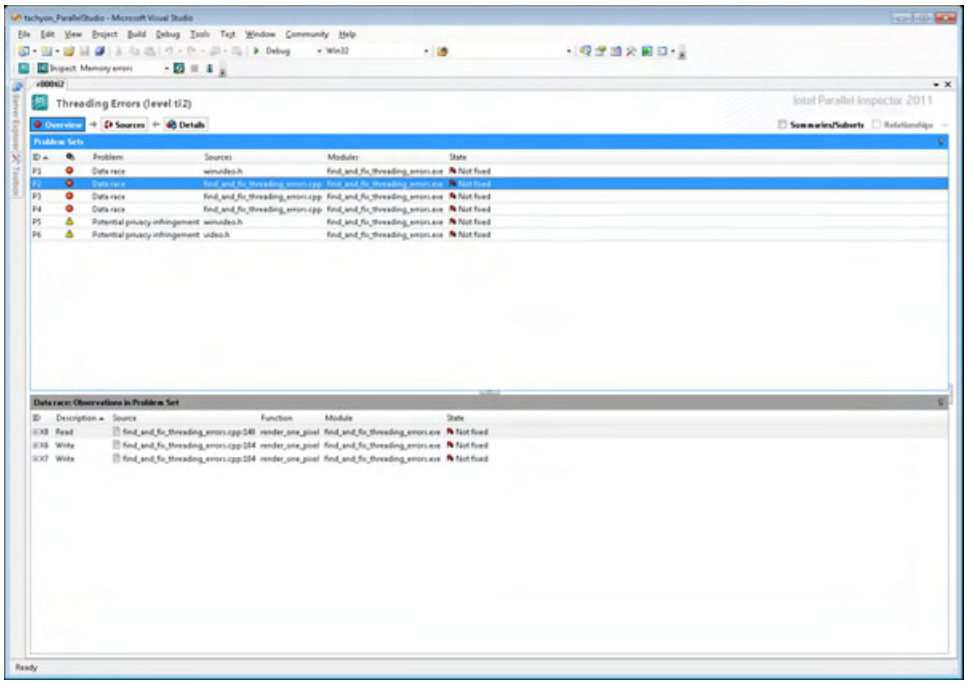
   Here, the Inspector selected a **Data race** problem set, which contains one or more problems composed of several **Read** and **Write** observations in the `winvideo.h` source file.

4. The **Filters** pane (toggled off by default) categorizes all displayed problem sets by specific criteria. Use it to temporarily limit the list in the **Problems** pane to only those problem sets that meet specific criteria.

   This tutorial does not cover searching for specific problem sets. For more details, check the Inspector Help Index for *select/problem set search criteria*.

To choose a problem set:

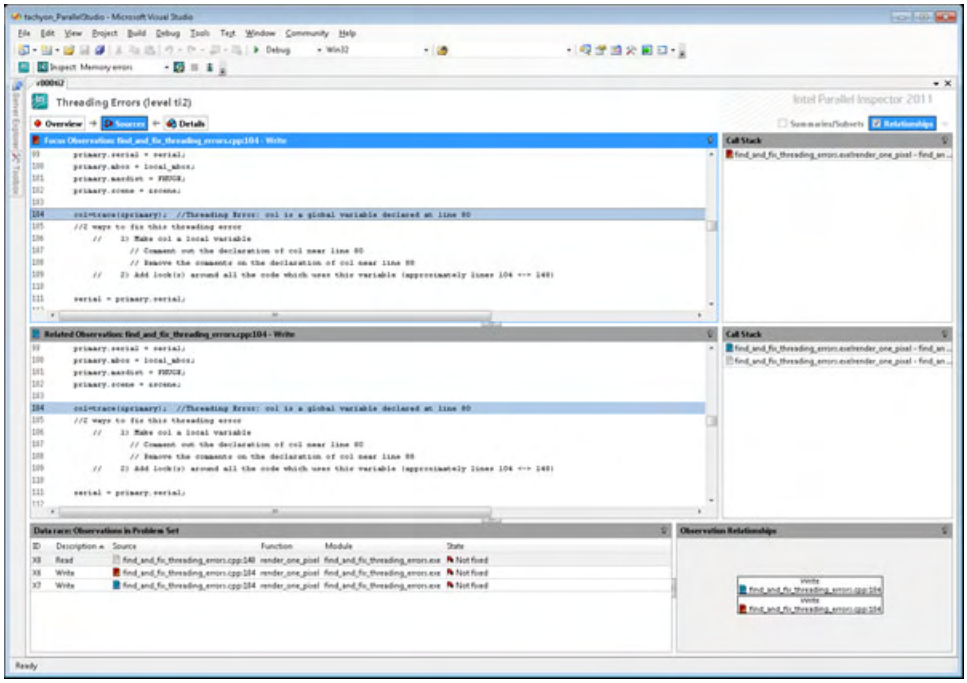
Click the data row for the **Data race** problem set in the `find_and_fix_threading_errors.cpp` source file to display a window similar to the following:



This **Data race** problem set contains one or more problems composed of **Read** and **Write** observations.

To choose a focus observation:

Double-click the data row for the X6 **Write** observation to display the **Sources** window, which provides more visibility into the error:



Recap

You started exploring a **Data race** problem set that contains one or more problems composed of **Read** and **Write** observations.

Key Terms and Concepts

- Term: Focus observation
- Term: Observation
- Term: Problem
- Term: Problem set
- Term: Result

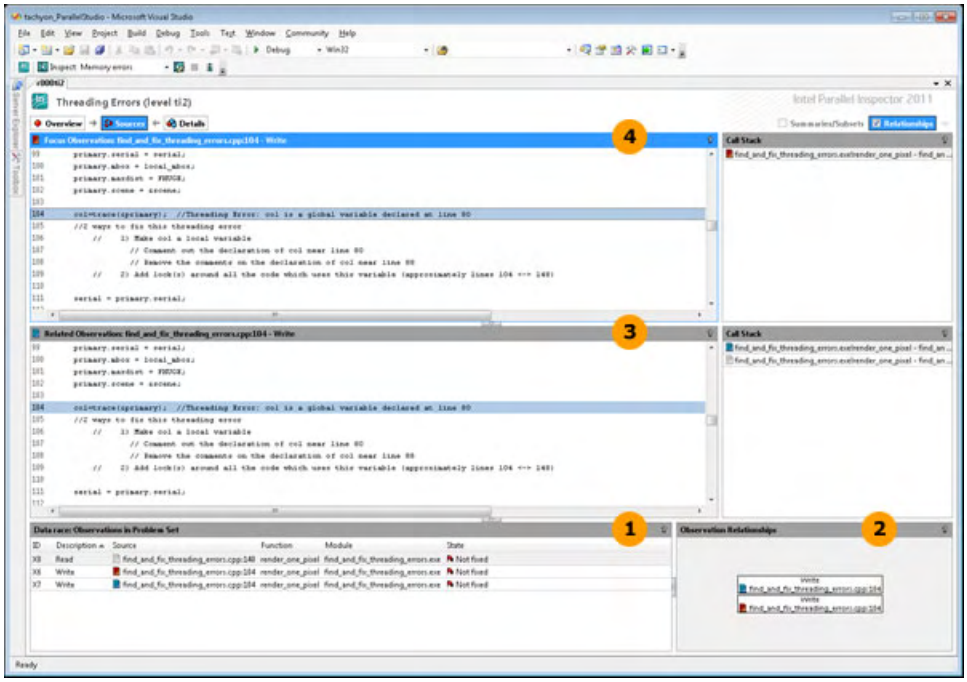
Interpret Result Data

To determine the cause of the detected threading error:

- Interpret the window.
- Interpret window icons.

- View source code for another observation.
- Understand the issue.
- Access more information on interpreting and resolving problems.

To interpret the Sources window:



1 Like the pane on the **Overview** window, the **Code Locations in Problem(s)** pane shows all the observations in one Write -> Write **Data race** problem and one Write -> Read **Data race** problem in the **Data race** problem set.

The Write -> Write **Data race** problem contains two observations:

- The X6 **Write** observation represents the instruction and associated call stack of the thread responsible for a memory write.
- The X7 **Write** observation represents the instruction and associated call stack of the thread responsible for a concurrent memory write.

The Write -> Read **Data race** problem also contains two observations:

- The X6 **Write** observation represents the instruction and associated call stack of the thread responsible for a memory write.
- The X8 **Read** observation represents the instruction and associated call stack of the thread responsible for a concurrent memory read.

Notice the X6 **Write** observation is in both problems.

2 The **Relationships** pane shows the relationship between the two **Write** observations in the Write -> Write **Data race** problem. In relationship diagrams:

- Each box in a diagram represents an observation in one problem in a problem set.
- A diagram with a single box is a trivial problem with no related observation.
- Vertically stacked boxes indicate concurrent observations. Here, the two **Write** observations occur concurrently.
- Boxes arranged left-to-right with connecting arrows indicate a time ordering.
- Boxes with connecting lines indicate association.

3 The **Related Code Location** pane shows the source code in the `find_and_fix_threading_errors.cpp` source file surrounding the X7 **Write** observation. (Notice the icon in the pane title matches the icon on the X7 **Write** observation data row in the **Code Locations in Problem(s)** pane.) The source code corresponding to the **Write** observation is highlighted.

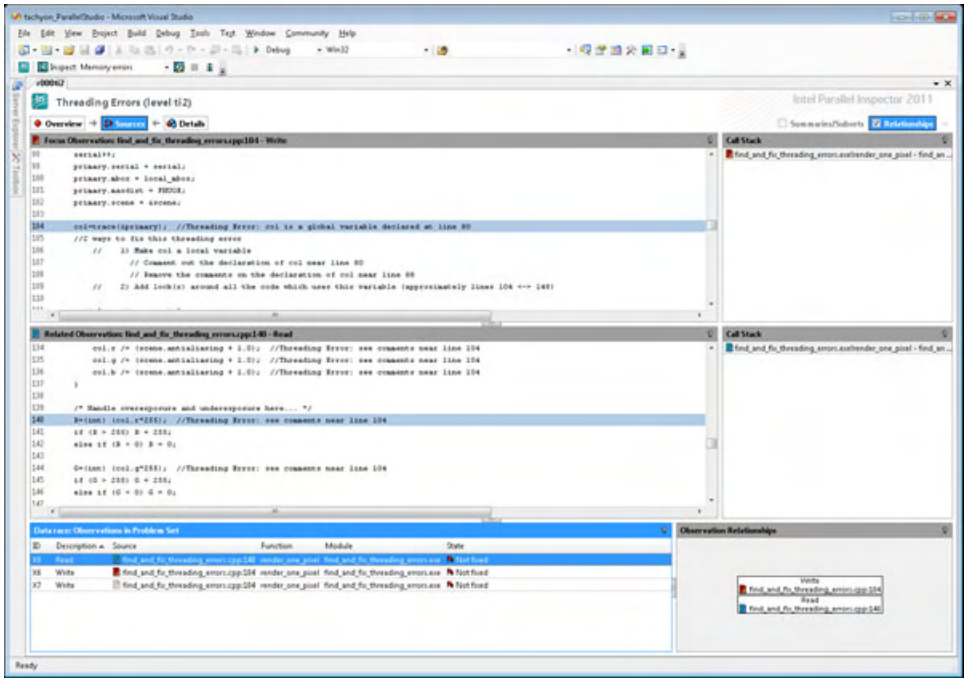
4 The **Focus Code Location** pane shows the source code in the `find_and_fix_threading_errors.cpp` source file surrounding the X6 **Write** observation. (Notice the icon in the pane title matches the icon on the X6 **Write** observation data row in the **Code Locations in Problem(s)** pane.) The source code corresponding to the **Write** observation is highlighted.

To interpret window icons:

Icon	Meaning	
	This observation is the focus observation. You chose it when you double-clicked the X6 **Write** observation on the **Overview** window. Its source code is currently displayed in the **Focus Code Location** pane.	Observation source code is available for viewing in the Inspector and editing in the Visual Studio* code editor.
	This observation is related to the focus observation. Its source code is currently displayed in the **Related Code Location** pane.	
	This is another observation in the problem or problem set. Its source code is not currently displayed onscreen.	

To view source code for another observation:

Double-click the data row for the **Read** observation in the **Code Locations in Problem(s)** pane to display a window similar to the following:



Notice the window changes:

- The **Relationships** pane now shows the relationship between the **Write** and **Read** observations in the Write -> Read **Data race** problem.
- The **Related Code Location** pane now shows the source code for the **Read** observation.
- The icon for the **Read** observation is now instead of in the **Code Locations in Problem(s)** pane (and throughout the **Sources** window).

To understand the issue:

A Write -> Write **Data race** problem occurs when multiple threads write to the same memory location without proper synchronization. A Write -> Read **Data race** problem occurs when one thread writes to, while a different thread concurrently reads from, the same memory location.

The commenting in the **Focus Code Location** window identifies the cause of the **Data race** problems: Multiple threads are concurrently accessing the global variable `col`. One possible correction strategy: Change the global variable to a local variable.

To access more information on interpreting and resolving problems:

1. Right-click any observation in the **Code Locations in Problem(s)** pane or box in the **Relationships** pane.
2. Choose **Explain Problem** to display Inspector Help information for the **Data race** problem type.

Recap

You determined the cause of a **Data race** problem set in the `find_and_fix_threading_errors.cpp` source file: Multiple threads are concurrently accessing the global variable `col`.

Key Terms and Concepts

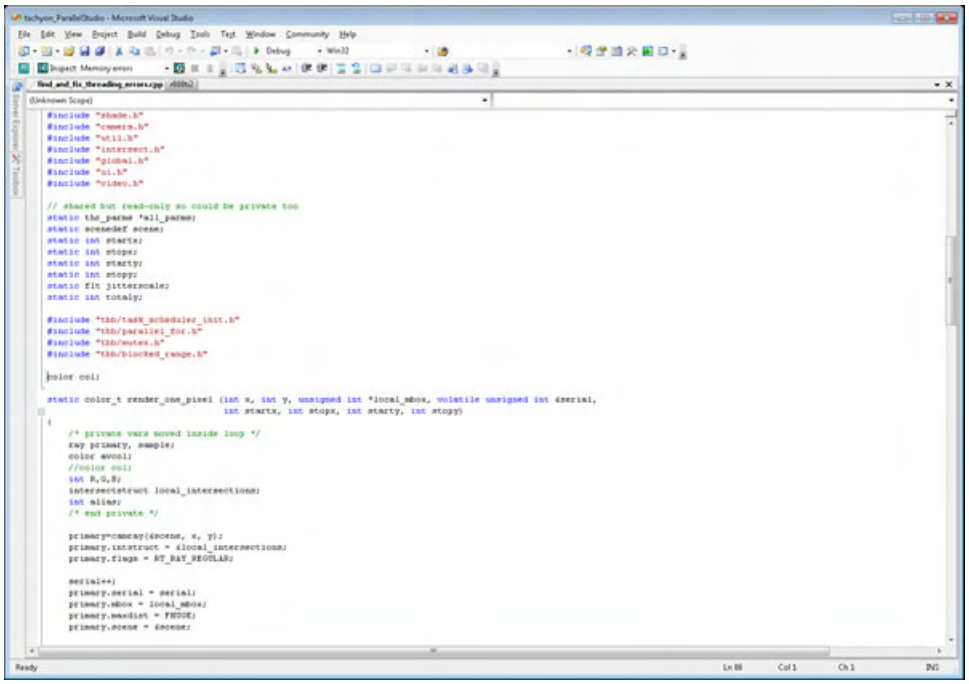
- Term: Focus observation
- Term: Observation
- Term: Problem
- Term: Problem set
- Term: Related observation
- Term: Result

Resolve Issue

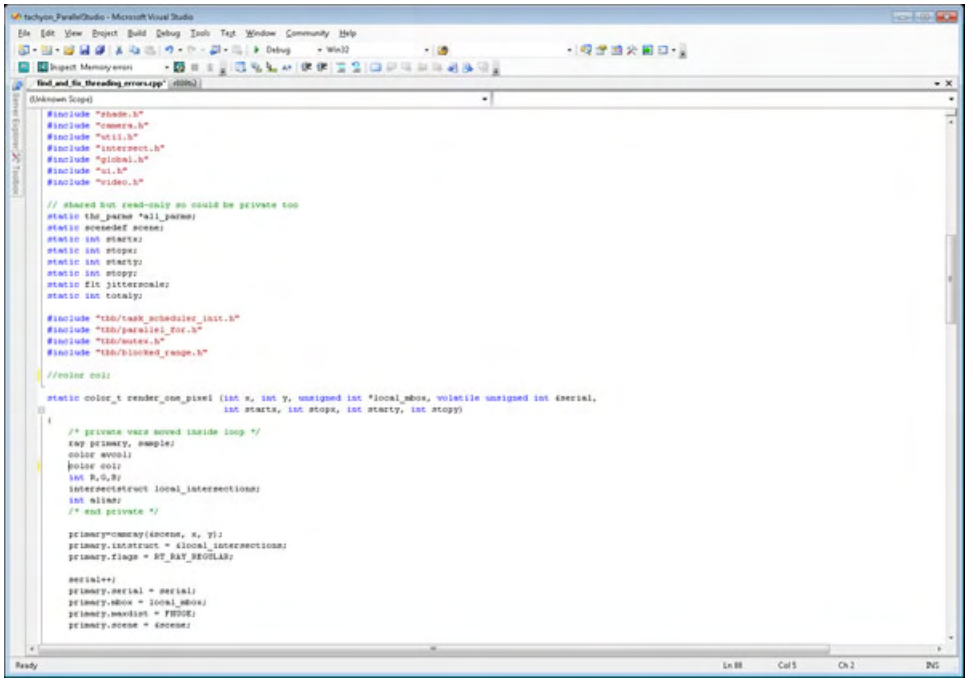
Access the Visual Studio* editor to fix the threading error.

To resolve the issue:

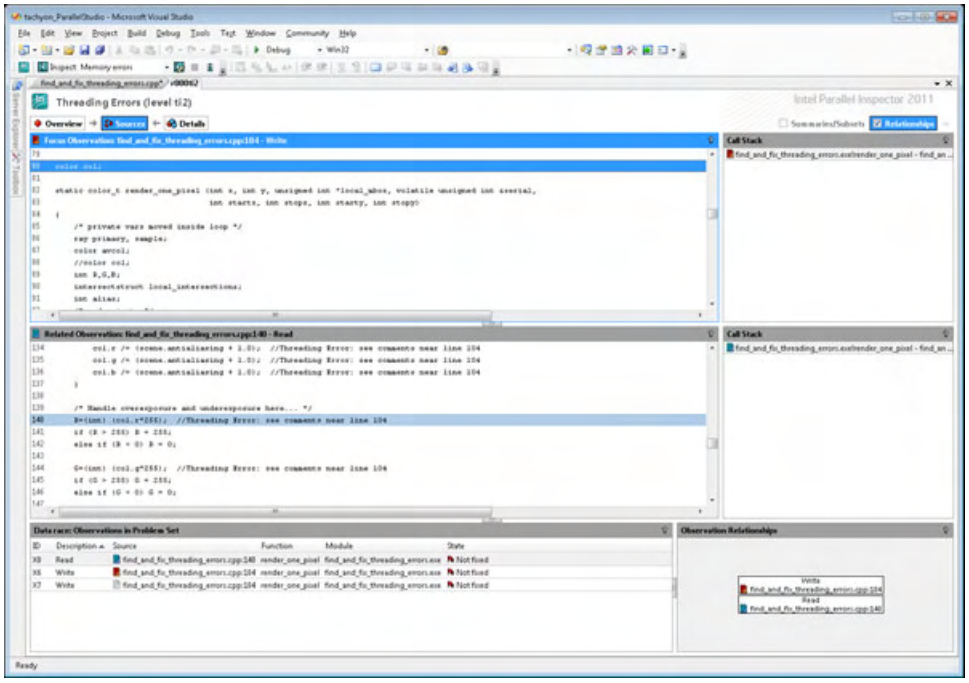
1. Scroll to this source code near line 80 in the **Focus Code Location** pane: `color col;`
2. Double-click the line to open the `find_and_fix_threading_errors.cpp` source file in a separate tab where you can edit it with the Visual Studio* editor:



3. Comment `color col;` and uncomment `//color col;` near line 89:



4. Click the **r000ti2** tab to redisplay the **Sources** window:



Recap

You edited the `find_and_fix_threading_errors.cpp` source file to fix the problems in the **Data race** problem set.

Key Terms and Concepts

- Term: Observation
- Term: Problem
- Term: Problem set
- Term: Result

Rebuild and Rerun Analysis

To see if your edits resolved the threading errors:

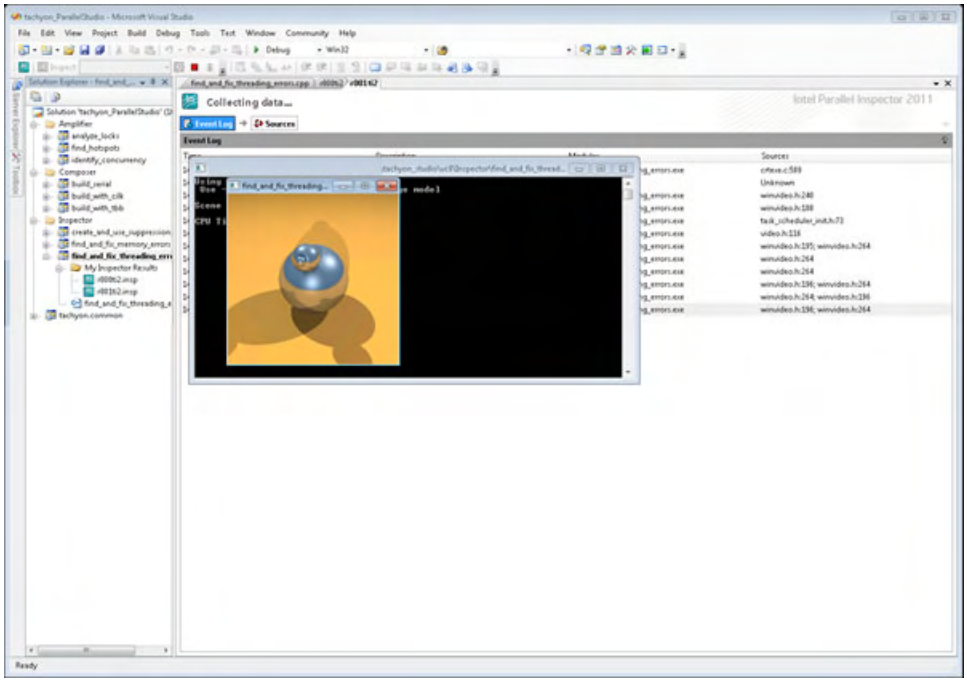
- Rebuild the application with your edited source code.
- Run another analysis.
- Start managing result data after collection.

To rebuild the application:

1. Right-click the **find_and_fix_threading_errors** project in the **Solution Explorer**.
2. Choose **Build**.

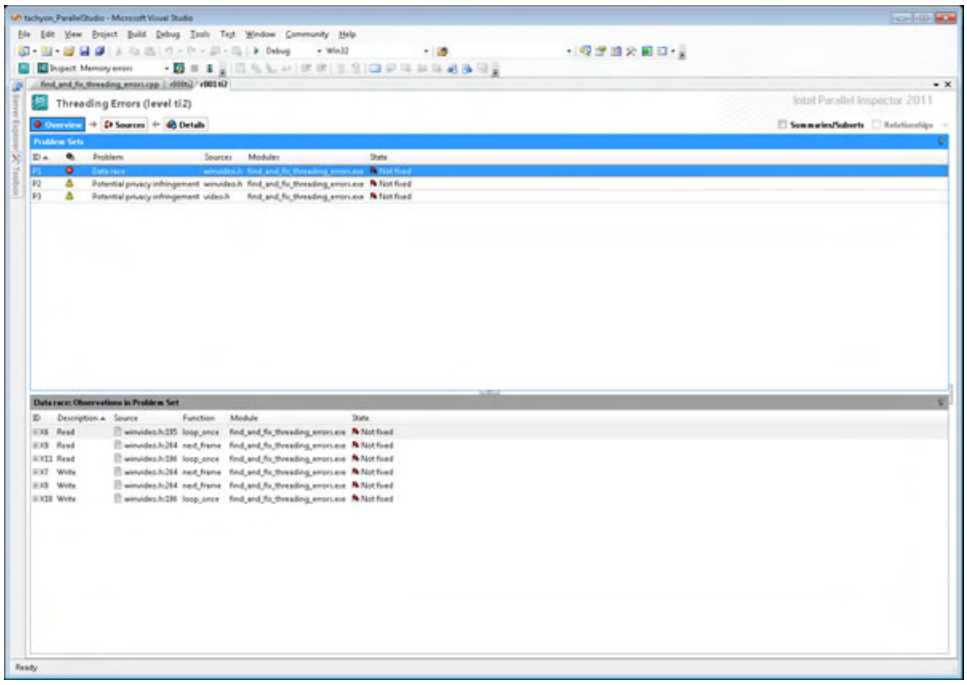
To run another analysis:

From the Visual Studio* menu, choose **Tools** > **Intel Parallel Inspector 2011** > **Re-inspect** to run another analysis of the same analysis type:



To start managing result data after collection:

Click the **Interpret Result** button to display the **Overview** window:



Notice the Inspector :

- Created a new result tab.

- No longer detects problems in the `find_and_fix_threading_errors.cpp` file.

Recap

You:

1. Recompiled and linked the **find_and_fix_threading_errors** application, saving your edited source code.
2. Reinspected the `find_and_fix_threading_errors.exe` file to verify you successfully resolved previously detected threading errors.

Key Terms and Concepts

- Term: Analysis

Summary

Here are some important things to remember when using the Inspector to manage threading errors:

Step 1. Choose Project and Build Application.

- Applications compiled/linked in debug mode using the following options produce the most accurate, complete results: `/Zi` or `/ZI`, `/Od`, and `/MD` or `/MDd`.
- You can control analysis cost without sacrificing completeness by using small, representative data sets. Data sets with runs in the seconds time range are ideal. You can always create additional data sets to ensure all your code is inspected.

Step 2. Collect Result.

Configure, then run an analysis:

- The Inspector offers preset threading analysis configurations to help you control analysis scope and cost. Widening analysis scope maximizes the load on the system, the time required to perform the analysis, and the chances the analysis will fail because the system may run out of resources.
- You can run threading error analyses from the **Tools** menu or toolbar. You can also use the `insp-cl` command.

Step 3. Manage Result.

Choose a problem set, interpret the result data, then resolve the issue:

- An observation is a fact the Inspector observes at a source code location. A problem is a small group of closely related observations (source code locations) that indicate an error in the target. A problem set is a larger group of more loosely related observations (source code locations) that could share a common solution.
- Think of the **Problems** pane on the **Overview** window as a *to-do* list: Start at the top and work your way down.
- Double-click an observation or problem set on the **Overview** window to navigate to the **Sources** window. Click the **Overview** button on the **Sources** window to return to the **Overview** window.

- Right-click an observation or problem set to display a pop-up menu, then choose **Explain Problem** to access more information on interpreting and resolving the problem.
- Double-click an observation on the **Sources** window to open the Visual Studio* editor.

More Resources

3

Getting Help

Browsing Help in the Microsoft Document Explorer

You can browse and search for topics in different ways:

- Use **Help** > **Contents** to open the Contents window and browse the Table of Contents.
- To view help for an installed parallel studio tool, select **Help > Intel Parallel Studio 2011 > Parallel Studio Help** and select the tool icon to open the title page of that tool.
- Use **Help** > **Index** to open the Index window and access an index to topics. Either type in the keyword you are looking for, or scroll through the list of keywords.
- Use **Help** > **Search** to open the Search page and search the full text of topics in the help.

Locating Intel Topics

To filter the documentation so that only the Intel documentation appears, select **Help** > **Contents** from the Visual Studio* user interface. In the **Filtered by:** drop-down list, select **Intel**.

To determine where the currently displayed topic appears in the table of contents (TOC), click the  **Sync with Table of Contents** button on the Visual Studio toolbar to highlight the topic in the Contents pane.

Where applicable, the Parallel Studio help topics provide a  **Where am I in the workflow?** button. Click the button to view the workflow with a highlight on the stage that this topic discusses.

Activating Intel Search Filters

With Microsoft Visual Studio 2005 and 2008, you can include Intel documentation in all search results by checking the **Intel** search filter box for the **Language**, **Technology**, and **Content Type** categories. You must check the **Intel** search box for all three categories to include Intel documentation in your searches. Unchecking all three **Intel** search boxes excludes Intel documentation from search results. The Intel search filters work in combination with other search options for each category.

Using Context-Sensitive Help

Context-sensitive help enables easy access to help topics on active GUI elements. The following context-sensitive help features are available on a product-specific basis:

- **? Help**: Click the ? button, in the upper-right corner of the dialog box or pane to get help for the dialog box or pane.
- **F1 Help**: Press F1 to get help for an active dialog box, property page, pane, window, or for the currently selected function in the code editor.
- **Dynamic Help**: Select **Help > Dynamic Help** to open the Dynamic Help window, which displays links to relevant help topics for the current window, or for the currently selected functions in the code editor.

Product Website and Support

The following links provide information and support on Intel software products, including Intel® Parallel Studio:

- http://www.intel.com/software/products/

  At this site, you will find comprehensive product information, including:

  - Links to each product, where you will find technical information such as white papers and articles
  - Links to user forums
  - Links to news and events

- http://software.intel.com/en-us/articles/intel-parallel-studio/

  Intel® Software Network, Parallel Studio Support page, with links to support forums, startup help, knowledge base, and getting started video.

- http://software.intel.com/en-us/articles/tools/

  Intel® Software Development Products Knowledge Base.

- http://www.intel.com/software/products/support/

  Technical support information, to register your product, or to contact Intel.

For additional support information, see the Technical Support section of your Release Notes.

System Requirements

For detailed information on system requirements, see the Release Notes.