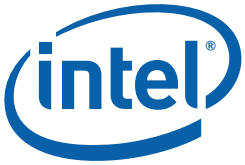


Intel® Parallel Inspector Sample Code Guide

This *Intel® Parallel Inspector Sample Code Guide*:

- Shows you how to resolve five common threading and memory issues in the Visual Studio* 2005 or 2008 development environment: data race, deadlock, memory leak, invalid memory access, and mismatched allocation/deallocation issues.
- Builds upon the foundation in the *Getting Started with Intel® Parallel Inspector* guide, which shows you how to use basic Intel® Parallel Inspector features to identify and analyze a threading and memory issue.

After reading both guides, you should be ready to use the Intel® Parallel Inspector to identify, analyze, and resolve threading and memory issues in your own multithreaded applications.

Contents

1 Overview

Guide Objective

Help you identify and analyze typical threading and memory errors that prevent five sample applications from correctly displaying an `abcde` banner.

Sample Banner Application

Tools

Use the following tools to achieve the guide objective:

- Visual Studio* 2005 or 2008 development environment
- The Intel® Parallel Inspector
- Intel® Parallel Inspector Help

  To display Help, from the Visual Studio* Help menu, choose **Intel Parallel Studio > Parallel Studio Help > Inspector Help**. While using Intel® Parallel Inspector in the Visual Studio* environment, you can also interactively display relevant Help by pressing the F1 key, clicking a **?** button, or clicking a link in the **Dynamic Help** window.
- Sample applications shipped with the Intel® Parallel Inspector

NOTE: All code examples are designed only to illustrate Intel(R) Parallel Inspector features. They do not represent best practices for creating multithreaded code. Results may vary depending on the nature of the analysis.

Preparation

1. Copy the samples `.zip` file from the `samples\`*`locale`* folder in the Intel® Parallel Inspector installation folder (the default installation folder is `C:\Program Files\Intel\Parallel Studio\Inspector`) to a writable directory or share

on your system, such as a `My Documents\Visual Studio 200x\Intel\samples` folder.

2. Extract the sample files from the `.zip` file.

2 Identify, Analyze, and Resolve Data Race Problems

A data race occurs when:

- Multiple threads write to the same memory location without proper synchronization (Write -> Write).
- One thread reads while a different thread concurrently writes to the same memory location without proper synchronization (Read -> Write).
- One thread writes to while a different thread concurrently reads the same memory location (Write -> Read).

Data races make applications nondeterministic: different runs with the same inputs can yield different, and therefore potentially unexpected or incorrect, results. And results that mysteriously appear and disappear make your debugging job difficult.

The Intel® Parallel Inspector:

- Spotlights application nondeterministic propensities.
- Highlights variable accesses that, when interleaved differently because threads are scheduled differently in different runs, can produce different results.

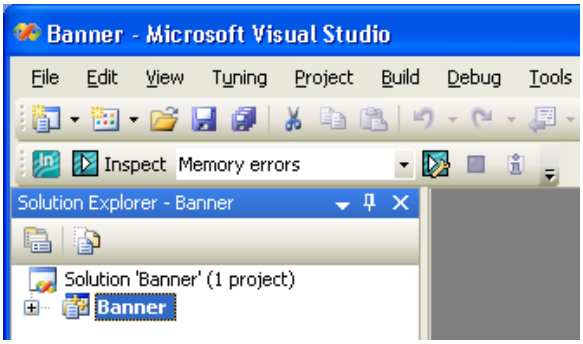
Choose and Build Target

NOTE:	This guide presents screen shots from the Visual Studio* 2005 development environment.

NOTE:	The sample applications are non-deterministic. Your screens may vary from the screens shown throughout this guide.

See **Choosing Targets > About Choosing Targets** in Inspector Help for information about compiler/linker options that produce the most accurate result data.

1. From the Visual Studio* menu, choose **File > Open > Project/Solution**.
2. In the **Open Project** dialog box, open the `banner\threadingDataRace\Banner.sln` file to display the solution/project in the **Solution Explorer**.
3. Ensure the project is highlighted in the **Solution Explorer**.
4. From the Visual Studio* menu, choose **Debug > Start Without Debugging** to compile, link, and test the project.
5. Check **Output** to ensure the project compiled successfully.
6. Check for a console window with a banner display. This ensures the project runs on your system without the Intel® Parallel Inspector. Press the Enter key twice to close the console window.




```
Banner - 0 error(s), 0 warning(s)
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

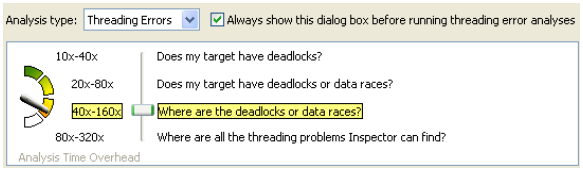
```
          b                         eee
aaa       b              ccc        e e
a a       bbb            c          eee
a a a     b b            c          e
aaaaa     bbb            ccc        eee
```

Collect Result Data

Configure and Run Analysis

See **Configuring Analyses > About Configuring Analyses** in Inspector Help for information about preset configuration levels.

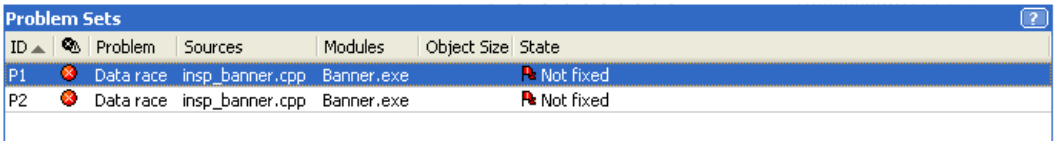
1. From the Inspector toolbar, choose **Threading errors** in the **Analysis type** drop-down list and click the **Inspect** button to display the **Configure Analysis** dialog box.
2. Ensure **Threading Errors** appears in the **Analysis type** drop-down list.
3. Drag the result scope slider to the **Where are the deadlocks or data races?** preset level.



4. Click the **Run Analysis** button to execute the target and collect result data in a `My Inspector Results\resultdir\*.insp` file in the solution folder.
5. Notice the banner probably displays incorrectly (some variation of `abcde`).
6. Press the Enter key to end target execution and close the console window.
7. On the **Event Log** window, click the **Interpret Result** button to start managing result data.

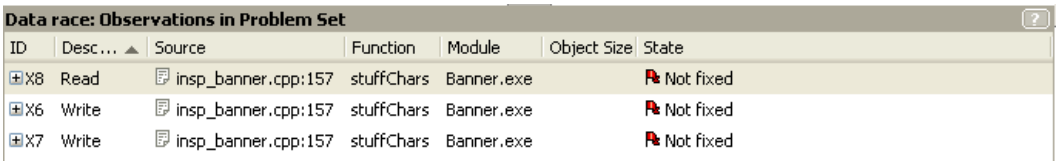
Manage Result Data

Choose Problem Set



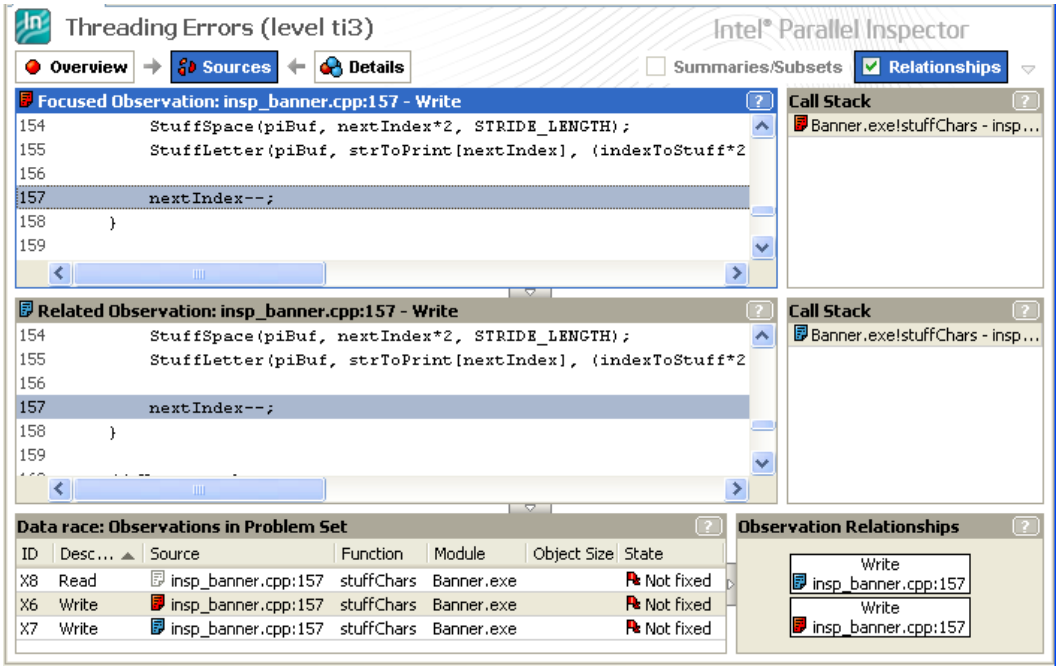
The Intel® Parallel Inspector selects the **P1** problem set for you by default.

Choose Focus Observation



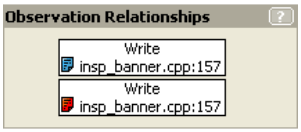
Double-click the **X6** observation.

Interpret Result Data



Right-click the diagram to display a pop-up menu, then choose **Explain Problem** to see *Inspector Help* information for this problem type.

Look at the diagram in **Observation Relationships** pane. It shows the relationship between the current focus observation (**X6**) and a related observation (**X7**) in a problem in the problem set.



This diagram clearly identifies the issue: Two threads are trying to write to the same memory location without proper synchronization.

Resolve Issue

Possible correction strategies include:

1. Find the smallest critical section encompassing a portion of code where you must ensure a read variable does not change until the reading thread finishes accessing that variable. For instance, in the example, one thread increments the variable `nextIndex`.
2. *Wrap* the critical region with a programming primitive that ensures mutual exclusion between one thread instance of that code portion and another instance.

To learn more about these correction strategies:

- See the comments embedded in the **BannerFixed** code.

- See *About Mutex Objects* in the MSDN Library at http://msdn.microsoft.com/en-us/library/ms686927(VS.85).aspx.

Under standard debugging circumstances, you would now do the following:

1. Double-click the source code you want to edit.

   Double-clicking opens the source file in a new tab where you can edit it with the Visual Studio* code editor.

2. When you're finished editing source code for this problem, return to the result data tab and click another observation in the **Observations in Problem Set** pane to explore other problems in the problem set. Edit source code as necessary.
3. When you're finished editing source code, click the **Overview** button on the result data tab to return to the Overview window.
4. Manage the remaining problem sets using the same techniques you used to manage the first problem set.
5. When you're finished managing problem sets, rebuild the target and rerun the analysis: from the Visual Studio* menu, choose **Tools > Intel Parallel Inspector > Re-inspect**.

NOTE: Result type and scope are persistent by target until you reconfigure.

If you handled all threading issues appropriately, the Intel® Parallel Inspector displays no problems sets and displays the banner correctly.

Instead, examine an Intel® Parallel Inspector result where there are no errors.

1. Close the **Banner** solution/project.
2. Open the **BannerFixed** solution/project in the `banner\threadingDataRace\BannerFixed.sln` file.
3. Build the **BannerFixed** target.
4. Configure and run a threading analysis at the **Where are the deadlocks or data races?** preset level.
5. Notice:
   - The banner displays properly.
   - When you press any key to end target execution and close the console window, and click the **Interpret Result** button on the **Event Log** window, the **Overview** window shows no problem sets.
6. If desired, use a diffing tool of your choice to compare the `banner\threadingDataRace\insp_banner.cpp` and `banner\threadingDataRace\insp_bannerfixed.cpp` files.

3 Identify, Analyze, and Resolve Deadlock Problems

A deadlock occurs when two or more threads are waiting for the other to release resources (mutex, critical section, thread handle, etc.), but none of the threads releases the resources. Thus no thread can proceed.

For example, two threads each hold a resource the other thread wants, and neither will give up its resource until it gets the other resource.

Choose and Build Target

1. Open the `banner\threadingDeadlock\Banner.sln` file.
2. Compile and link the project.

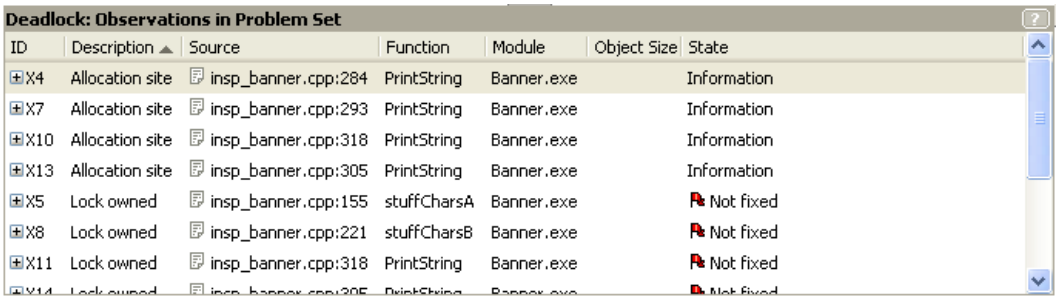
Collect Result Data

Configure and Run Analysis

1. Configure and run a threading error analysis at the **Where are the deadlocks or data races?** preset level.
2. In the console window, notice the banner does not display and the sample application hangs.
3. When the **Event Log** window displays an **Error: Deadlock** event, click the **Close** button on the system console window.
4. In the **Event Log** window, click the **Interpret Result** button to display the **Overview** window.

Manage Result Data

Choose Focus Observation



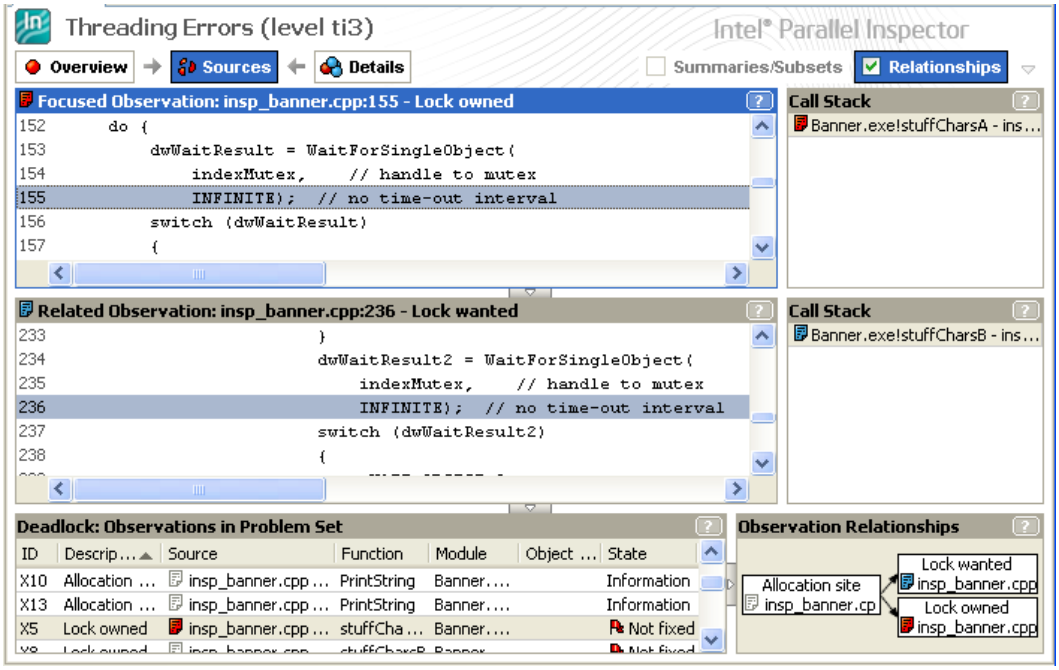
The Intel® Parallel Inspector selects the single problem set for you by default.

Notice the observations in the problem set:

- **Allocation site** – Represents the location and associated call stack where the resource was created.
- **Lock owned** – Represents the location and associated call stack of the thread holding the object requested by another thread.
- **Lock wanted** – Represents the location and associated call stack of the thread requesting the object held by another thread.

Double-click any **Lock owned** or **Lock wanted** observation.

Interpret Result Data



This is where the **Observation Relationships** diagram is particularly helpful:

- Each box in a diagram represents an observation in a problem.
- Vertically stacked boxes indicate concurrent observations.
- Boxes arranged left-to-right with connecting arrows indicate a time ordering.

Look at the source code associated with the **Lock wanted** and **Lock owned** observations in the diagram. The code shows where a thread has a resource (the mutex object in the **Lock owned** observation) that another thread wants (the same mutex object in the **Lock wanted** observation).

Resolve Issue

For the general case, when a thread wants a resource another thread owns, make the owning thread give up the resource. This can be as simple as recognizing when the owning thread is done with the resource and then giving it back to the system.

The general case doesn't work when you have circularity: each thread is an owning and a wanting thread. In this case (as with this example), you must break the circularity.

- If the resources to be acquired can be acquired in any order, make sure all threads acquire the resources in the same order. Thus, when a thread acquires the first desired resource, it:
  1. Prevents any other threads from acquiring that first resource.
  2. Is able to acquire the remaining resources.
  3. Finishes using the resources.
  4. Releases the resources for other threads to acquire.
- If the resources to be acquired can be pre-empted (that is, taken back even if owned), make sure you stipulate this during the resource creation. For example, in the example given here, do not specify INFINITE hold time.

You can now do one or more of the following:

- Resolve the issue just as you would under standard debugging circumstances.
- Analyze the `banner\threadingDeadlock\BannerFixed.sln` file.
- Use a diffing tool of your choice to compare the `banner\threadingDeadlock\insp_banner.cpp` and `banner\threadingDeadlock\insp_bannerfixed.cpp` files.

4 Identify, Analyze, and Resolve Memory Leak Problems

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

Choose and Build Target

1. Open the `banner\memoryLeak\Banner.sln` file.
2. Compile and link the project.

Collect Result Data

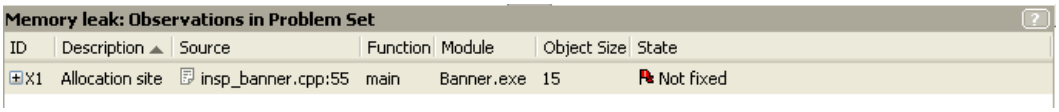
Configure and Run Analysis

1. Configure and run a memory error analysis at the **Does my target leak memory?** (default) preset level.
2. Notice the banner probably displays correctly. Memory-related problems are often sleeper issues not apparent at this level of execution.

3. After you press any key to end target execution and close the console window, click the **Interpret Result** button on the **Event Log** window to display the **Overview** window.

Manage Result Data

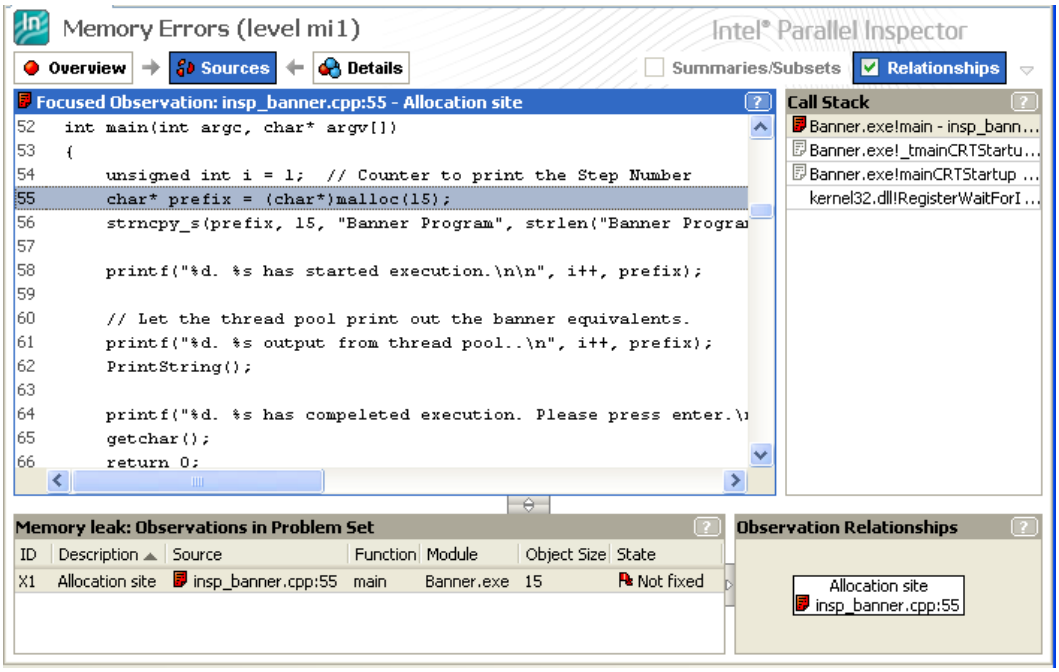
Choose Focus Observation



The Intel® Parallel Inspector selects the single problem set for you by default.

Double-click the **X1** observation in the problem set.

Interpret Result Data



The issue: there is an allocation source location but no deallocation source location.

Resolve Issue

To resolve the memory leak problem (while avoiding a related mismatched allocation/deallocation problem illustrated later in this document): 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 application 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.

You can now do one or more of the following:

- Resolve the issue just as you would under standard debugging circumstances.
- Analyze the `banner\memoryLeak\BannerFixed.sln` file.
- Use a diffing tool of your choice to compare the `banner\memoryLeak\insp_banner.cpp` and `banner\memoryLeak\insp_bannerfixed.cpp` files.

NOTE: The only difference between the sources for the broken and fixed versions is the presence of the `free()` call.

5 Identify, Analyze, and Resolve Invalid Memory Access Problems

An invalid memory access occurs when a read or write instruction references memory that is logically or physically invalid.

For example, an application tries to read from or write to memory not allocated to the application, or allocated to the application via a memory allocator, but the application goes outside the bounds of usable space into the allocator's internal space.

Choose and Build Target

1. Open the `banner\memoryInvalidWrite\Banner.sln` file.
2. Compile and link the project.

Collect Result Data

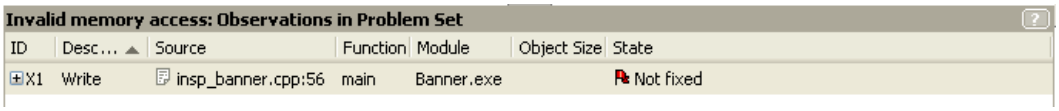
Configure and Run Analysis

1. Configure and run a memory error analysis at the **Does my target have memory access problems?** preset level.
2. Notice the banner probably displays correctly. Memory-related problems are often sleeper issues not apparent at this level of execution.
3. After you press any key to end target execution, the system displays a debug error message.
4. Click the **Abort** button.
5. Close the console window and click the **Interpret Result** button on the **Event Log** window to display the **Overview** window.

NOTE: The **Does my target have memory access problems?** preset level is sufficient for most write errors, such as in this example. For other more obscure errors of the same type, consider using the **Where are the memory access problems?** preset level.

Manage Result Data

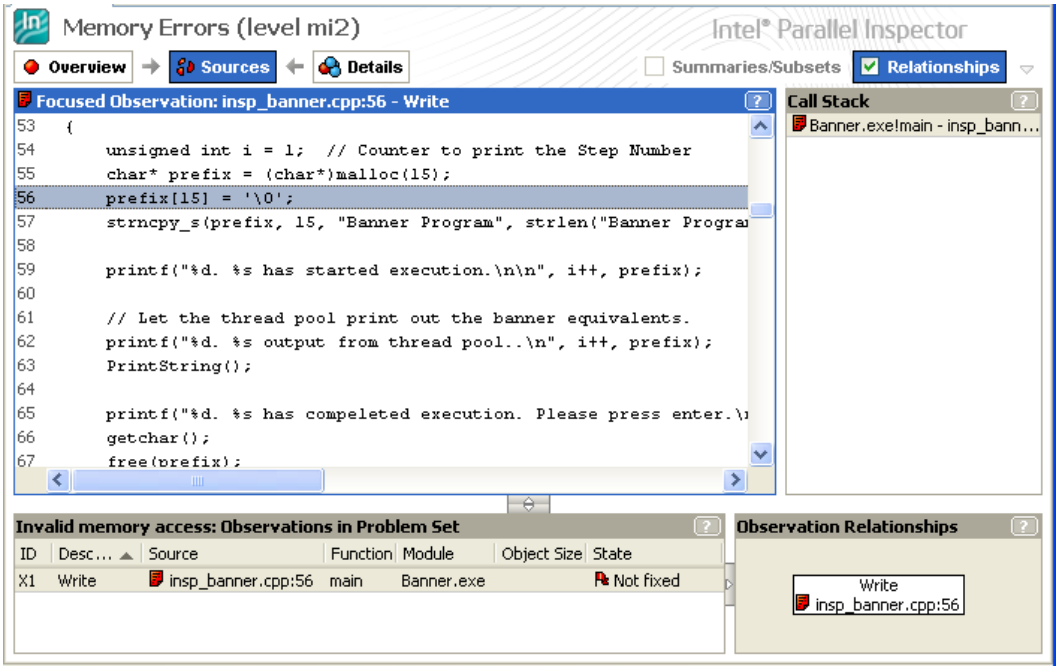
Choose Focus Observation



The Intel® Parallel Inspector selects the single problem set for you by default.

Double-click the **X1** observation.

Interpret Result Data



In this particular case, the issue is obvious; however, this is not normally the case with this type of error.

Here we are writing into index location 15 of `prefix` (the 16th memory location) within an aggregate data structure with only 15 locations. We *walked past* the end of the memory allocated for this data item. The reason this problem is clear is the allocation of space for this data item immediately precedes the invalid write. The problem source location may not be as close to the problem symptom location in your application.

Resolve Issue

Correction strategies include:

- If the item written to is an array, make sure the array is declared with enough space. (Remember that indices are zero-based.)
- If the item written to is a pointer, make sure the item points to memory legitimately available to the application for writing at the time of the write. Do not point to space on the stack from a called function that is no longer active, or memory that has already been `free`-ed, etc.

You can now do one or more of the following:

- Resolve the issue just as you would under standard debugging circumstances.

- Analyze the `banner\memoryInvalidWrite\BannerFixed.sln` file.
- Use a diffing tool of your choice to compare the `banner\memoryInvalidWrite\insp_banner.cpp` and `banner\memoryInvalidWrite\insp_bannerfixed.cpp` files.

NOTE: The only difference between the sources for the broken and fixed versions is the replacement of the invalid index (15) with a valid index (14).

6 Identify, Analyze, and Resolve Mismatched Allocation/Deallocation Problems

A mismatched allocation/deallocation occurs when a deallocation is attempted with a function that is not the logical reflection of the allocator used.

In the C++ programming language, the following are matched reflections:

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

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

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.

Choose and Build Target

1. Open the `banner\memoryMismatch\Banner.sln` file.
2. Compile and link the project.

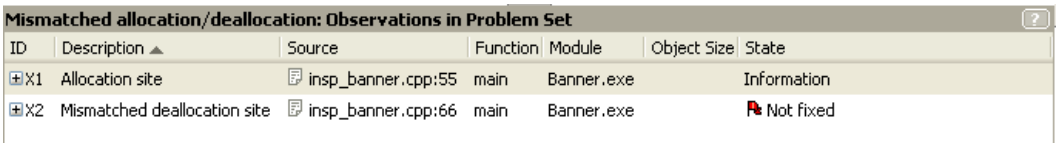
Collect Result Data

Configure and Run Analysis

1. Configure and run a memory error analysis at the **Does my target have memory access problems?** preset level.
2. Notice the banner probably displays correctly. Memory-related problems are often sleeper issues not apparent at this level of execution.
3. When you press any key to end target execution and close the console window, click the **Interpret Result** button on the **Event Log** window to display the **Overview** window.

Manage Result Data

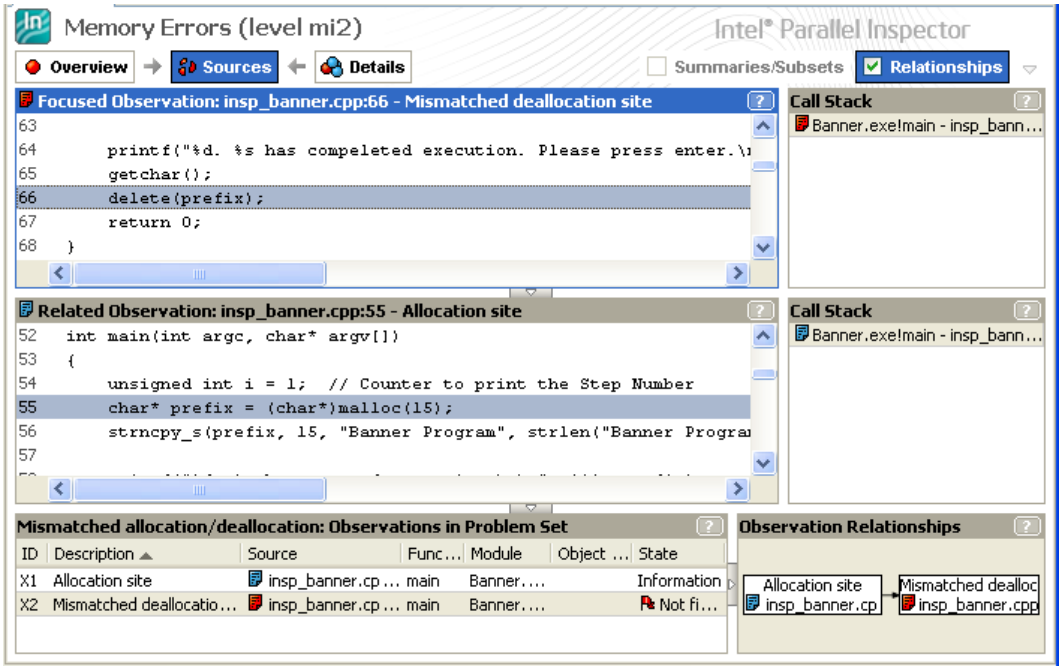
Choose Focus Observation



The Intel® Parallel Inspector selects the single problem set for you by default.

Double-click the **X2** observation.

Interpret Result Data



The issue: `delete` is not the appropriate deallocation mechanism for a `malloc()`.

Resolve Issue

Allocation/Deallocation mismatch problems are easy to resolve:

- Match `malloc()` with `free()`, new with `delete`, and `new[]` with `delete[]`. Never mix and match.
- Don't overlook the presence/absence of the `[]` when matching `new[]` with `delete[]`. Make both either scalar (`new/delete`) or aggregated (`new[]/delete[]`).

You can now do one or more of the following:

- Resolve the issue just as you would under standard debugging circumstances.
- Analyze the `banner\memoryMismatch\BannerFixed.sln` file.
- Use a diffing tool of your choice to compare the `banner\memoryMismatch\insp_banner.cpp` and `banner\memoryMismatch\insp_bannerfixed.cpp` files.

NOTE: The only difference between the sources for the broken and fixed versions is the replacement of the `delete` call with a `free()` call.

7 User and Reference Documentation

This guide focuses on using basic Intel® Parallel Inspector features to identify and analyze common parallelism and memory issues. To explore more features and get the most out of the Intel® Parallel Inspector, check the following resources:

Resource	Notes
Intel® Parallel Inspector Documentation	Use this HTML page to locate other Intel® Parallel Inspector resources. To open this HTML page, from the Windows* Start menu, choose **Intel Parallel Studio > Parallel Studio Documentation > Inspector Documentation**.
Intel® Parallel Studio resources	Intel Parallel Studio provides the most comprehensive set of tools for parallelism including Intel® Parallel Amplifier, Intel® Parallel Composer, and Intel® Parallel Inspector. Refer to the *Getting Started with the Parallel Studio* for an overview of all the components in the Parallel Studio. From the Windows* Start menu, choose **Intel Parallel Studio > Getting Started >Parallel Studio Getting Started Guide**. To open documentation that points to more resources for each installed Intel Parallel Studio tool, from the Windows* Start menu, choose **Intel Parallel Studio > Parallel Studio Documentation > Inspector Documentation**. You can find additional information on the Intel® Parallel Studio at http://www.intel.com/software/products

Legal Information