IBM Rational PurifyPlus for Windows
IBM Rational Purify for Windows

Getting Started Guide

Contents

Welcome to PurifyPlus

IBM Rational PurifyPlus: What it is

IBM® Rational® PurifyPlus™ consists of three dynamic software analysis components that help you you develop high-quality applications more efficiently:

- **Purify**® An automatic error detection tool for finding memory errors and leaks in every component of your program.

- **Quantify**® A performance analysis tool for resolving performance bottlenecks so your program can run faster.

- **PureCoverage**® A code coverage tool for making sure your code is thoroughly tested before you release it.

These tools are easy to use, yet provide invaluable information to help your team develop faster and more reliable applications in C/C++, Visual Basic, Java, or managed code in any language that Microsoft Visual Studio supports.

If you're developing code in Visual Studio, start the PurifyPlus tools from the Visual Studio menu. You can use Purify, for example, along with your Visual Studio debugger and editor to save time correcting a software defect. You can also use the tools as standalone applications when you don't need all the resources of Visual Studio.

If you're testing software, incorporate PurifyPlus into existing test scripts and harnesses to automate error detection, code-coverage monitoring, and performance testing. Use the tools from the beginning with your nightly tests so that you can easily spot regressions as soon as they occur.

Don't waste days looking for problems that PurifyPlus can pinpoint in seconds. And don't release a product with hidden bugs that these tools can detect easily. Consistent use of PurifyPlus, from the time you start development until you ship, will provide solid benefits both to you and to your customers.

Tips for development engineers

Here are some tips for using PurifyPlus to develop fast, reliable code.

Find memory errors early

Use Purify as you code to pinpoint hard-to-find bugs. Memory errors don't always show up right away, but they're the ones that will make your program crash someday.

Prevent performance bottlenecks

Whenever you write new code or modify existing code, use Quantify to catch any incremental performance losses before they turn into bottlenecks.

Quantify gives you the information you need to write more efficient code. It can turn everyone on your team into a performance engineer.

Purify

PureCoverage

Quantify
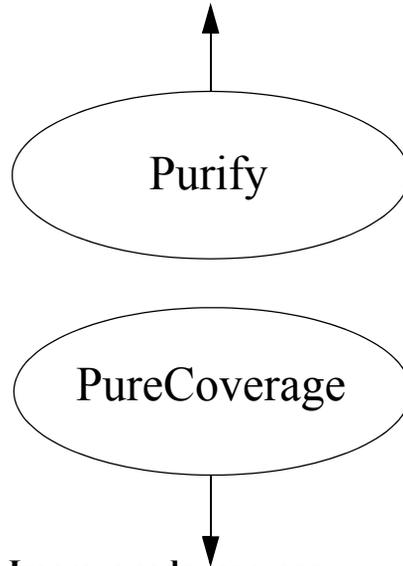
Improve code coverage

You haven't Purify'd® code you haven't run.

Use PureCoverage to make sure you're exercising all your code during pre-checkin testing.

For C/C++ code, you can run PureCoverage from within Purify—just click **Coverage, error, and leak data** in Purify's Run Program dialog.
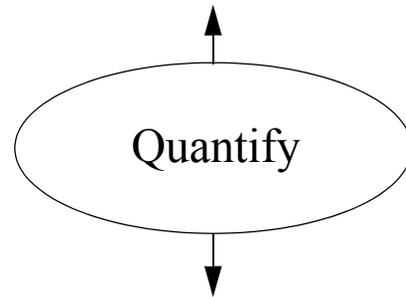
Analyze code structure

A common reason for writing new code is to improve the performance of a program. But how can you effectively improve the performance of code that might have been developed over several years by many different people?

Use Quantify not only to find performance bottlenecks, but also to learn more about how your code is structured. It will help you to make effective performance improvements.

Tips for test engineers

Here are some tips for using PurifyPlus to guarantee quality software.

Find the internal errors in your code

For best results, run all your tests on a Purify'd version of your program. This will find the internal memory problems that your external functionality tests can't uncover.

Test all your code daily

Use PureCoverage every day to make sure you're testing all your code. With ongoing coverage feedback, you can be sure your tests are keeping pace with your code development.

Purify

PureCoverage

Quantify

If coverage goes down . . .

If code coverage drops, your existing tests may not be exercising new code. Or the new code might have introduced a defect that's causing a large section of code not to be tested. Use an automated testing tool like Rational Robot or Rational Visual Test® to write test cases that exercise the new code.

If performance improves . . .

An unexpected improvement in performance can indicate that a large part of your code is no longer being exercised. Compare the most recent PureCoverage results with a previous run to see if you're still getting the same level of coverage.

If performance drops . . .

A sudden drop in performance is probably caused by the most recent code checked in. Let Quantify show you which parts of your program became slower compared to a previous run that had acceptable performance.

Other PurifyPlus resources

Additional information is available for all the PurifyPlus tools:

To learn how to pinpoint hard-to-find bugs in C/C++ native and .NET managed code, go to
Getting Started: Purify on page 7
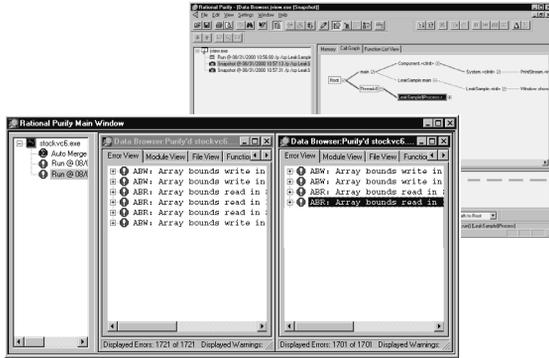
To learn how you can avoid shipping untested code, go to
Getting Started: PureCoverage on page 41
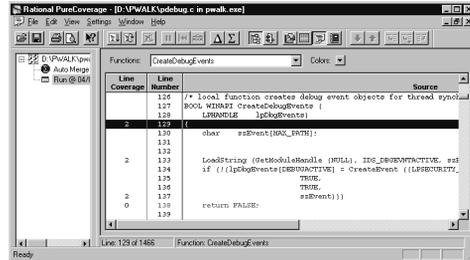
To learn how to highlight performance bottlenecks, go to
Getting Started: Quantify on page 57
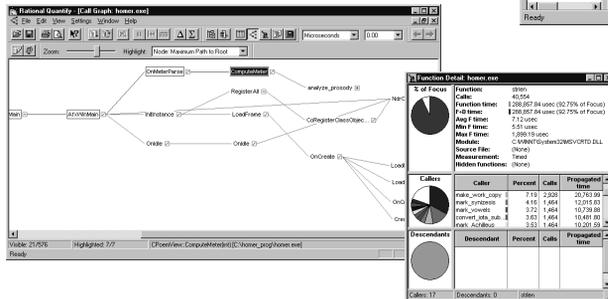
The online help systems for Purify, Quantify, and PureCoverage contain detailed information about using the products and interpreting the data they collect.

For information about other IBM Rational products, go to
http://www.ibm.com/software/rational.

Contacting IBM customer support

The IBM software support Internet site provides you with self-help resources and electronic problem submission. The IBM Software Support Home page for Rational products can be found at www.ibm.com/software/rational/support/.

Voice Support is available to all current contract holders by dialing a telephone number in your country (where available). For specific country phone numbers, go to www.ibm.com/planetwide/.

Note: When contacting IBM Software Support, please be prepared to supply the following information:

- Your name, company name, ICN number, telephone number, and e-mail address

- Your operating system, version number, and any service packs or patches you have applied

- Product name and release number

- Your PMR number (if you are following up on a previously reported problem)

Getting Started: Purify

Whether you're working in C/C++ native code or .NET managed code, Rational® Purify® can save you time and improve the quality of your code.

Purify for C/C++ developers and testers

What it does

Run-time memory errors and leaks are among the most difficult errors to locate and the most important to correct. The symptoms of incorrect memory use are unpredictable and typically appear far from the cause of the error. The errors often remain undetected until triggered by some random event, so that a program can seem to work correctly when in fact it's only working by accident.

That's where Purify can help you get ahead. Purify provides:

- Fast, comprehensive run-time error detection for C/C++ programs
- Error checking even when the source is not available
- Code-coverage data that shows you code you haven't tested

Purify automatically integrates into Microsoft Visual Studio and requires no special builds. You can use Purify without changing the way you work.

Find errors before they occur

Purify detects the following kinds of memory errors—and many others—before they actually occur, so that you can resolve them before they do any damage:

- Array bounds errors
- Accesses through dangling pointers
- Uninitialized memory reads
- Memory allocation errors
- Memory leaks

More information? For a complete list of the errors that Purify detects in C/C++, select **Purify Messages** from the Purify Help menu.

Check every component in your program

Software development today is component based. To deliver quality applications on time, you not only need to make sure your own code is error free, you also need a way to check the components your software uses—even when you don't have the source code. Errors that occur within a component may be the result of your code supplying the component with unexpected data; only Purify can detect such errors so that you can correct your use of the component and improve the reliability of your application.

Purify can check every component in your program, even in complex multi-threaded, multi-process applications, including:

- .dll's, including Windows .dll's and Microsoft Foundation Class Library .dll's

- C/C++ components embedded within Visual Basic applications, Internet Explorer, Firefox, or any Microsoft Office application

- Microsoft Excel and Microsoft Word plug-ins

- COM-enabled applications using OLE and ActiveX controls

Purify checks calls to Windows API functions, including GDI, Internet services, system registry, and COM and OLE interface API functions. It also validates parameters such as memory handles and pointers.

Look for errors in the right places

In addition to finding the critical errors that occur when you exercise your program, Purify can also tell you how thoroughly you've covered your program's code. If you have Rational PureCoverage installed, Purify can collect coverage data automatically for every run, report exactly how much of your code you've checked, and identify untested lines and functions. Using this information you can make sure you're finding the errors in all your code, and that you won't be caught off-guard by undiscovered problems in lines or functions that you overlooked.

More information? Look up *coverage data* in the Purify online Help index.

Use Purify from the start

For maximum benefit, start using Purify as soon as your code is ready to run and continue using it regularly throughout your development cycle, especially for:

- **Code check-in.** Reduce the risk that bugs in your code might impact other code modules.

- **Nightly tests.** Incorporate Purify into your test harness to verify that modules work together and to expose code dependencies and collisions. Collect coverage data for every run to make sure that your tests are exercising any code that has been added or modified.

- **Acceptance tests.** Validate third-party code or code from other development groups before incorporating it into your application.

By using Purify consistently from the time you start development, you'll release clean, reliable products on time.

The basic steps

With Purify, you can deliver more reliable C/C++ code in a few easy steps:

1 Run your program with Purify to collect:

- Error data

- Code coverage data

2 Analyze the error data and correct your source code.

3 If you've collected coverage data, analyze it to find any parts of your code that you have not Purify'd®.

4 Rerun your program with Purify.

The following pages show you how to use Purify integrated with Microsoft Visual Studio, but you can also use Purify in other ways. Read the following:

- *Using Purify standalone* on page 22

- *Testing C/C++ code with the command-line interface* on page 23.

Running a C/C++ program with Purify

Open your project in Visual Studio, then engage Purify from the Purify menu.

If you have installed Rational PureCoverage, select the menu item **PurifyPlus > Purify > Collect Coverage Data** to collect coverage data in addition to checking for errors and memory leaks.

Build and execute your program using commands from the Visual Studio **Build** menu. To get the maximum level of detail in Purify error and coverage data, build your program with debug and relocation data.

More information? For information about building programs with debug and relocation data, look up *debug data* in the Purify online Help index.

Purify copies the program and each library the program calls, then *instruments* the copies using Object Code Insertion (OCI) technology. The instrumentation process inserts instructions that validate every read, write, and memory allocation and deallocation. If you're collecting coverage data, Purify also inserts instructions that increment counters when you exercise specific lines and functions.

Purify reports its progress as it instruments each module.

The module that Purify is currently instrumenting ⎯⎯

The instrumentation level for error checking and coverage monitoring for each module ⎯⎯

Purify Instrumenting stockvc6.exe ...				
➡ stockvc6.exe	344150	Precise	Line	
RPCLTC1.DLL	13584	Minimal	Exclude	
RPCLTS1.DLL	8976	Minimal	Exclude	
KERNEL32.DLL	372496	Minimal	Exclude	
USER32.DLL	330512	Minimal	Exclude	
GDI32.DLL	165648	Minimal	Exclude	
COMDLG32.DLL	185104	Minimal	Exclude	
WINSPOOL.DRV	92944	Minimal	Exclude	
ADVAPI32.DLL	246544	Minimal	Exclude	

Cancel Instrumentation   Help

Purify instruments each module at a default instrumentation level. If you want to focus on a specific part of your program, you can override the default and customize the instrumentation level.

More information? For an explanation of instrumentation levels and how to use them, read *Customizing instrumentation* on page 20. For more detail, look up *instrumenting* in the Purify online Help index.

Purify caches the instrumented copy of each module. When you rerun a program, Purify saves time and resources by using the cached modules, re-instrumenting only the ones that have changed since the previous run.

As you exercise your program, Purify detects run-time errors and memory leaks and displays them in an Error View tab in the Purify Data Browser window.

Purify Error View tab,
Data Browser window

stockvc6 - Microsoft Visual C++ - [Rational Purify Main Window]

File Edit View Insert Project Build Tools Purify PureCoverage Quantify Window Help

Data Browser:Purify'd stockvc6.exe

Error View | Module View | File View | Function List View

Starting Purify'd stockvc6.exe at 08/05/99 16:22:52
Starting main
ABW: Array bounds write in CStockApp::CStockApp(void
UMR: Uninitialized memory read in SetWindowTextA (14
UMR: Uninitialized memory read in strcmp {17 occurre
UMR: Uninitialized memory read in lstrlenA {14 occur
UMR: Uninitialized memory read in MultiByteToWideCha
UMR: Uninitialized memory read in MultiByteToWideCha
ABW: Array bounds write in sprintf {47 occurrences}
ABR: Array bounds read in SetWindowTextA {646 occurr
ABR: Array bounds read in lstrlenA {830 occurrences}
ABR: Array bounds read in SetWindowTextA {184 occurr
ABW: Array bounds write in sprintf {13 occurrences}

Displayed Errors: 1721 of 1721   Displayed Warnings: 53 of 53   Bytes leaked: 0+0

Workspac
Stock
Sc

stockvc6.exe
Auto Merg
Run @ 08

Purify: Instrumenting D:\Program Files\Rational\Purify\samples\StockNT\Release\stockvc6.e
Purify: Instrumentation finished

Build Debug Find in Files 1 Find in Files 2 Results SQL Debugging

Ready

More information? Look up *error view* in the Purify online Help index.

Note: If you're debugging client/server and multi-process applications, you can debug several processes and see the error reports for each running application simultaneously. To do this, run each process in a separate instance of Visual Studio with Purify engaged. Alternatively, you can use the standalone Purify user interface. See *Using Purify standalone* on page 22.

Seeing all your errors at a glance

Purify displays error and warning messages about run-time errors and memory leaks, and informational messages about the progress of your program's execution.

Color-coded icons show message severity:
● informational   **▲** warning   **●** error

Acronyms like ABW identify message type

For a description of a message, right-click the message, then select **Describe**

```
Data Browser:Purify'd stockvc6.exe                          _ □ ×

Error View | Module View | File View | Function List View |

⊞ ● Starting Purify'd stockvc6.exe at 08/05/99 16:22:52       ▲
   ● Starting main
⊞ ● ABW: Array bounds write in CStockApp: ┌──────────────────────────┐ ence}
⊞ ▲ UMR: Uninitialized memory read in Set│ Describe ABW: Array Bounds Write │ s}
⊞ ▲ UMR: Uninitialized memory read in str│                          │
⊞ ▲ UMR: Uninitialized memory read in lst│ Copy                     │
⊞ ▲ UMR: Uninitialized memory read in Mul│                          │ ences
⊞ ▲ UMR: Uninitialized memory read in Mul│ Submit ClearQuest Defect │ ences
⊞ ● ABW: Array bounds write in sprintf {4│                          │
⊞ ● ABR: Array bounds read in SetWindowTe│ Expand                   │
⊞ ● ABR: Array bounds read in lstrlenA {8 │ Expand Branch            │
⊞ ● ABR: Array bounds read in SetWindowTe│ Collapse                 │
⊞ ● ABW: Array bounds write in sprintf {1 │                          │
⊞ ● EXH: Handled exception in AfxThrowUse│ QuickFilter              │ ence}
⊞ ▲ HAN: Handle 0x00000001 is invalid in │ Create Filter...         │
⊞ ▲ HAN: Handle 0xffffffff is invalid in │                          │
   ● Summary of all memory leaks... {0 byt│ View Source File         │
⊞ ● Summary of all memory in use... {29034│ Select Source File       │
                                            └──────────────────────────┘  ▼
◄                                                                      ►

Displayed Errors: 1721 of 1721   Displayed Warnings: 53 of 53   Bytes leaked: 0+0
```

When you exit the program, Purify reports memory leaks. In addition to memory leaks, you can set Purify to report memory in use at exit and handles in use at exit.

More information? Look up *error and leak settings* in the Purify online Help index.

When identical errors repeat

An error often repeats many times in a program, particularly if it occurs inside a loop. To provide a concise overview of a program's errors, Purify by default displays each error message only once, the first time an error occurs, and then updates a counter whenever the error repeats.

This uninitialized memory
read (UMR) occurred 17 times
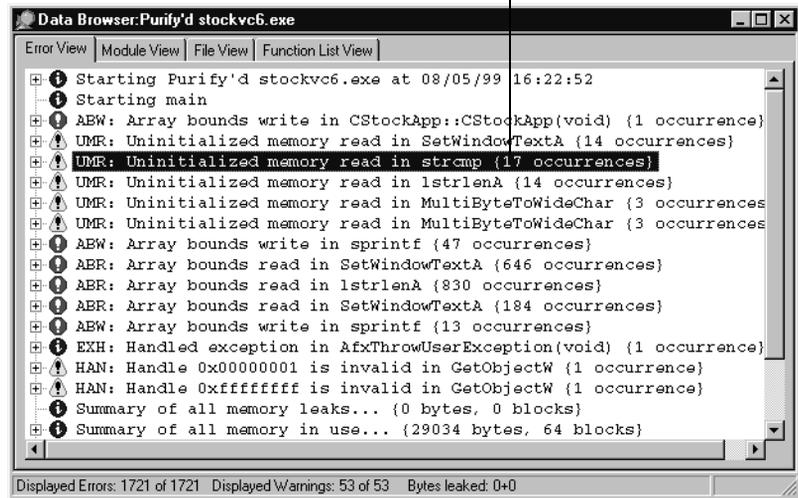
More information? If you want Purify to display each occurrence of a message individually, instead of reporting counts, you can change the default setting. Look up *error and leak settings* in the Purify online Help index.

Focusing on critical errors first

A large program can generate hundreds of messages. To focus on the most critical error messages quickly, create filters to hide all other messages from the display.

You can filter messages individually, or you can filter them based on their type and source. Consider hiding all informational messages, for example, or all messages originating from a specific file.

An *unfiltered* error view displays all the messages from the program

A *filtered* error view displays only the messages you want to see

Right-click a message and select **QuickFilter** to hide the message immediately

Or select **Create Filter** to define a set of filtering criteria
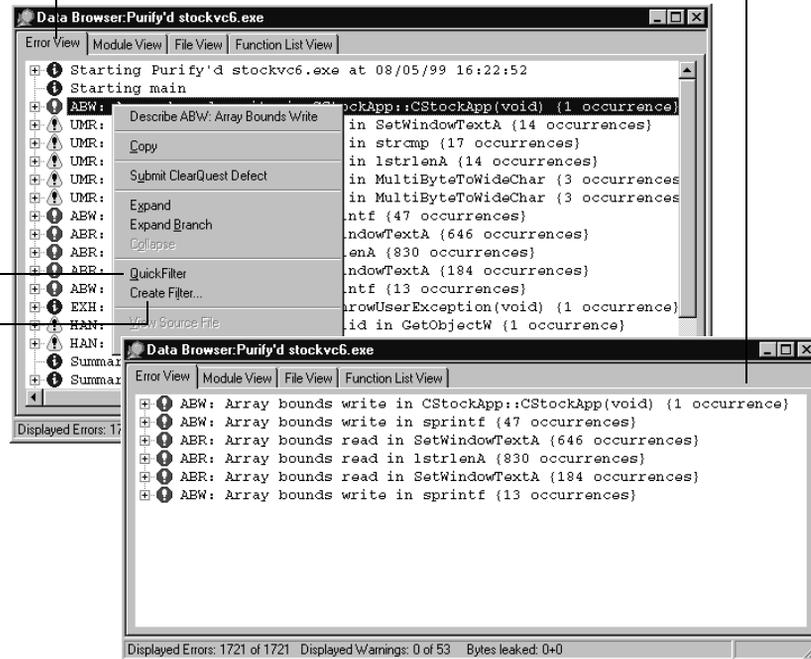
Once created, filters apply to the current run and to all future runs of the program until you disable them. Disabling a filter causes hidden messages to be redisplayed in the error view.

Working with filters

Purify filters are very flexible. Click the Filter Manager tool ▣ to create individual filters or groups of filters, and to apply them to specific programs or modules. You can also create global filters that apply to all programs and modules. And you can share filters, which Purify saves as .pft files, with other members of your team.

Click to enable or disable filters   The checked filters apply to the selected program until you disable or delete them

The Filter Manager creates a filter group for each program you run

Drag and drop filters to move or copy them

Name	Type	Enabled	Date	Comment
☑ ▣ HAN: Handle ...	Filter	Yes	08/06/99 10:28:08	QuickFilter
☑ ▣ HAN: Handle ...	Filter	Yes	08/06/99 10:28:08	QuickFilter
☑ ▢ My Filters	Group	Yes	08/06/99 10:37:19	Assigned to GB
☑ ▣ Starting Purify...	Filter	Yes	08/06/99 10:27:12	
☐ ▣ UMR: Uninitia...	Filter	No	08/06/99 10:55:02	

Purify Filter Manager: Error Data - stockvc6.exe

Filter Edit View Help

☐ ☑ 🔍 All Filters
  ☐ ☑ ■ stockvc6.exe
     ☑ 🗀 My Filters

OK   Cancel   Apply   Help

More information? Purify provides filters for coverage data as well as for error data. Look up *filtering data* in the Purify online Help index.

In addition to filtering, you can also use Purify's PowerCheck feature to focus on specific modules and simultaneously minimize instrumentation time. For information about the PowerCheck feature, read *Customizing instrumentation* on page 20.

Analyzing Purify error data

You can expand Purify's messages to pinpoint *where* errors occur and to obtain diagnostic information for understanding *why* they occur.

Here's an example of an expanded ABW (array bounds write) error message:

The location in memory where the error occurs —

Call stack showing the function calls leading to the error —

Flag indicating the line where the error occurs —

Call stack showing the function calls leading to the allocation of the memory block associated with the error —

```
Data Browser:Purify'd stockvc6.exe                                    _ □ ×
Error View │ Module View │ File View │ Function List View │
ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}
    Writing 1 byte to 0x001540f8 (1 byte at 0x001540f8 illegal)
    Address 0x001540f8 is 1 byte past the end of a 80 byte block at 0
    Address 0x001540f8 points to a HeapAlloc'd block in the default he
    Thread ID: 0xcb
    Error location
    CStockApp::CStockApp(void) [Stock.cpp:156]
        CStockApp::CStockApp()
        {
            m_Ticker = (char *)HeapAlloc(GetProcessHeap(), 0, TICKER_
    ➡       m_Ticker [TICKER_SIZE] = 0;
        }

        ////////////////////////////////////////////////////////////
    $E188 (C++ ctor/dtor)  [Stock.cpp:162]
    $E191 (C++ ctor/dtor)  [stockvc6.exe]
    _unlockexit       [stockvc6.exe]
    WinMainCRTStartup [memcpy.obj]
    Allocation location
        HeapAlloc       [KERNEL32.dll]
    CStockApp::CStockApp(void) [Stock.cpp:155]
Displayed Errors: 1721 of 1721   Displayed Warnings: 53 of 53    Bytes leaked: 0+0
```

The level of detail provided in call stacks depends on the availability of debug and relocation data. Even if you build your program in release mode, you can still get the highest possible level of detail. For more information, look up *debug data (C/C++), release version* in the Purify online Help index.

You can customize the format of Purify's messages. For example, you can increase the number of lines of source code that are displayed, or include instruction pointers and offsets to make locating errors easier.

More information? Look up *preferences, source code (C/C++)* in the Purify online Help index.

Correcting errors

Purify makes it easy to correct errors.

Double-click the line where the error occurs

Purify opens the source code in the editor, positioned at the exact location of the error
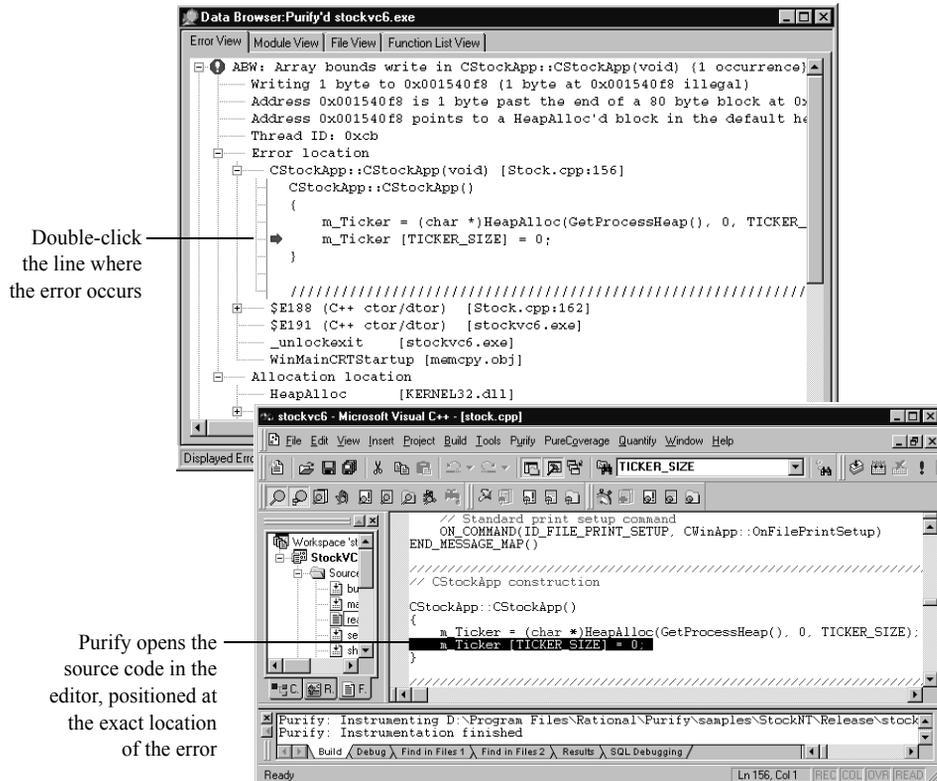
More information? Look up *source code, editing* in the Purify online Help index.

Checking code coverage with Purify

To make sure that you find errors in your code wherever they occur, use Purify to monitor code coverage each time you run your program. With Purify's coverage feature, you can check that you're exercising all your code, especially those parts that have recently been added or modified.

Purify displays coverage data in views that you can sort to find the largest gaps in your testing.

The Module View tab groups functions based on module

The File View tab groups functions based on source file

The Function List View tab lists all functions in the program across modules and files

Click any column header to sort the coverage data

Double-click a function to display it in an Annotated Source window

Purify can also display line-by-line coverage information marked directly on a copy of your code in an Annotated Source window. The color of each line of code indicates whether it is tested, untested, or partially tested, so that you can tell at a glance where you need to tighten up your testing.

Click to display information about color coding

The Annotated Source window displays coverage information in a copy of your code

This line was exercised once

This line was not exercised

Based on the coverage data, refine your approach to exercising your code to make sure you are testing all the critical lines and functions. If you are testing manually, try different menu commands, or enter new values for variables. If you are testing automatically, revise or add test scripts.

More information? Look up *coverage data (C/C++)* in the Purify online Help index.

Comparing program runs

When you are satisfied that you've made good progress in eliminating errors, and that you can exercise the parts of your program that most need testing, rebuild. Then rerun the program under Purify.

After rerunning your corrected program, you can easily compare runs to verify your corrections. Purify's Navigator window, which you can display from the Purify **View** menu, helps you keep track of multiple runs and multiple programs.

The Navigator window groups runs by program

A color-coded icon indicates the maximum message severity displayed in the error view for the run
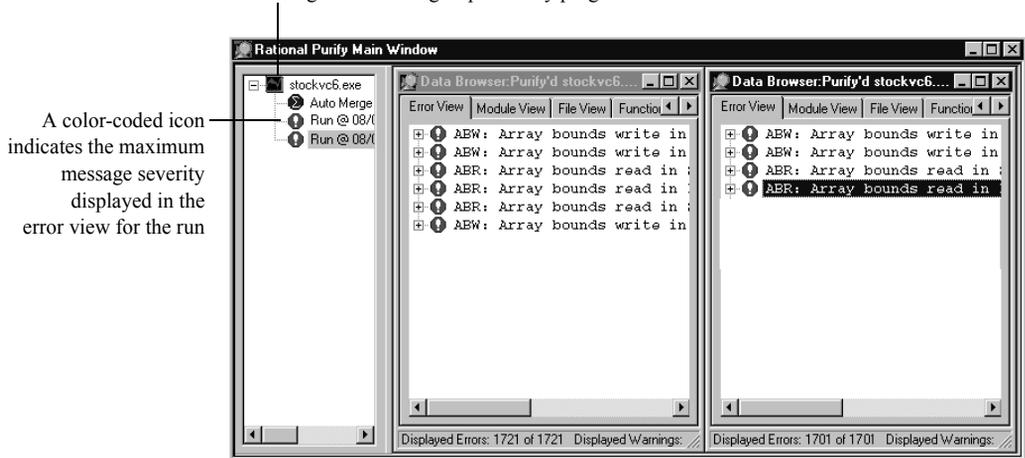
More information? You can compare coverage data from different runs using the Compare Runs tool ▵ . Look up *comparing runs* in the Purify online Help index.

Saving Purify data

You can save Purify error data from a run and analyze it later, share it with other members of your team, or include it in reports. Purify can save data in the following formats:

- Purify data files (.pfy, .pcy). The file extension Purify uses depends on whether you are saving error data alone, or error and coverage data. You can save merged coverage data to PureCoverage data files (.cfy).

- ASCII text files (.txt). You can process this data with scripts or use it in spreadsheet and word-processing applications.

More information? Look up *saving data* in the Purify online Help index.

Advanced features

Customizing instrumentation

Purify uses one of the following error-checking instrumentation levels as the default for each module, depending on the module's size and the availability of debug and relocation data:

- *Precise* instrumentation, which provides full run-time error detection to pinpoint problems in any part of your program

- *Minimal* instrumentation, which improves Purify's performance while providing a basic level of error detection

For coverage monitoring, Purify uses one of the following levels as the default:

- *Line-level* instrumentation, which reports line-by-line coverage data

- *Function-level* instrumentation, which improves performance but reports only function-by-function coverage data

Use the **PowerCheck** tab in the settings dialogs to modify default levels for error detection . . .

and for coverage monitoring

Click to override the defaults for individual modules
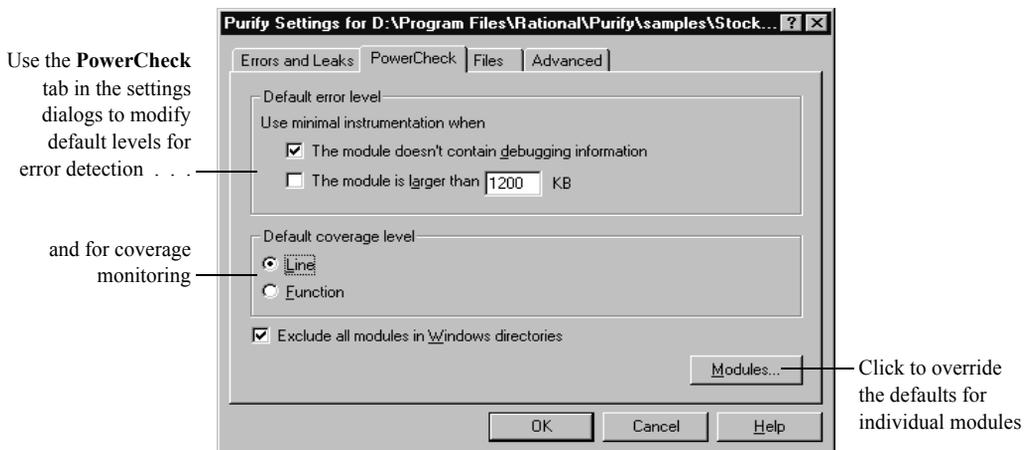
You can override the default and specify the level for each module to meet your own requirements.

Select one or more modules in the list

Then specify the instrumentation level for the selected modules

Purify Module Instrumentation

Module	Purify	Size	Coverage	Debug
d:\program files\rational\purify\samples\stocknt\rel...	(Precise)	348246	(Line)	Yes
c:\winnt35\system32\rpcltc1.dll	(Minimal)	13584	(Exclude)	No
c:\winnt35\system32\ntshrui.dll	(Minimal)	37648	(Exclude)	No
c:\winnt35\system32\ole32.dll	(Minimal)	704272	(Exclude)	No
c:\winnt35\system32\netrap.dll	(Minimal)	17168	(Exclude)	No
c:\winnt35\system32\netapi32.dll	(Minimal)	224528	(Exclude)	No
c:\winnt35\system32\samlib.dll	(Minimal)	41744	(Exclude)	No
c:\winnt35\system32\msvcrt.dll	(Minimal)	254005	(Exclude)	Yes
c:\winnt35\system32\version.dll	(Minimal)	36112	(Exclude)	No
c:\winnt35\system32\lz32.dll	(Minimal)	12560	(Exclude)	No

Error level
- ○ Default
- ○ Precise
- ○ Minimal

Coverage level
- ○ Default
- ○ Line
- ○ Function
- ○ Exclude

☑ Show full path names

Clear Cache   OK   Cancel   Help

Try using the Precise error level for the most critical modules in your program and the Minimal level for the others. Later, you can change the Minimal level to Precise for a thorough check of the other modules.

More information? Look up *instrumentation levels (C/C++)* and *powercheck settings (C/C++)* in the Purify online Help index.

Using just-in-time debugging

Purify's just-in-time debugging support provides instant access to your debugger when you need to solve tough problems. Click 🖑 to enable Break on Error. Purify now stops your program just before an error executes so that you can debug it. You can also run a Purify'd program directly under the debugger.

Error View

⊞ ❶ ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}

With just-in-time debugging, Purify raises a breakpoint exception when it detects an error or warning

stockvc6$PC_D_Program Files_Rational_Purify_samples_StockNT_Release.exe - Application E...

❌ The exception Breakpoint
A breakpoint has been reached.
(0x80000003) occurred in the application at location 0x6fca82c4.

Click on OK to terminate the application
Click on CANCEL to debug the application

[OK]   [Cancel]

Click **Cancel** to explore the error in your debugger

Displayed Errors: 1 of 1   Displayed Warnings: 0 of 0   Bytes leaked: 0+0

To quickly debug *only* the most critical errors in your program, use Break on Error together with Purify filters. First, filter out all the less critical messages, then enable Break on Error. Purify breaks only for the unfiltered messages. When you're ready to debug the remaining errors, just disable the filters.

More information? Look up *break on error tool (C/C++)* in the Purify online Help index.

Using Purify standalone

When you don't need all of the Microsoft Visual Studio resources, you can use Purify standalone. Purify's independent user interface provides the same error-detection and coverage capabilities as when you use Purify integrated with Visual Studio.

Note: You can also use Purify's independent user interface while continuing to work integrated with Visual Studio by deselecting **Embed Data Browsers** in the Purify Settings menu.

To use Purify as a standalone application, launch Purify from the Start menu. Then click **Run** in the Purify Welcome Screen to display the Run Program dialog.

Use the Browse button to select the program that you want to check . . .

Run Program

Program name:

Command-line arguments:

Working directory:

Collect:
- ○ Error and leak data
- ● Coverage, error, and leak data
- ○ Memory profiling data

☐ Run under the debugger
☑ Pause console after exit

Run — Then click **Run**

Cancel

Settings...

Help

and select whether you want to collect **Error and leak data**, or **Coverage, error, and leak data**
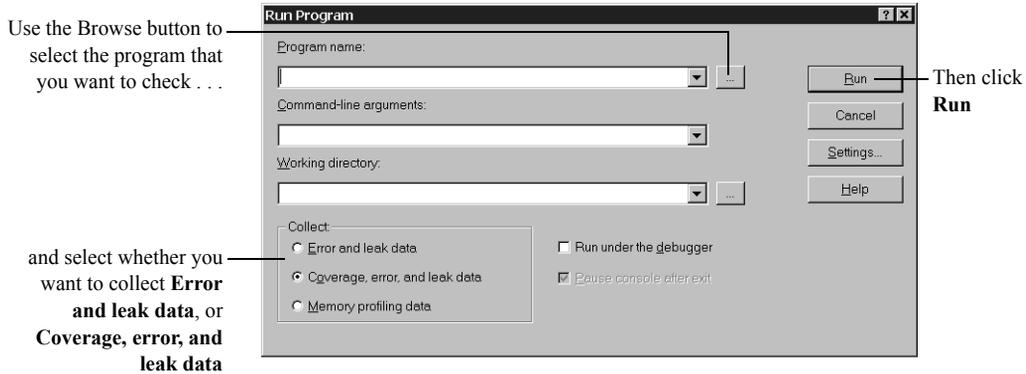
Purify instruments your code and displays the results in a Data Browser window.

More information? For information about a tool, menu command, or dialog, click ▶? and then click the item.

Testing C/C++ code with the command-line interface

Using the Purify command-line interface, you can use Purify with existing makefiles, batch files, or Perl scripts. For example, if you have a test script that runs a program, you can easily modify the script to instrument and run the program. To do this, change the line that runs *Exename*.exe to:

```
purify Exename.exe
```

Alternatively, to run the instrumented version of *Exename*.exe consistently throughout your tests, add this line to the beginning of your test script:

```
purify /Replace=yes /Run=no Exename.exe
```

This line instructs Purify to save the original *Exename*.exe to a .bak file, and to instrument *Exename*.exe but not to run it at this time. Now, whenever your test script runs *Exename*.exe, it runs the instrumented version of the program, providing Purify's detailed diagnostics.

To collect coverage data as well as error data when you run a program from the command line, use the /Coverage option:

```
purify /Coverage=yes Exename.exe
```

You can run Purify without the graphical interface by using the /SaveTextData option. This option saves Purify's diagnostic messages to a text output file. You can use the error and warning messages in this file as additional criteria for your test results.

More information? Look up *command line* in the Purify online Help index.

Extending error checking with Purify API functions

Purify includes a set of API functions that extend its error checking capabilities and give you greater control over tracking errors.

Using Purify's API functions, you can set memory state, test memory state, and search for memory and handle leaks. For example, by default Purify reports memory leaks only when you exit your program. But you can use the API function `PurifyNewLeaks` to check for leaks more frequently. Click the NewLeaks tool to call `PurifyNewLeaks` while your program is running, or add calls to `PurifyNewLeaks` at key points in your code. Purify reports any new memory leaks it has detected since the last time you called the function. This periodic checking enables you to track memory leaks more closely.

You can call Purify API functions from the Purify View menu as your program runs. You can also call them from the QuickWatch dialog in the Visual Studio debugger, as well as by including them in your code.

More information? For the complete listing of Purify API functions, including functions related to coverage monitoring, look up *api function list*. For instructions on using the functions, look up *api functions, using* in the Purify online Help index.

Using IBM Rational product integrations

Rational tools integrate into your working environment to help you do your job more effectively and efficiently. For example, you can use Purify with IBM Rational ClearQuest™, a change request management tool, and with Rational Robot, a functional testing tool.

Using Purify with ClearQuest

If you have IBM Rational ClearQuest installed, you can submit a defect as soon as Purify detects an error or warning, or when you find a coverage problem.

Right-click on an error message and select **Submit ClearQuest Defect**
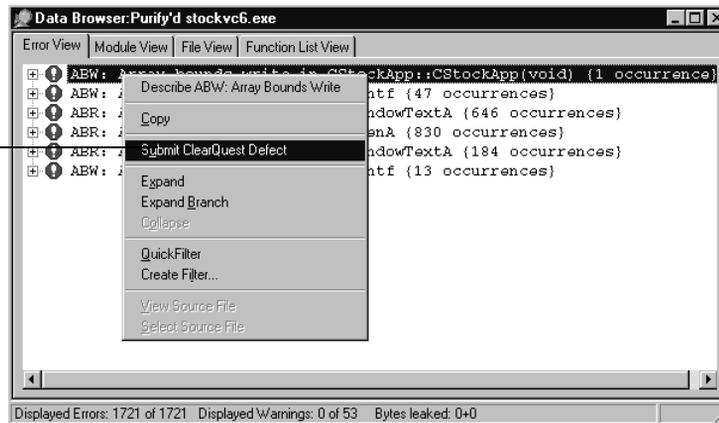
Purify automatically supplies entries for a number of fields in the submission form and specifies the category of error. You can easily attach Purify data files to further document the error.

Using Purify with Rational testing tools

If you have Robot installed, you can set a playback option in Robot to collect Purify error and leak data when you run a Robot test script. Purify detects memory errors as the code is executed. Robot also includes a playback option that allows you to collect code coverage information as well as error and leak data.

More information? Look up *clearquest* and *robot* in the Purify online Help index, and refer to the ClearQuest and Robot documentation.

Now you're ready to put Purify to work on your C/C++ code. Remember that Purify's online Help contains detailed information to assist you.

Purify for .NET managed code developers and testers

What it does

Purify finds and reports memory leaks in .NET managed code (assemblies, .exe's, .dll's, OLE/ActiveX controls, and COM objects).

Memory leaks in managed code

Managed code can leak memory, which can cause problems for your program.

The .NET garbage collector automatically removes memory objects that your program no longer needs, and so avoids most of the memory leaks that occur in other programming contexts. But managed code applications can still consume more and more memory over time. There are two major categories of leaks in managed code: object references that are no longer needed, and system resources that are not freed.

Object references that are no longer needed

Very often, managed code retains references to memory that it no longer needs, and this prevents the memory from being garbage collected. Managed code objects typically include references to other objects, so a single object can hold an entire tree of objects in memory. Problems can occur, for example, when you do any of the following:

▫ Add objects to arrays and forget about them.

▫ Retain references to an object until the next time you use the object. A menu command, for example, can create an object and not release references to the object until the next time the command is invoked which may never happen.

▫ Change an object's state while some references still reflect the old state. For example, when you store properties for a text file in an array and then store properties for a binary file, some fields, such as "number of characters," continue to hold memory that is no longer needed.

▫ Allow a reference to be pinned by a long-running thread. Setting the object reference to NULL does not help; the memory won't be garbage collected until the thread terminates or goes idle.

System resources that are not freed

Managed code methods can also allocate heap memory that exists outside of managed data instances, such as resources for windows and bitmaps. Managed code allocates these resources by calling unmanaged code like native C and C++ routines.

How Purify can help

Purify helps you find these managed code memory leaks by reporting the methods, classes, and objects that are responsible for monopolizing large chunks of memory that the garbage collector does not free.

Using the data Purify gathers, you can zero in on memory problems. Once you've located them, you can eliminate references to unneeded objects, or force garbage collections in key areas of your code.

More information? In addition to detecting excessive memory consumption with Purify, you can also improve your application's performance and increase your confidence in your testing using the other PurifyPlus tools, PureCoverage and Quantify. PureCoverage can show you the areas in your code that your tests are not reaching, and Quantify can help you find the bottlenecks that slow down your code. For more information, read *Getting Started: PureCoverage* on page 41, and *Getting Started: Quantify* on page 57.

The basic steps

To use Purify to profile managed code memory usage:

1 Run your managed code program with Purify.

2 Take a snapshot when memory usage stabilizes.

3 Execute code that may be leaking and take another snapshot.

4 Compare the two snapshots to identify methods that may be causing memory problems.

5 Pinpoint the leaked objects allocated by these methods, and identify the references that are preventing the objects from being garbage collected.

The following pages show you how to use Purify integrated with Microsoft Visual Studio, but you can also use Purify in other ways. Read the following:

▪ *Using Purify standalone* on page 37

▪ *Integrating Purify into your managed code test environment* on page 38.

Running your managed code program with Purify

To Purify your managed code program in Visual Studio, open your project in Visual Studio, then engage Purify using the Purify menu.

Build and execute your program as usual, using commands from the Visual Studio menu. To get the maximum level of detail in Purify memory profiling data, build your program with debug data.

More information? Look up *running programs* in the Purify online Help index.

As your program runs, Purify intercepts and tabulates messages related to memory usage from the .NET runtime environment. Based on these messages, Purify keeps track of how much memory your program has allocated to each method and object at any given time.

Taking snapshots of memory use

To zero in on memory leaks in your managed code program, wait until your application's memory usage has stabilized (typically after it completes initialization), then select the menu item **PurifyPlus > Purify > Run Control > Snapshot Data** to take a snapshot of the current memory usage status. This snapshot is your baseline for investigating how your program uses memory as it runs.

Now exercise the program in a way that you suspect may be leaking memory. As your program runs, the Purify Data Browser's Memory tab displays a graph that indicates the amount of memory your program is using.

Take your first snapshot when your program's baseline memory usage has stabilized

Watch for increasing memory usage, then take a second snapshot
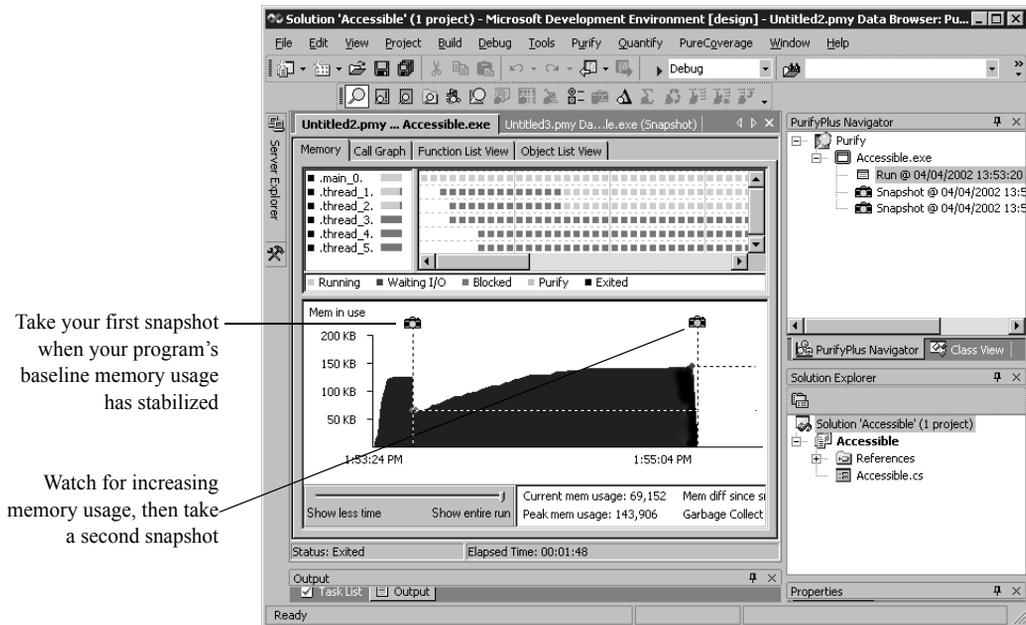
Watch the graph for fluctuations in memory usage. A large increase in memory usage may indicate a problem, especially when you can't reduce it by selecting the menu item **PurifyPlus > Purify > Run Control > Garbage Collect** to force a garbage collection.

Now take another snapshot so that you have a "before" and "after" record of what's going on, and exit your program.

More information? Look up *taking snapshots* and *garbage collection* in the Purify online Help index.

Comparing snapshots to identify problem methods

Select your second snapshot in the Navigator and select the menu item **PurifyPlus > Purify > Run Control > Compare Runs** to compare the second snapshot with the first.

Purify now displays a call graph showing the methods that are responsible for allocating the largest amounts of memory during the interval between the first and second snapshots.

The thickest lines indicate the paths where the difference in memory between the two snapshots is greatest

The call graph overview helps you orient yourself within the call graph
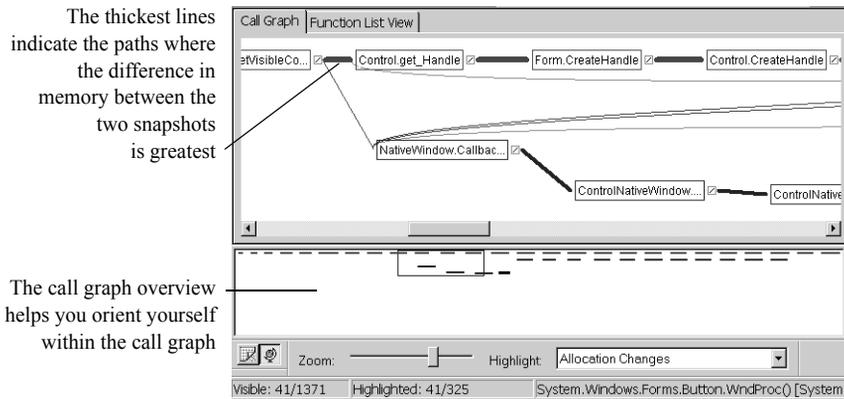
The call graph also shows you the calling relationship between methods. This can give you clues about which methods are holding references to unneeded objects and preventing the garbage collector from doing its job.

Move your cursor over the method or path you want to investigate. A tool tip pops up to give you memory-related statistics for that method.

Memory usage data is available directly from the call graph
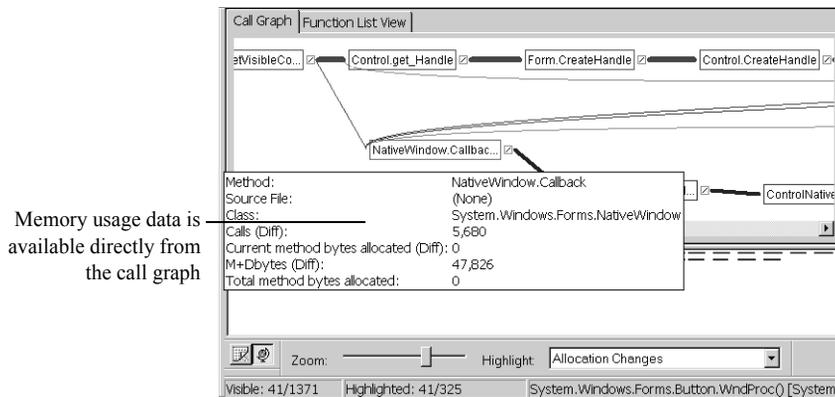
This allows you to zero in on the method that is consuming memory, as well as its descendants.

To view your code from within Purify, right-click a method for which source is available, then select **Source File**.

More information? Look up *diffing snapshots, call graph,* and *source code, viewing* in the Purify online Help index.

Diagnosing leaks with the Function List View tab

The Function List View tab in the Data Browser provides a textual, non-hierarchical view of the same data that appears in the call graph. You can do full-program sorts in the Function List View to find the biggest memory-consuming methods in your entire program.

Click a column header to sort the memory profiling data

Call Graph	Function List View					
Method	Calls (New)	Calls (Base)	Current method bytes allocated (Diff)	Class	Source File	
ColorOptions.Calculate	132	1	18,720	ColorOptions	(None)	
ButtonBase.CommonLa...	132	1	9,424	System.Windows.Forms.B...	(None)	
ButtonBase.CommonRe...	132	1	5,808	System.Windows.Forms.B...	(None)	
Control.PaintBackground	167	2	3,348	System.Windows.Forms.C...	(None)	
DibGraphicsBufferMana...	264	2	2,480	DibGraphicsBufferManager	(None)	
Graphics.BeginContainer	132	1	2,096	System.Drawing.Graphics	(None)	
Graphics.FromHdcIntern...	173	8	2,080	System.Drawing.Graphics	(None)	
Graphics.Save	167	2	2,032	System.Drawing.Graphics	(None)	
ButtonBase.DrawText	132	1	1,488	System.Windows.Forms.B...	(None)	
ValueCollection.GetEnu...	987	12	288	ValueCollection	(None)	
LayoutOptions.Layout	132	1	148	LayoutOptions	(None)	
Message.GetLParam	29	2	96	System.Windows.Forms.M...	(None)	
LayoutOptions.CalcText...	132	1	64	LayoutOptions	(None)	
MessageBox.ShowCore	32	0	56	System.Windows.Forms.M...	(None)	
ControlPaint.GetFocusR	97	0	52	System.Windows.Forms.C	(None)	

Visible: 1371/1371   System.Windows.Forms.Contr

More information? Look up *function list view tab* in the Purify online Help index.

Focusing on a method with the Function Detail window

By double-clicking any method in the call graph or function list view, you can open a Function Detail window. This window shows how the method, its callers, and its descendants allocated memory.

Method:	ColorOptions.Calculate
Calls (Diff):	131
Calls (New):	132
Calls (Base):	1
Current method bytes allocated (Diff):	18,720
Total method bytes allocated:	20,436
Number of Objects:	120
M bytes (New):	18,720
M bytes (Base):	0
M+Dbytes (Diff):	18,656
M+D bytes (New):	18,826
M+D bytes (Base):	170
Class:	ColorOptions
Source File:	(None)
Hidden methods:	Yes

Double-click a method in the Caller or Descendant column to see the memory data for that method

Caller	Current method bytes allocated (Diff)	
ButtonBase.PaintWorker	18,656	

Descendant	Current method bytes allocated (Diff)	
Color.get_G	0	
Graphics.GetNearestColor	0	
Color.op_Equality	0	

Callers: 1   Descendants: 11   ColorOptions.Calculate()

More information? Look up *function detail window* in the Purify online Help index.

If the amount of memory attributed to any method seems unexpectedly high, it may be the case that another method, possibly a descendant, has created a reference to an object that is preventing the memory from being garbage collected. For example, a descendant method may have created a static variable as part of a string array. This would keep the memory for the entire array from going out of scope, which may slow your program down, and even kill it.

When you've located a method that appears to be causing memory problems, go on to look at the method's objects. Purify provides extensive information not only about methods, but also about all objects in your program and their use of memory.

Looking for unneeded objects

Objects that a program no longer needs often prevent memory from being garbage collected and so, over time, slow down your program. Purify displays comprehensive memory data for objects in several formats, so that you can easily track down this sort of problem.

Note: Use a snapshot data set to examine object data. Comparison data sets, which are generated by selecting the menu item **PurifyPlus > Purify > Run Control > Compare Runs,** do not contain object data.

Getting from a suspicious method to its objects

The Function Detail window, in addition to its information about a method, also lists objects that have been allocated by the method. You can sort the objects in the list by clicking on any column heading.

The objects that the method currently has allocated. Double-click an object to display the Object Detail window with comprehensive memory data for the object

Method:	ColorOptions.Calculate
Calls:	132
Current method bytes allocated:	18,720 (21.07% of Focus)
Total method bytes allocated:	20,592
Number of Objects:	120
M+D bytes:	18,826 (21.19% of Focus)
Avg M bytes:	156
Min M bytes:	156
Max M bytes:	156
Class:	ColorOptions
Source File:	(None)
Hidden methods:	0 bytes (0.00% of M+D bytes)

Object Name	Class Name	Size	O+R Size	GCs Survived
System.String 205E0A20	System.String	156	156	2

Note that Function Detail windows for snapshots include pie charts showing memory allocation

Caller	Calls	Current method bytes allocated
ButtonBase.Paint...	132	18,826

Descendant	Calls	Current method bytes allocated
Color.FromArgb	264	106
SystemColors.get...	132	0

Callers: 1   Descendants: 11   ColorOptions.Calculate()

Examining object details

When you double-click an object in the Function Detail window, the Object Detail window opens. This window contains complete memory-related information for the object. Use it to identify objects that are holding on to large chunks of memory, and determine how long these objects have been in existence.

The object reference graph shows the objects that reference, and are referenced by, the current object

Zoom: Highlight: Object: Maximum Path to Root

```
.Root — System.Object 215C0018 —
                                  System.String 205C84A0
                                  System.String 205C84B8
                                  System.String 205C86C0     System.String 205E0A20
                                  System.String 205C8788
                                  System.String 205C87A8
```

Details about the object currently selected in the reference graph, including size and creation time

Object Name:	System.String 205E0A20
Class Name:	System.String
Method Name:	ColorOptions.Calculate
Size:	156
O+R Size:	156
GCs Survived:	2
Creation Time:	4:04:03 PM
Line Number:	0
References:	0
Referees:	1
Root Type:	

Name	Value
Object Dump	Not available

References: 0   Referees: 1   System.String 205E0A20

Looking at all allocated objects together

To review the top-level objects in a program, open the Data Browser window for the snapshot that reveals potential memory problems, and click the Object List View tab.

Click any column head to sort
the list

Memory data for all the
currently allocated
top-level objects in the
program

The status bar shows the
selected line number and
the total number of
objects

| Memory | Call Graph | Function List View | Object List View |

Object Name	Method Name	Size	O+R Size	GCs Survived	Line Number
System.Object 215C0018	Runtime00	2,064	10,622	0	
System.String 205E0A20	ColorOptions.Calculate	156	156	2	
System.String 205C36F0	Unknown	120	120	0	
System.String 205DE45C	Unknown	110	110	1	
System.String 205C60E8	Unknown	108	108	0	
System.String 205DFC2C	Unknown	108	108	1	
System.String 205DB048	Unknown	102	102	0	
System.String 205C4B60	Unknown	100	100	0	
System.String 205DF5FC	Unknown	94	94	1	
System.String 205C4D4C	Unknown	90	90	0	
System.String 205C91C8	Unknown	90	90	0	

Object: 2/183   System.String 205E0A20   ColorOptions.Calculate()

The object list shows all top-level objects that were allocated at the time the snapshot was taken. In addition to the size of the objects, the object list provides information such as the time the object was created and the number of garbage collections it has survived. You can sort the list to find the objects that are holding on to the most memory, and the oldest objects in the list.

You can open the Object Detail window for any object by double-clicking the entry for the object.

When you locate an object that may no longer be needed, look at your code. If you determine that the object is in fact no longer needed, modify your code to release all references to the object so that the object can be garbage collected.

More information? Look up *function detail window*, *object detail window*, and *object list view tab* in the Purify online Help index.

Saving Purify memory profiling data

You can save Purify data and analyze it later, share it with other members of your team, or include it in reports. Purify can save managed code data in the following formats:

- Purify memory profiling files (`.pmy`). You can open these files and view them in Purify, just as you would any run, snapshot, or other dataset.

- ASCII text files (`.txt`). You can process this data with scripts or use it in spreadsheet and word-processing applications.

More information? Look up *saving data* in the Purify online Help index.

Advanced features

Highlighting methods that share key attributes

You can highlight methods in the call graph to display specific memory-related characteristics or to show calling relationships.

Click to display the Highlight list

Select **Maximum Path to Root**, for example, to highlight all methods between the selected method and .Root on the path where the most memory is allocated
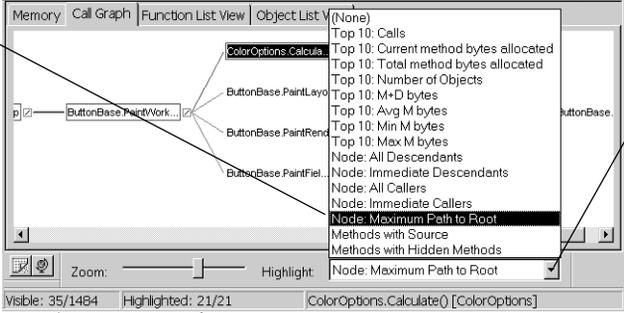
35 of the 1484 methods in the current dataset are displayed in the call graph

All 21 of the 21 functions on the maximum path to .Root are displayed in the call graph

More information? Look up *highlighting* in the Purify online Help index.

Focusing your data

Use Purify's *filter* commands to remove a selected method, or all methods in a class file, from the set of data that Purify has collected. Alternatively, use *subtree* commands to focus on or remove a specific method and all its descendants from the dataset. Right-click a method in the call graph, function list view, or function detail to perform these operations.

You can hide or delete individual methods, all methods in a class, or entire subtrees.

Filter	▶	Hide Method LeakSample.<init>
Subtree	▶	Hide Class LeakSample
		Delete Method LeakSample.<init>
Expand/Collapse	▶	Delete Class LeakSample
Line Scale Factors	▶	Undo Last Filter Operation
Colors	▶	
Method Name...		Filter Manager...
Source File		
✔ Data Browser... Ctrl+B		

The Filter Manager offers additional filtering options

Hide methods or subtrees to sum up their memory and attribute it to their callers; delete them to discard their memory completely

Filter	▶	
Subtree	▶	Focus on Subtree
		Hide Subtree
Expand/Collapse	▶	Delete Subtree
Line Scale Factors	▶	Undo Hide Subtree
Colors	▶	
Method Name...		Reset to .Root.
Source File		
✔ Data Browser... Ctrl+B		

Select **Focus on Subtree** to delete all methods except those in the subtree

Purify has undo capabilities for all filter and subtree commands so that you can easily return to any previous dataset configuration.

The call graph also provides a series of expand and collapse commands that work with subtrees. Unlike the filter and subtree commands, however, these commands affect only what is displayed in the call graph; they do not change the dataset.

In addition to the menu commands, you can use the Filter Manager to select the data you need.

Filter Manager: Memory Profiling Data

Classes | Methods

Filter	Enabled	Memor
☑ ArrayListEnumeratorSimple	Yes	Yes
☑ ArraySubsetEnumerator	Yes	Yes
☐ ColorOptions	No	Yes
☑ CommonHandles	Yes	Yes
☐ ComponentManager	No	Yes
☐ ControlCollection	No	Yes
☐ ControlNativeWindow	No	Yes
☐ Decoder	No	Yes

☐ Show full path names

Memory
○ Retain
○ Delete

OK   Cancel   Help
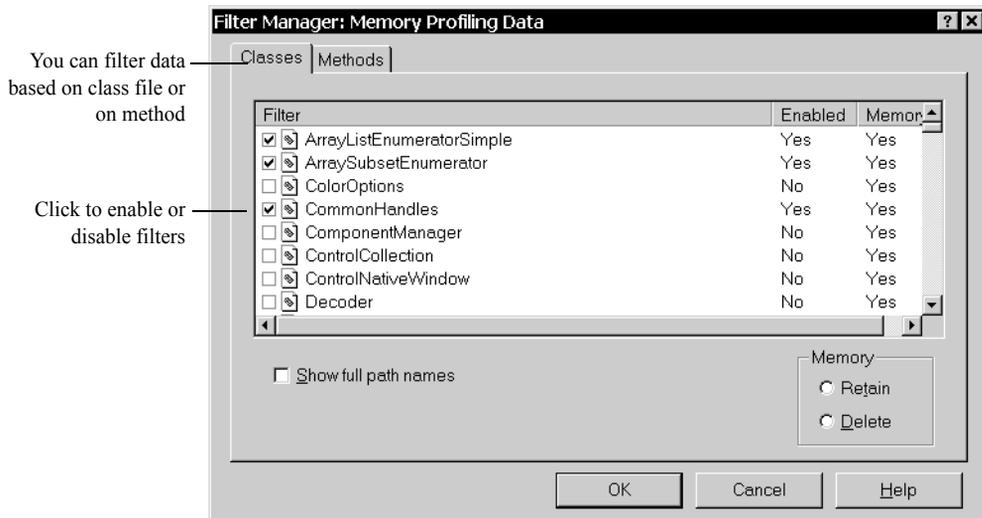
More information? Look up *filtering data* and *subtrees* in the Purify online Help index.

Using Purify standalone

When you don't need all of the Microsoft Visual Studio resources, you can use Purify standalone. Purify's independent user interface provides the same memory profiling capabilities as when you use Purify integrated with Visual Studio.

To use Purify as a standalone application, launch Purify from the Start menu. Then click **Run** in the Purify Welcome Screen to display the Run Program dialog.

Use the Browse button to select the managed code program that you want to profile . . .

and select the button for collecting **Memory profiling data**
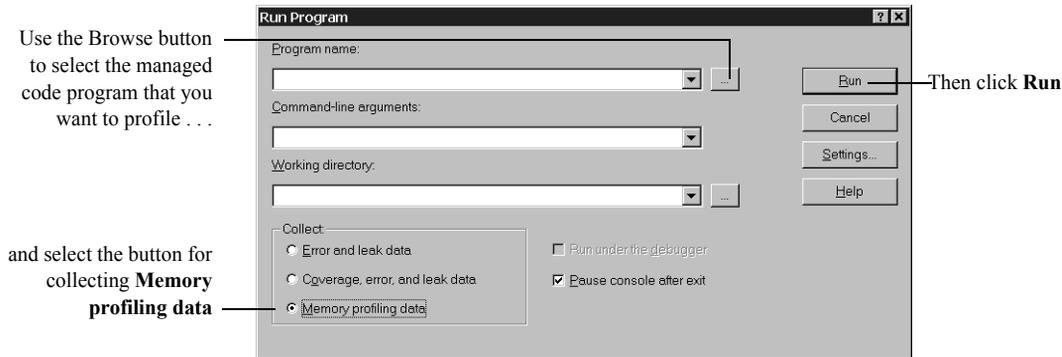
Then click **Run**

Purify instruments your code and displays the results in a Data Browser window.

More information? For information about a tool, menu command, or dialog, click ⬛ and then click the item.

Integrating Purify into your managed code test environment

The Purify command-line interface makes it possible for you to collect memory profiling data in your automated testing environment. Modify existing makefiles, batch files, or Perl scripts to run your program under Purify. For example, if you have a test script that runs a managed code program, change the line that runs it to:

```
Purify /SaveData /Net Exename.exe
```

This command runs your managed code program and collects memory profiling data, then saves the data to a .pmy file that you can open and analyze in the Purify interface or share with other members of your team.

Use the `/SaveTextData` option instead of the `/SaveData` option to save your data in a .txt file. You can develop scripts to process this data and generate reports about your program's use of memory. For example, you might want to compare the dataset from the current nightly test with that from the previous nightly test to detect memory-related regressions as soon as they occur.

To control your automated data collection and ensure that you generate comparable datasets from every test, use the Purify API. Read *Controlling managed code memory profiling with the Purify API*, immediately following.

More information? Look up *command line* in the Purify online Help index.

Controlling managed code memory profiling with the Purify API

Purify includes a set of API functions that give you greater control over its memory profiling capabilities.

The API is especially useful if you are doing automated testing. You can programmatically determine the parts of your code that are profiled, excluding your program's initialization activities and focusing on specific modules or routines. You can also clear your data after initialization, then continue collecting data as your program runs, and save it just before the program terminates; this is equivalent to comparing two snapshots in the Purify user interface.

More information? For the complete listing of Purify API functions, including functions related to coverage monitoring, look up *api function list*. For instructions on using the functions, look up *api functions, using* in the Purify online Help index.

Now you're ready to put Purify to work on your managed code. Remember that Purify's online Help contains detailed information to assist you.

Getting Started: PureCoverage

What it does

Before you ship your products, you need the assurance that the code you're responsible for has been exercised thoroughly—every line, every function, procedure, or method.

That's where PureCoverage® can help you get ahead. PureCoverage automatically evaluates the completeness of your testing and pinpoints the parts of your code you're failing to reach. As a C++, Visual Basic, Java, or .NET managed code programmer, you can easily monitor testing coverage as you run your program. As a quality engineer, you can include PureCoverage in your test harness to generate comprehensive coverage reports automatically for every test you run.

Using PureCoverage you can:

- See immediately what percentage of your code has and has not been exercised

- Identify untested, or insufficiently tested, functions, procedures, or methods

- Locate individual untested lines in your source code

- Customize data collection for maximum efficiency

- Customize displays to focus on the details you need

- Merge coverage data from multiple runs of a program

- Save coverage data to share with other team members or to generate reports

- Monitor code coverage from within your development environment by using the PureCoverage integration with Microsoft Visual Studio

Check every component in your program

PureCoverage analyzes coverage for every component in:

- C/C++ code in .exe's, .dll's, OLE/ActiveX controls, and COM objects

- p-code .exe's, native-code .exe's, .dll's, OLE/ActiveX controls, and COM objects

- Java applets, class files, `jar` files, and code launched by container programs

- .NET managed code .exe's generated in Microsoft Visual Studio.

- Components launched from container programs such as Microsoft Internet Explorer, the Microsoft Transaction Server, jexegen'd executables, Jview.exe, Tstcon32.exe, Firefox, or any Microsoft Office application

- Microsoft Excel and Microsoft Word plug-ins

Note that any discussion that applies to functions and modules also applies to Java methods and class files, and to Visual Basic procedures and object libraries.

Use PureCoverage throughout the engineering cycle

Start using PureCoverage early in the development and testing cycles to find and eliminate gaps in both your formal and informal tests. You'll know that you're exercising all your code right from the beginning and finding errors while there's time to correct them. Continue using PureCoverage whenever you exercise new or modified code, up to the time of final product release.

Tips for development engineers

Let's say you've just put together a new routine. You can use PureCoverage to collect coverage data and easily focus on the information for your new code. You'll see immediately whether you've tested everything before check-in. PureCoverage provides coverage data with minimal effort on your part.

If you're exercising your code manually, use PureCoverage to monitor and guide your testing as you work. PureCoverage shows you interactively the percentage of your code's functions, procedures, or methods that you've exercised.

PureCoverage automatically integrates with Microsoft Visual Studio, so you can use it without changing the way you work if you're developing code in this environment.

Tips for test engineers

As a test engineer, use PureCoverage to gauge how well your test suite is keeping pace with the evolution of the program you're testing. You can add one or two lines of code to your test scripts to run PureCoverage automatically

in batch mode whenever you test. With immediate and continuous feedback about the effectiveness of your test suite, you can guarantee that you are exercising every modification in the program you're testing.

More information? PureCoverage's online Help provides detailed reference information and step-by-step instructions for using PureCoverage. For a start, look up *purecoverage, introduction* in the online Help index.

The basic steps

With PureCoverage®, you can ensure that all of your code is exercised in a few easy steps:

1 Run a program with PureCoverage.

2 Get the big picture with the Coverage Browser and Function List windows. Use PureCoverage filters to focus on the areas that concern you most.

3 Identify unexercised lines in the Annotated Source window.

4 Modify your test run to cover missed lines, conditions, functions, procedures, or methods.

5 Rerun the program to verify that you've improved coverage. Save coverage data to share information with other team members.

This chapter describes how to use PureCoverage as a standalone desktop application. The same principles apply when you use PureCoverage integrated with Microsoft Visual Studio, or when you incorporate it into your test harness. For more information, read *Integrating PureCoverage with your development desktop* on page 49 and *Integrating PureCoverage in your test environment* on page 54.

Note: PureCoverage monitors coverage of functions and, if debug line information is available, of individual lines as well. If you want line-level data for programs built in release mode, you must supply debug line information. For specific instructions, look up *debug data* in the PureCoverage online Help index.

Running a program

To monitor code coverage for an application, launch PureCoverage from the Start menu. Then click **Run** in the PureCoverage Welcome Screen to display the Run Program dialog.
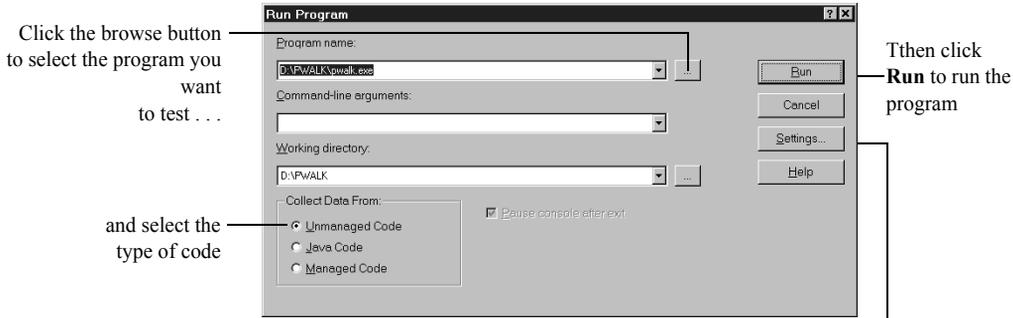
Click the browse button to select the program you want to test . . .

and select the type of code

Tthen click **Run** to run the program

You can click **Settings** to specify data-collection instructions for special situations, such as selective instrumentation of specific modules

Your program begins to execute. As it runs, PureCoverage collects comprehensive information about what lines and functions have been exercised.

PureCoverage displays a Run Summary window as the program runs, showing the current status of program coverage.

As the program runs, the Run Summary window shows the number of functions, procedures, or methods that have and have not been exercised

A color-coded indicator shows how calls are distributed among the functions, procedures, or methods
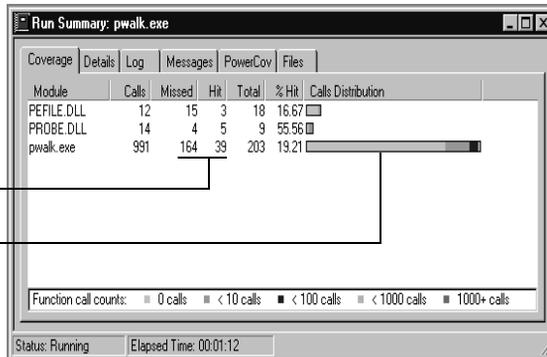
Getting the big picture

The Coverage Browser and Function List windows show you the overall coverage status of your program at a glance:

▫   Every function that has not been called is reported as missed. Those that have been called at least once are reported as hit.

▫   The number of lines of code missed and hit is also reported, if debug line information was available to PureCoverage.

With this information, you can easily identify testing hot spots—major areas that your tests have not covered.

The Coverage Browser window provides coverage data organized hierarchically according to source file.

The Module View tab groups data by file within modules

The File View tab groups data by file across all modules in your program

The Coverage Browser window shows hierarchical coverage information for functions, procedures, or methods . . .

and for lines
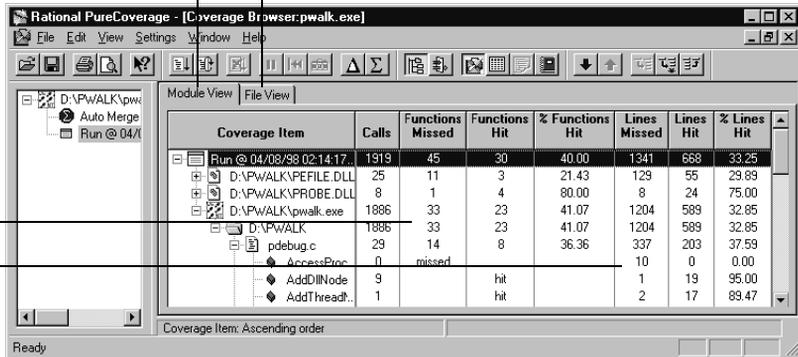
The Function List window provides a textual, non-hierarchical view of the same data. You can do full-program sorts in the Function List window to find the least tested components in your entire program.

Click any column heading for full-program sorting
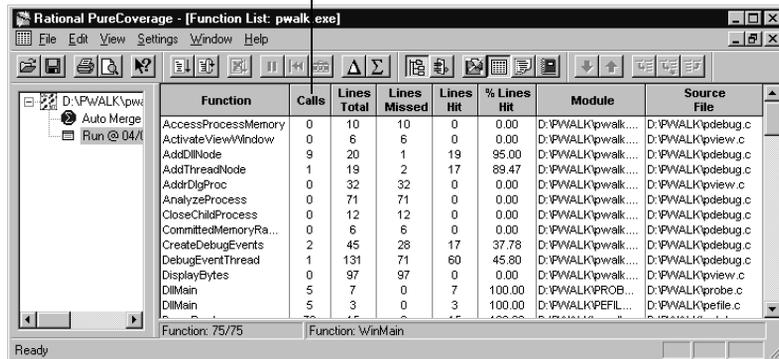
More information? To learn how to customize the data display, look up *coverage browser window* and *function list window* in the PureCoverage online Help index.

Focusing coverage data with filters

PureCoverage collects coverage information for every module in your program, but, by default, does not display all the data it collects. In order to highlight the coverage information that you are most likely to find interesting, PureCoverage applies a default filter set to hide the data for certain system and third-party components of your program.

To see the data that PureCoverage has filtered out, or to change the filtering to display other information that concerns you, click the Filter Manager tool 🔳 to open the Filter Manager dialog.

Use the Modules tab to filter out data by module

Use the Files tab to filter out data by source file

Use the Functions/Procedures/Methods tab to filter out data by function, procedure, or method

Click to turn the filter on and off

Type other patterns to use as filters
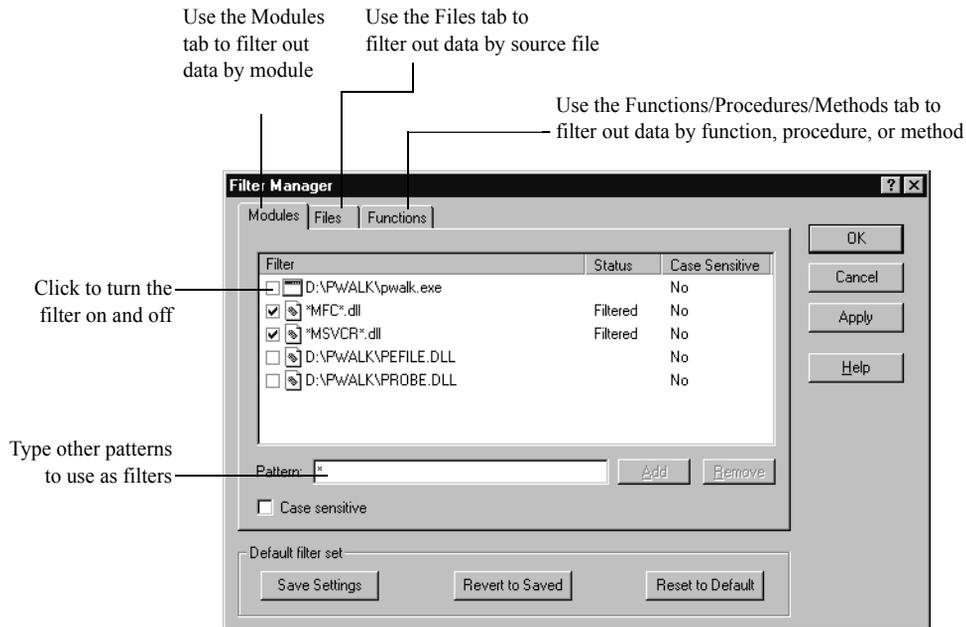
More information? Look up *filters* in the PureCoverage online Help index.

Identifying unexercised lines

PureCoverage displays line-by-line coverage data as annotations in a copy of your source file. Double-click a function, procedure, or method in the Coverage Browser or Function List window to display the code in the Annotated Source window.

Click to display or change the color coding for coverage annotations

The Annotated Source window displays a copy of your source with notes about line coverage

This line was exercised twice

This line was not exercised
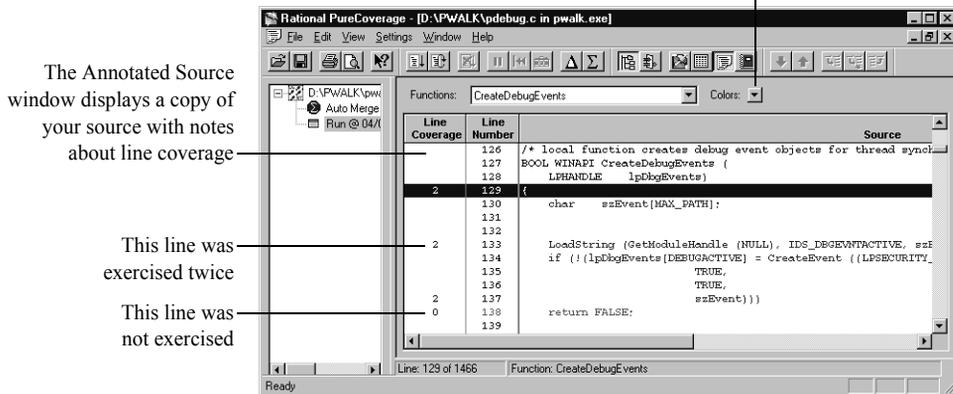
By default, PureCoverage displays untested lines in red, tested lines in blue, and dead lines (typically in functions, procedures, or methods for which no active call is present in the code) in black. PureCoverage displays partially tested multi-block lines in pink. These lines often occur in conditional expressions for which you haven't tested the entire range of possible values.

You can ensure that multi-block lines are fully tested by using the QuickWatch dialog in Visual Studio. With the program running, type in the name of the partially tested function or procedure and supply the parameter values you still need to test.

More information? Look up *annotated source window* and *colors, using* in the PureCoverage online Help index. For help with the QuickWatch dialog, see your Visual Studio documentation.

Modifying your test run

Now you know what sections of code you missed when you exercised the program. If you're running the program informally, consider how you can exercise the code that you missed previously. If you're working with a test suite, you can add or adjust test scripts to improve coverage.

In either case, with the information PureCoverage provides, you're working with your eyes open. You know what parts of your code need to be covered—no guesswork.

Rerunning your program

Now test again, and check your results. Check not only the coverage data for the new run, but also the Auto Merge data. The Auto Merge data is a composite of the coverage data from the new run and any available previous runs of the program.

The Navigator window identifies merged data sets
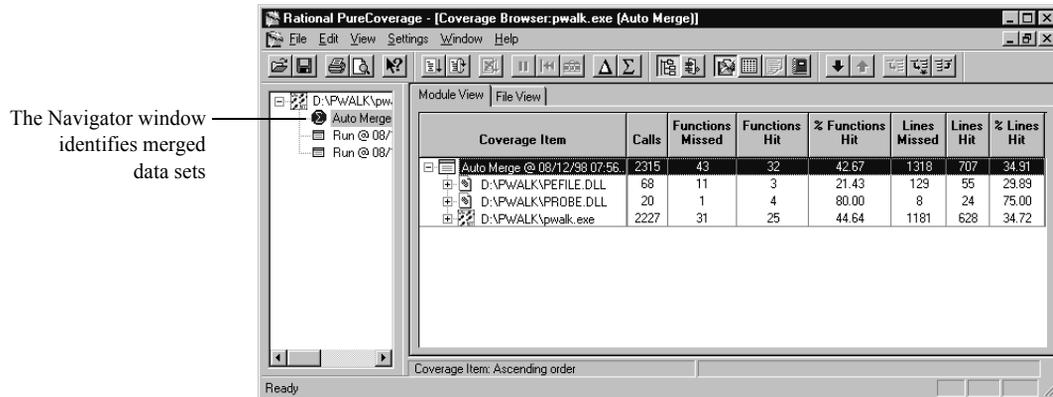
You can also merge data for specific runs manually.

More information? Look up *merging runs* in the PureCoverage online Help index. For information about merging data from a series of tests automatically, read "Integrating PureCoverage in your test environment" on page 54.

Saving coverage data

PureCoverage saves you time during testing by making it easy to share information with other team members. To save data, and share information, click the Save Copy As tool 🖫

PureCoverage supports two data formats:

▫ PureCoverage data files (.cfy), which you can open later in PureCoverage to analyze or to compare to future program runs. Or you can share .cfy files for use by other team members who are using PureCoverage.

- ASCII text files (.txt), for use in spreadsheet and word-processing programs. You can also communicate testing status effectively by including .txt files in email messages or bug reports.

You can also save data from the command line, which is essential if you're running PureCoverage without the interface for your nightly tests.

More information? Look up *saving data* in the PureCoverage online Help index.

Advanced features

PureCoverage provides powerful features that can help you make maximum use of the coverage data you've collected . For example, you can:

- Integrate PureCoverage with your development desktop

- Fine tune data collection

- Use selective instrumentation to collect data for a subset of your program

- Zero in on key program areas

- Integrate PureCoverage in your test environment

This section gets you started using these features to monitor your code more efficiently, and to focus on untested sections of code quickly and easily.

Integrating PureCoverage with your development desktop

PureCoverage's integrations put powerful coverage data within easy reach while you develop and test your code using your favorite tools. You can integrate PureCoverage with Microsoft Visual Studio, Rational Robot, and Rational ClearQuest™.

During installation, a PureCoverage menu is automatically added to Visual Studio so you can monitor your code at any time during development, without leaving your development environment. .

Select the PurifyPlus > PureCoverage menu, then run your program

View and work with coverage data directly within Visual Studio .
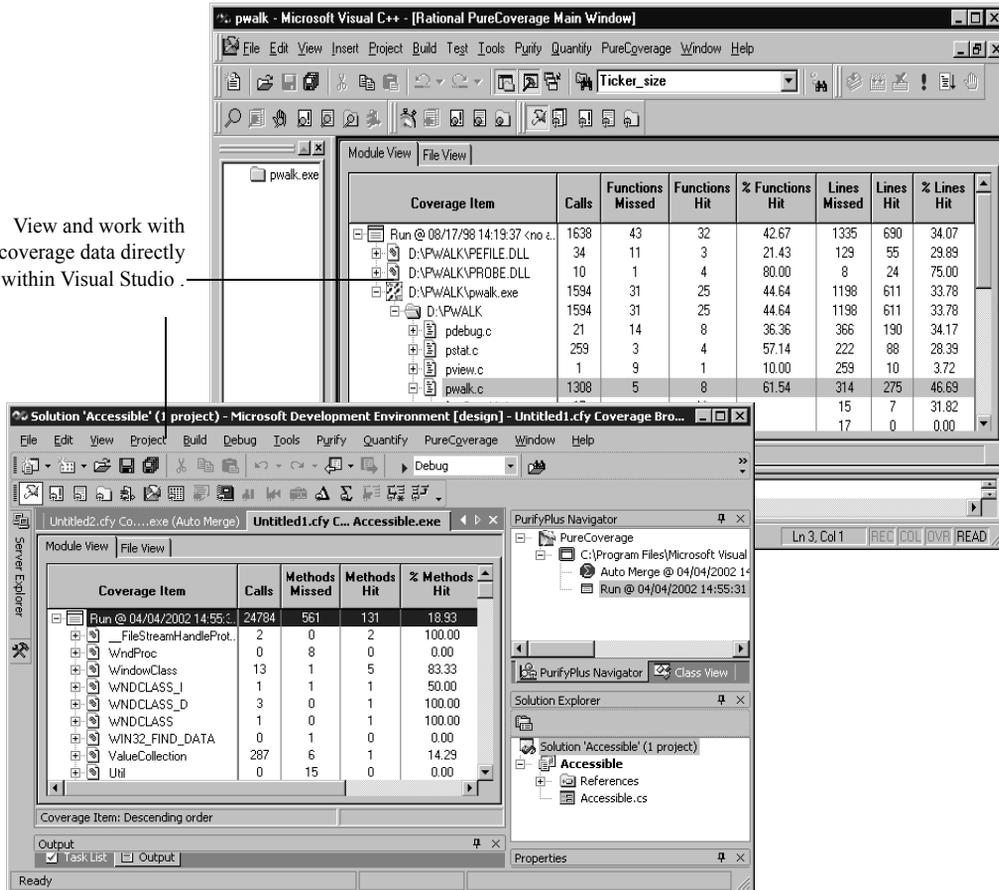
If you have Robot installed, you can run a test script for a program and monitor the program at the same time, without leaving Robot. With ClearQuest, you can submit a coverage defect, and attach a PureCoverage data file (.cfy), as soon as you find untested code, without leaving PureCoverage.

More information? Look up *integrating* in the PureCoverage online Help index.

Fine-tuning data collection

Using the PureCoverage PowerCov™ options, you can fine-tune the level of code coverage reported for any module in your program at any stage of development and testing. You can set default settings that apply to all programs. You can also assign settings that apply only to the current program.

Select **Line** or **Function** as the default coverage level

To customize coverage levels for specific modules, click **Configure**

Then select one or more modules in the Module Coverage dialog . . .

and set a coverage level for them
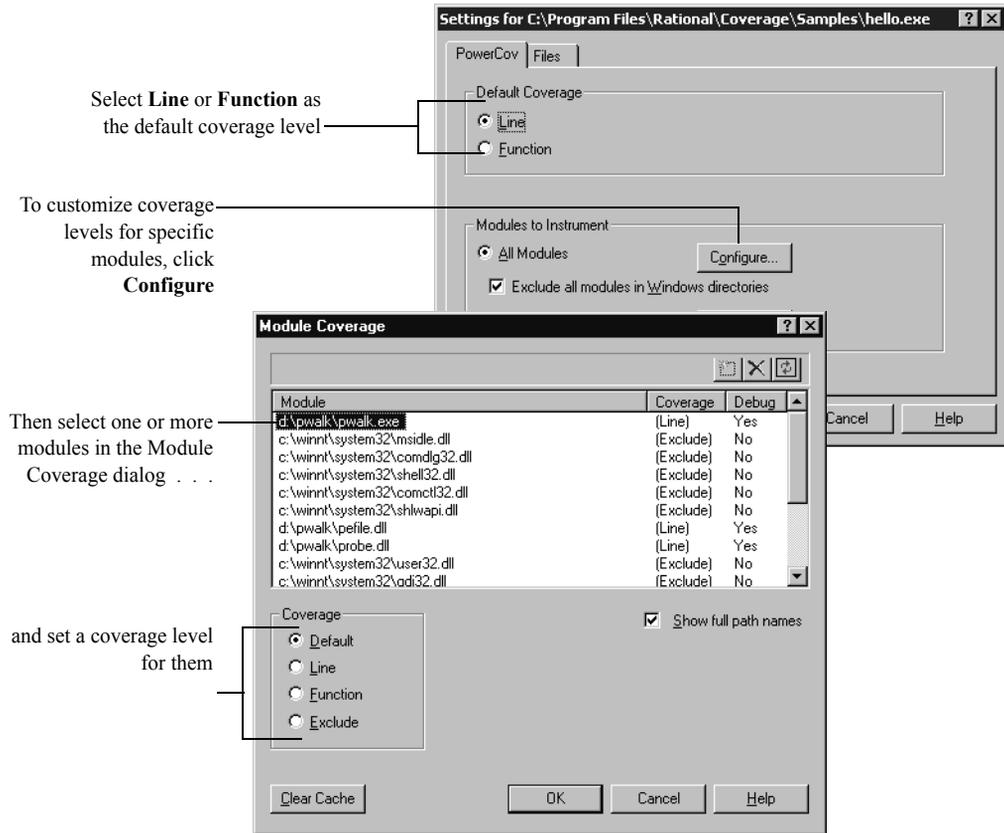
To concentrate on specific modules in your code, use PowerCov options to select **Line** as the coverage level for only those modules. You can improve instrumentation and run-time performance by selecting **Function** as the coverage level for the other modules. Or you can exclude some modules from coverage.

More information? Look up *settings, overview* and *coverage levels, overview* in the PureCoverage online Help index.

Using Selective instrumentation

If you are working in C/C++ or Visual Basic native-compiled code, PureCoverage offers you the option of selecting for instrumentation one or more modules or .dll's, rather than instrumenting all modules. This has the advantage of automatically focusing your coverage data on the code you're most concerned with, and it also saves time when you run your code under PureCoverage.

For example, assume you are working on a plug-in application that is to be loaded by Microsoft Internet Information Server (IIS). You don't need to instrument and profile all of IIS. All you need to do is instrument your plug-in, and then run it as usual under IIS. PureCoverage collects code coverage data as your plug-in runs, and presents this data to you when the plug-in exits.

To instrument your plug-in, select **Settings > Default settings** in PureCoverage to display the Settings dialog, and then in the dialog select **Modules to Instrument: Selected Modules**. Click on **Configure** to open the Module Instrumentation dialog for specifying the name of your plug-in.

Click to add a module to the list

Selected modules must have debug data avaliable; an .exe module also requires relocation data

With transient instrumentation, the instrumented version of the module is kept only for the duration of one program run, and is then replaced by the uninstrumented original

Module Coverage [Selective]

Module	Coverage	Debug	Instrumentation State
c:\hello.exe	(Line)	Yes	Transient

Coverage
- Default
- Line
- Function
- Exclude

Instrumentation State
- Transient
- Permanent
- None

☑ Show full path names

Restore All Modules   OK   Cancel   Help
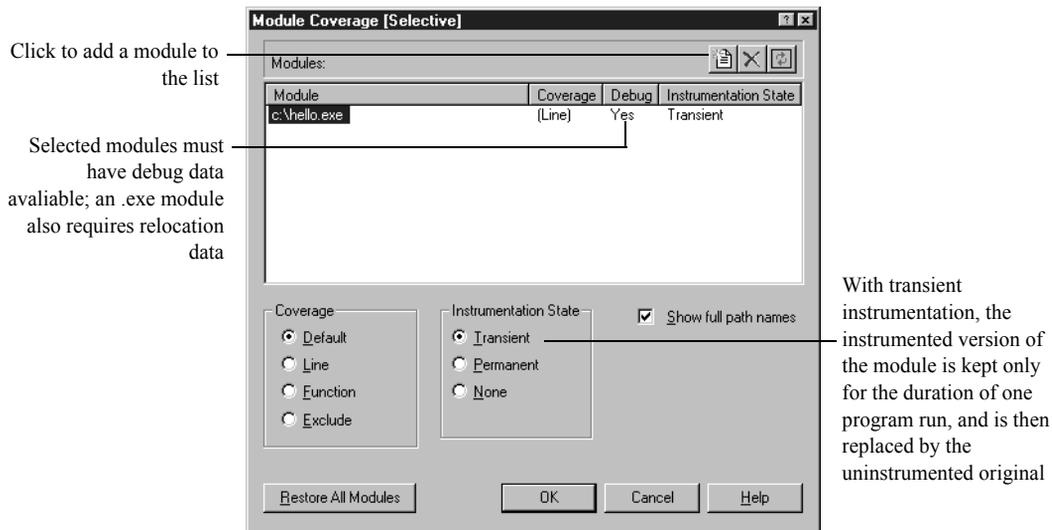
Run your plug-in as usual. PureCoverage collects and displays profiling data.

More information? Look up *selective instrumentation* in the PureCoverage online Help index.

Zeroing in on key program areas

With PureCoverage, you can capture coverage data for your entire program or for any section of it. You can capture coverage information for specific sections using:

▫ Interactive snapshots

▫ PureCoverage API functions

Taking interactive snapshots

Using PureCoverage, you can take snapshots of coverage data for individual routines as you exercise your program.

Start and stop recording Clear data Take a data snapshot

Follow program coverage in the Run Summary window
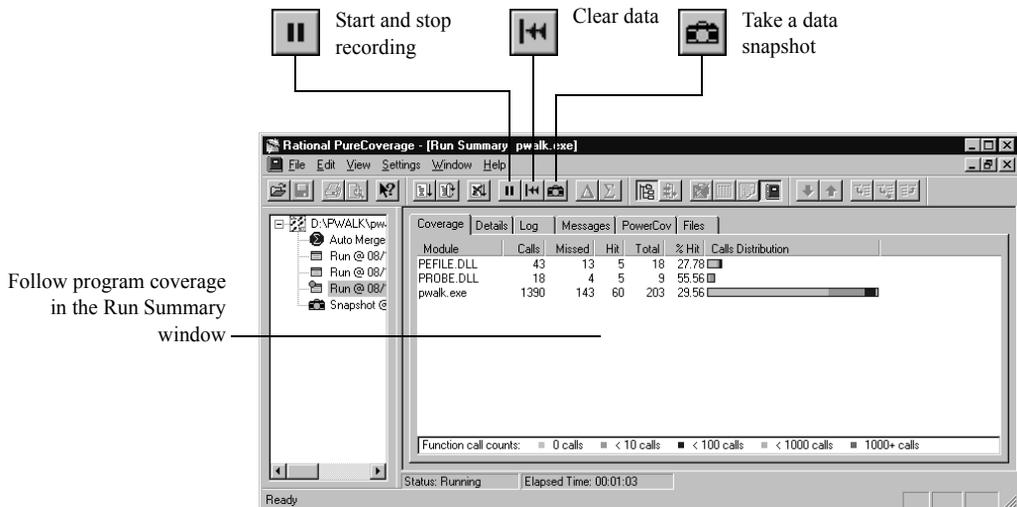
More information? Look up *snapshots* and *run summary window* in the PureCoverage online Help index.

Using PureCoverage API functions

PureCoverage includes a set of Application Programming Interface (API) functions that give you additional control over the collection of coverage data. You can use them to start and stop data collection or to save data at any time during a run, collecting only the coverage data you need to focus on a specific area of your program.

You can call PureCoverage API functions from your program, from the QuickWatch dialog in Visual Studio, or from whatever debugger you're using.

More information? Look up *api functions, using* in the PureCoverage online Help index.

Integrating PureCoverage in your test environment

Integrating PureCoverage with your test environment gives you a powerful tool for continuous coverage monitoring. For example, you can easily run PureCoverage from an existing makefile, batch file, or Perl script by adding the command:

```
Coverage /SaveTextData Exename.exe
```

to run your program under PureCoverage. The `/SaveTextData` option generates coverage data in text-file format, without the graphical interface. You can incorporate the information from this file into your test results report.

PureCoverage can also merge coverage data from multiple runs. Say you're running a series of automated tests on a program, each time using a different set of data. You can modify the script to merge the coverage data into a single file. Add the following line to the beginning of your test script:

```
del Exename_AutoMerge.cfy
```

to delete any existing Auto Merge files.

Then, each time you run your program, substitute the following for the run command:

```
Coverage /SaveMergeData /SaveMergeTextData Exename.exe
```

This command merges the coverage data from all runs of the program and saves it to a PureCoverage data file, `Exename_AutoMerge.cfy`, and to an ASCII text file, `Exename_AutoMerge.txt`.

Java, .NET managed code, and Visual Basic programmers: For Java code, the command line must include the /Java switch. For .NET managed code and and Visual Basic p-code programs, the command line must include the /Net switch. For example, if you have a test script that runs a Java class file, change the line that runs it to:

```
Coverage /SaveData /Java Java.exe Classname.class
```

For managed code and p-code programs, the command is:

```
Coverage /SaveData /Net Exename.exe
```

More information? For details, and additional command-line options, look up *command line* and *scripts* in the PureCoverage online Help index.

If you have Rational Robot installed, you can run a test script for a program and monitor the program at the same time, without leaving Robot.

More information? Look up *robot* in the PureCoverage online Help index.

Now you're ready to put PureCoverage to work. Remember that the online Help contains detailed information to assist you.

Getting Started: Quantify

What it does

Your customers want the fastest possible software. They want your program to work instantaneously and make the most of their computing resources. Inferior performance reduces their satisfaction with the features you worked so hard to include.

So what can you do about it?

The practical solution is to identify bottlenecks, and then to reduce or eliminate them, through systematic performance engineering. Begin monitoring performance just as soon as you have a program that runs, when it's easiest and most economical to make structural changes. Continue working on performance until you're ready to ship. Weigh the cost of implementing each improvement against the benefits you expect from it.

How can you get the data you need for performance engineering?

Quantify® puts successful performance engineering within your grasp. It collects complete, accurate performance data and displays it in easy-to-understand graphs and tables, so that you can see exactly where your code is least efficient. Using Quantify, you can make virtually any program run faster, and you can measure the results.

Quantify profiles performance for code written in all commonly used programming languages:

- C/C++ code in .exe's, .dll's, OLE/ActiveX controls, and COM objects

- p-code .exe's, native-code .exe's, .dll's, OLE/ActiveX controls, and COM objects

- .NET managed code assemblies, .exe's, .dll's, OLE/ActiveX controls, and COM objects

- Components launched from container programs such as Microsoft Internet Explorer, the Microsoft Transaction Server, Tstcon32.exe, Firefox, or any Microsoft Office application

- Microsoft Excel and Microsoft Word plug-ins

Quantify can profile all components of your code, whether you have source code or not. For native-code applications written in C/C++ and Visual Basic, Quantify also allows you to select exactly which modules you want to profile.

Quantify automatically integrates with Microsoft Visual Studio, so you can use Quantify without changing the way you work if you're developing code in this environment.

This chapter shows you how to use Quantify to find performance bottlenecks, and introduces the features that make Quantify a powerful, flexible performance engineering tool. As you read this chapter, keep in mind that any discussion that applies to functions and modules also applies to Visual Basic procedures and object libraries.

The basic steps

Quantify provides a complete, accurate set of performance data for your program and its components, and shows you exactly where your program spends most of its time.

To improve a program's performance:

1   Run the program with Quantify to collect performance data.

2   Use the Quantify data windows to analyze the performance data and find bottlenecks.

3   Modify your code to reduce or eliminate bottlenecks.

4   Rerun the program and use the Compare Runs tool to verify performance improvements.

This chapter describes how to use Quantify as a standalone desktop application. The same principles apply when you use Quantify integrated with Microsoft Visual Studio, or when you incorporate Quantify into your test harness. For more information, read *Integrating Quantify with your development desktop* on page 68 and *Integrating Quantify in your test environment* on page 75.

Running a program

To collect performance data for a program, launch Quantify from the Windows Start menu and click **Run** in the Quantify welcome screen to open the Run program dialog.

Click the browse button to select the program you want to profile . . .
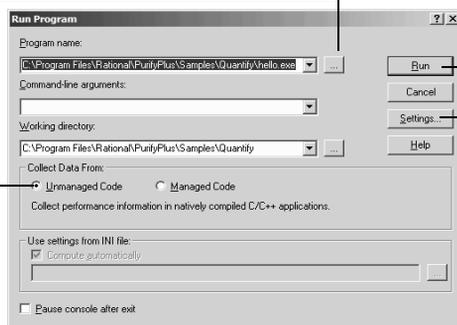
and then click **Run** to run the program

and select the type of code

You can click **Settings** to specify data-collection instructions for special situations, such as selective instrumentation of specific modules

Quantify profiles performance for functions and, if debug line information is available, for individual lines as well. If you want line-level data for programs built in release mode, you must supply debug line information. For specific instructions, look up *debug data* in the Quantify online Help index.

Quantify displays a Run Summary window as the program runs, showing the current status of all program threads.

Click to pause and resume profiling in order to focus on specific routines
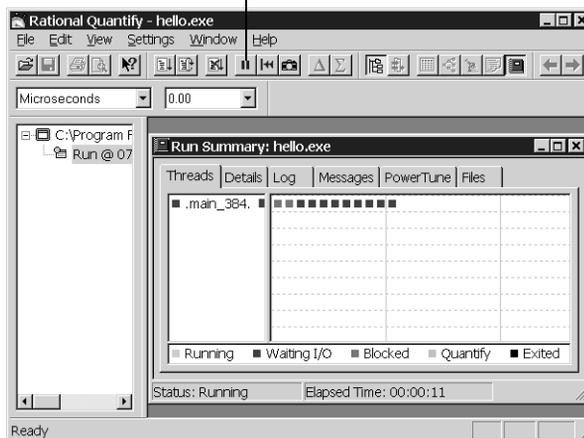
Quantify saves all instrumented components. When you rerun a program, Quantify saves time by using these instrumented components, reinstrumenting only the ones that have changed since the previous run.

When you exit your program, Quantify has an accurate profile of its performance.

More information? Look up *profiling*, *selective instrumentation*, *run summary* and *recording data* in the Quantify online Help index.

Analyze the performance data

The second step in improving your program's performance is to analyze the performance data that Quantify has collected.

Using the Quantify Call Graph window

When you exit your program, Quantify displays the Call Graph window. The window's initial display focuses on the heavy-duty components of your code, the areas where any performance improvement would have the greatest impact.

The Quantify Call Graph initially displays the 20 most expensive functions in a program

A root node, representing the total time for the run, brings the number of visible nodes to 21
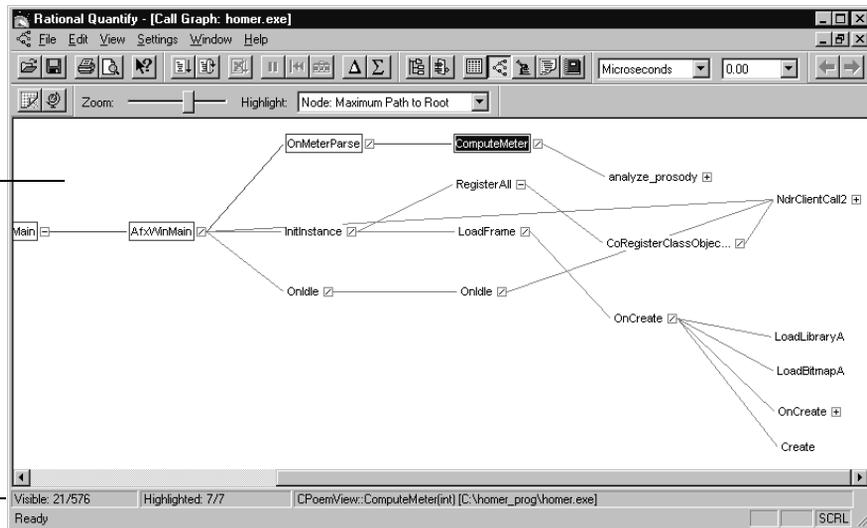
The call graph initially highlights the most expensive path. You can choose instead to highlight functions based on various criteria, including performance, calling relationships, and possible causes for bottlenecks. You can also show additional functions, hide functions, and grab and move functions to see calling relationships more clearly.

Use the call graph to find functions that are taking more time than you think they should.

For example, the programmer who wrote this code knows that the ComputeMeter function should be so fast that it wouldn't show up in the initial call graph display at all.

Having located a suspicious function, you can isolate it to examine where it spends its time.

The Subtree commands adjust the focus of the dataset
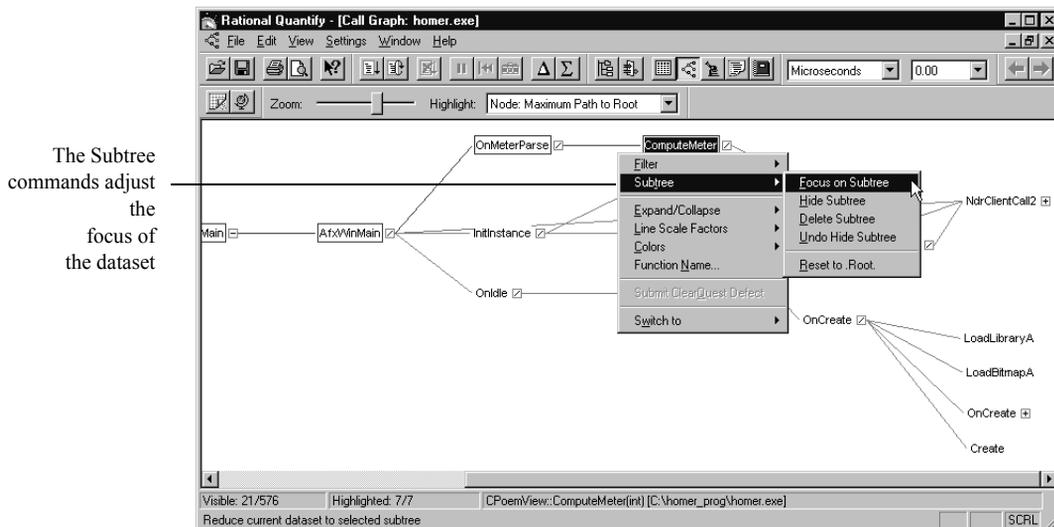
Quantify adjusts the dataset so it contains only ComputeMeter and its descendants. You can now can expand the ComputeMeter subtree to see what's going on downstream.

The Expand and Collapse commands help you explore a program's structure
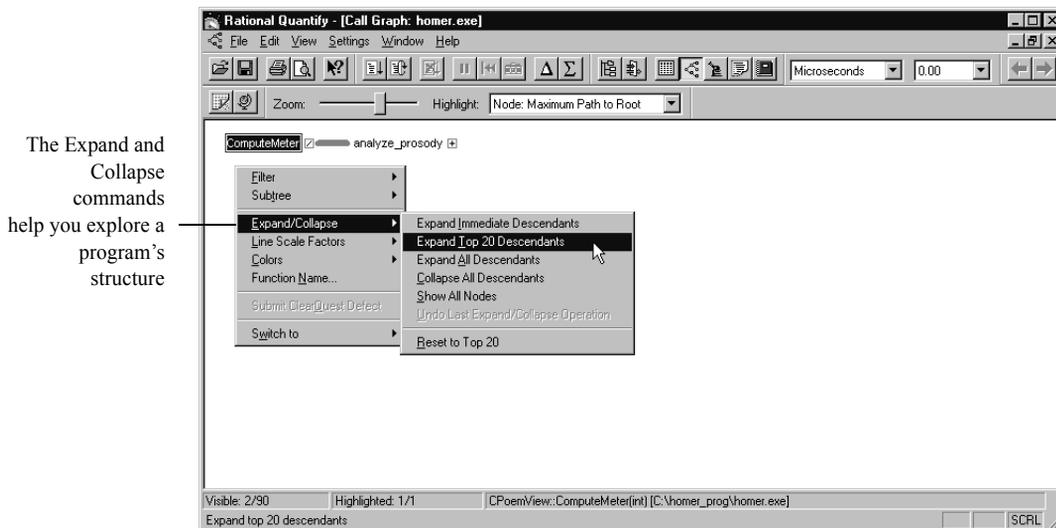
The most expensive paths in the `ComputeMeter` subtree lead to the `SetWindowTextA` function.

You can judge the relative expense of paths by the thickness of the lines
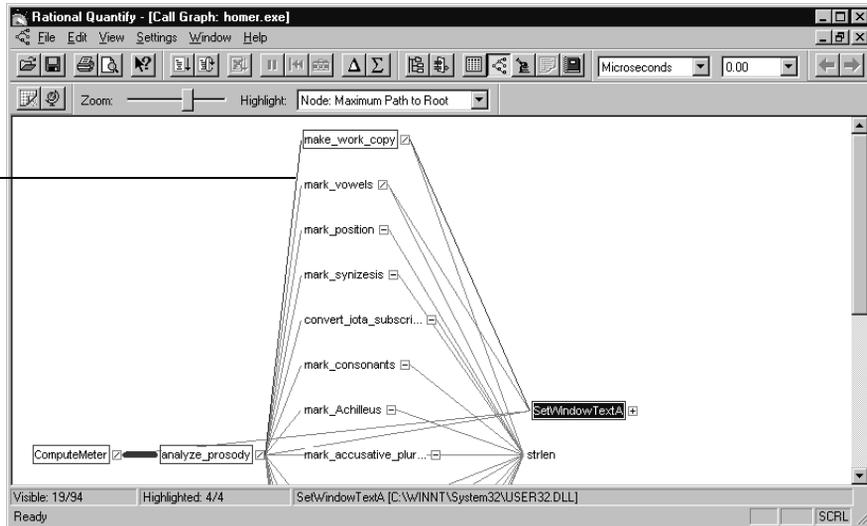
The programmer who wrote this code intended this function to provide feedback when he was developing his algorithm, and not to be part of the released application. Removing the function will significantly improve performance.

Using the Function List window to analyze numerical data

Quantify starts by orienting you in your program with the call graph, and then provides additional ways to zero in on problems. You can use the Function List window to display and sort numerical performance data.

Click the Function List tool to display numerical data

In this example, the Function List window shows exactly how much time the obsolete calls to SetWindowTextA are costing. The data displayed is all the data for the SetWindowTextA subtree.

F+D time includes the time the program spends in the function and in all its descendants

This is one of the most expensive functions in terms of F+D time

Function	Calls	Function time	F+D time	F time (% of Focus)	F+D time (% of Focus)	Avg F time
ComputeMeter	1	0.94	607,924.64	0.00	100.00	0.94
analyze_prosody	1	82.34	599,951.65	0.01	98.69	82.34
SetWindowTextA	2,929	210,938.51	296,491.81	34.70	48.77	72.02
strlen	40,554	288,857.84	288,857.84	47.52	47.52	7.12
make_work_copy	1	220.01	173,630.44	0.04	28.56	220.01
mark_vowels	1	127.89	165,474.34	0.02	27.22	127.89
mark_position	1	735.56	95,641.80	0.12	15.73	735.56
mark_synizesis	1	72.84	12,088.67	0.01	1.99	72.84
convert_iota_subscripts	1	263.46	10,745.26	0.04	1.77	263.46
mark_consonants	1	230.33	10,399.87	0.04	1.71	230.33
mark_Achilleus	1	57.78	10,259.37	0.01	1.69	57.78
mark_accusative_plurals	1	71.91	10,243.30	0.01	1.68	71.91
mark_finalvowels	1	141.18	10,209.56	0.02	1.68	141.18
mark_a_e	1	71.07	10,192.27	0.01	1.68	71.07
find_dipthongs	1	86.30	10,162.29	0.01	1.67	86.30

Function List: homer.exe

Visible: 94/94   SetWindowTextA

Consider the percentages: SetWindowTextA takes up almost 50% of the subtree's total time. Since this function serves no purpose in the current version of the program, this is a clear example of unnecessary processing, one of the most common causes of performance bottlenecks.

Doing interactive 'what-ifs'

In addition to analyzing your program's current performance, you can use Quantify to project performance improvements.

In this example, you could right-click SetWindowTextA in the Call Graph and then delete the SetWindowTextA subtree. Quantify discards the subtree's time from the displayed dataset and recomputes the remaining data so that you can see exactly how the program will perform without the subtree.

Function	Calls	Function time	F+D time	F time (% of Focus)	F+D time (% of Focus)	Avg F time
ComputeMeter	1	0.94	311,432.84	0.00	100.00	0.94
analyze_prosody	1	82.34	303,945.32	0.03	97.60	82.34
strlen	40,554	288,857.84	288,857.84	92.75	92.75	7.12
mark_position	1	735.56	95,641.80	0.24	30.71	735.56
make_work_copy	1	220.01	27,748.88	0.07	8.91	220.01
mark_vowels	1	127.89	16,443.10	0.04	5.28	127.89
mark_synizesis	1	72.84	12,088.67	0.02	3.88	72.84
convert_iota_subscripts	1	263.46	10,745.26	0.08	3.45	263.46
mark_consonants	1	230.33	10,399.87	0.07	3.34	230.33
mark_Achilleus	1	57.78	10,259.37	0.02	3.29	57.78
mark_accusative_plurals	1	71.91	10,243.30	0.02	3.29	71.91
mark_finalvowels	1	141.18	10,209.56	0.05	3.28	141.18
mark_a_e	1	71.07	10,192.27	0.02	3.27	71.07
find_dipthongs	1	86.30	10,162.29	0.03	3.26	86.30
mark_dubiousvowels	1	63.20	10,106.56	0.02	3.25	63.20

Visible: 93/93   CPoemView::ComputeMeter(int)

The time for the ComputeMeter subtree, which took over 600,000 microseconds before the change, is now just over 311,000 microseconds

Using the Function Detail window

The Function Detail window lets you display performance data from the point of view of an individual function

Click the Function Detail tool for data about a specific function

The strlen function in this example has shown up both in the function list and the call graph. The function list shows this run of the program called it over 40,000 times. Referring to the call graph, you can see that all the expensive functions in this part of the program call strlen.
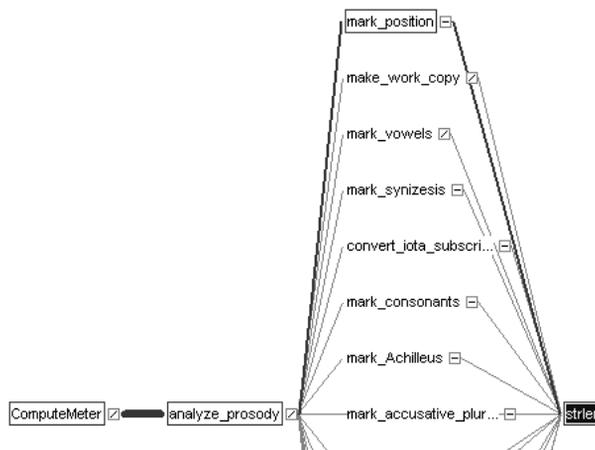
This part of the code manipulates lines of text as strings. These functions apply a collection of complex rules in sequence to each line in order to identify patterns. But calling `strlen` so many times suggests that there is a performance issue.

Click a to sort the list

strlen is the most expensive single function in the subtree

Function List: homer.exe						
Function	**Calls**	**Function time**	**F+D time**	**F time (% of Focus)**	**F+D time (% of Focus)**	**Avg F time**
strlen	40,554	288,857.84	288,857.84	92.75	92.75	7.12
tolower	1,463	1,161.33	1,161.33	0.37	0.37	0.79
mark_position	1	735.56	95,641.80	0.24	30.71	735.56
ExtTextOutA	2	507.00	562.46	0.16	0.18	253.50

By itself, `strlen` uses around 92% of the total subtree time, now that `SetWindowTextA` has been discarded from the dataset.

Opening the Function Detail window gives you a different angle on the data for `strlen`: specific information, in numerical and graphical format, about calls to it from other functions.

Detailed data for a function

Function Detail: homer.exe	
Function:	strlen
Calls:	40,554
Function time:	288,857.84 usec (92.75% of Focus)
F+D time:	288,857.84 usec (92.75% of Focus)
Avg F time:	7.12 usec
Min F time:	5.51 usec
Max F time:	1,899.19 usec
Module:	C:\WINNT\System32\MSVCRTD.DLL
Source File:	(None)
Measurement:	Timed
Hidden functions:	(None)

Double-click a slice in the Callers or Descendants pie chart to display data for that function

Callers

Caller	**Percent**	**Calls**	**Propagated time**
make_work_copy	7.19	2,928	20,763.99
mark_synizesis	4.16	1,464	12,015.83
mark_vowels	3.72	1,464	10,739.88
convert_iota_sub...	3.63	1,464	10,481.80
mark_Achilleus	3.53	1,464	10,201.59

Data about the calls made to a function (by callers). . .

Descendants

Descendant	**Percent**	**Calls**	**Propagated time**

and by a function (to descendants)

Callers: 17   Descendants: 0   strlen

As you examine this data, you might observe that most of the functions make about the same number of calls to `strlen`. To see exactly what is going on, you can look at your source code,

Using the Annotated Source window

The Annotated Source window shows
your code, annotated with line-by-line
performance data. Here is the code for
mark_consonants, one of the functions
that call strlen.

Click the Annotated Source
tool to relate performance data
to the source code

Line time shows
the time spent
in each line

L+D time shows
the time spent in the
line and the
functions it calls
(its descendants)

This line spends
most of its time
in descendant
functions

```
C:\homer_prog\prosody.cpp                                              _ □ ×

Functions:  mark_consonants(void)              ▼

 Line    L+D     Percent of    Percent of    Line
 time    time   Function time    F+D time   number                                    Sour

                                                   *******************************************
                                                   Function:                    mark_consonan
                                                   Called:                      1
                                                   Function time:               230.33 usec
                                                   F+D time:                    10,399.87 use
                                                   Distribution to Callers:
                                                      Called 1 times.  analyze_prosody(char *)
                                                   *******************************************
  0.01    0.01      0.01         0.00       200  {
                                             201     int i;
 37.86 10,207.40   16.44        98.15       202     for (i=0;i<SPECIMEN_LENGTH(specimen);i++)
                                             203     if (specimen.work_copy[i]=='c'||
                                             204         specimen.work_copy[i]=='d'||
                                             205         specimen.work_copy[i]=='f'||
                                             206         specimen.work_copy[i]=='g'||
                                             207         specimen.work_copy[i]=='j'||

Line: 202 of 1045      Microseconds    Function
```

Look at the data for the for statement. Its line + descendants time is much
greater than its line time alone, which means the line calls other functions
heavily. The only part of the line that could possibly represent a function call is
i<SPECIMEN_LENGTH(specimen), and SPECIMEN_LENGTH is in fact
defined in this program as a call to strlen. In effect, the program is calling
strlen every time it traverses one of these loops. And it's the same for all the
other parallel functions.

This wastes computing time, since all the program needs to do is call strlen
once for each string, then cache the value. This is a case of unnecessary
recomputation, another common cause of a performance bottleneck.

Compare the modified program's performance

The final step in improving your program's performance is to eliminate the
bottlenecks you've found with Quantify and to compare performance data
from two runs, to verify that your modifications have helped.

Assume now that you've eliminated all the
strlen calls, and run the program again.
Compare the first run to the new run, to see
how the performance has improved.

Click the Compare Runs
tool to see improvements

The Diff call graph highlights in green paths and functions whose performance is improved
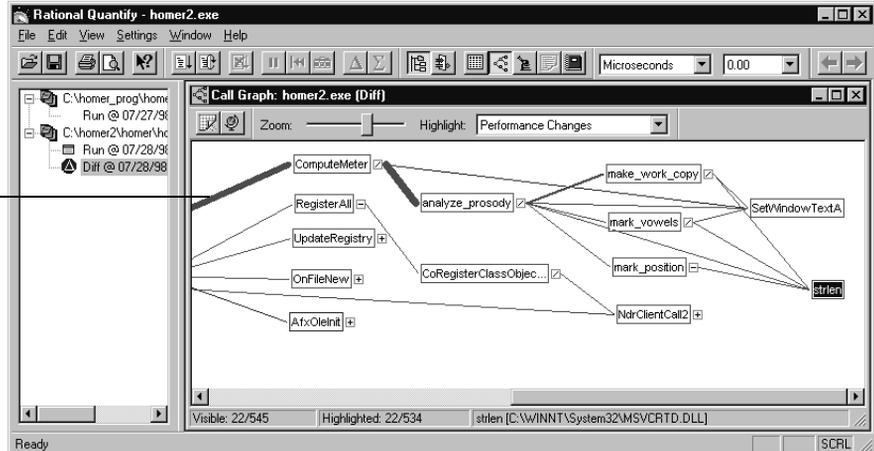
The Diff call graph highlights `ComputeMeter`, `strlen`, and `SetWindowTextA` in green, meaning their performance is improved.

Open the Diff function list to get the numerical comparison.

The Diff function list shows performance improvements as negative values

The total time for `ComputeMeter` is now around 12,000 microseconds, an improvement of more than 595,000 microseconds

Function	F+D time (Diff)	F+D time (New)	F+D time (Base)	Function time (Diff)	F time (Base)	F time (New)	Calls (Diff)
analyze_prosody	-596,939.23	3,012.42	599,951.65	-12.14	82.34	70.20	0
OnMeterParse	-595,664.54	12,673.44	608,337.98	0.00	0.12	0.12	0
ComputeMeter	-595,661.23	12,263.41	607,924.64	-0.01	0.94	0.93	0
SetWindowTextA	-306,965.79	326.41	307,292.20	-7,457.42	7,461.99	4.57	-2,927
strlen	-288,850.26	7.58	288,857.84	-288,850.26	288,857.84	7.58	-40,553
make_work_copy	-172,845.47	784.97	173,630.44	-42.95	220.01	177.06	0
mark_vowels	-165,339.74	134.59	165,474.34	-30.81	127.89	97.08	0
OnIdle	-105,621.61	601,279.77	706,901.38	261.38	78,222.31	78,483.70	25,586
OnIdle	-96,932.31	616,172.30	713,104.61	2,095.13	259.52	2,354.65	25,586
mark_position	-95,016.94	624.86	95,641.80	-110.70	735.56	624.86	0
InitInstance	-41,186.37	722,108.38	763,294.75	0.00	0.70	0.70	0
UpdateRegistry	-34,700.28	84,866.06	119,566.34	-16.44	600.86	584.41	0
ExtractIconA	-29,318.56	3,014.65	32,333.21	3.33	102.33	105.66	0
PrivateExtractIconsW	-28,728.05	30,655.72	59,383.77	824.09	6,646.13	7,470.22	0
AssertValid	-25,272.32	126,539.00	151,811.32	-47.17	268.25	221.08	-1,216

You can save datasets as a Quantify data file (.qfy) to use for further analysis or to share with other Quantify users. You can save data to a tab-delimited ASCII text file (.txt) to use outside of Quantify, for example, in test scripts or in Microsoft Excel. You can also copy data directly from the Function List window to use in Excel.

Advanced features

Quantify provides powerful features that help you make maximum use of the performance data. For example, you can:

- Integrate Quantify with your development desktop
- Select specific modules for instrumentation and profiling
- Control data recording interactively
- Highlight functions that share key attributes
- Focus on critical data
- Fine tune data collection
- Integrate Quantify in your test environment

This section gets you started using these features to profile the important parts of your code more efficiently, and to zero in on bottlenecks.

Integrating Quantify with your development desktop

Quantify's integration—for example, with Microsoft Visual Studio, Rational Robot, and Rational ClearQuest™—puts powerful performance profiling within easy reach while you develop your code using your favorite tools.

During installation, a Quantify menu is automatically added to Visual Studio so you can profile your code at any time during development, without leaving your development environment.

Select the PurifyPlus > Quantify menu, then run your program

View and work with performance data directly within Visual Studio
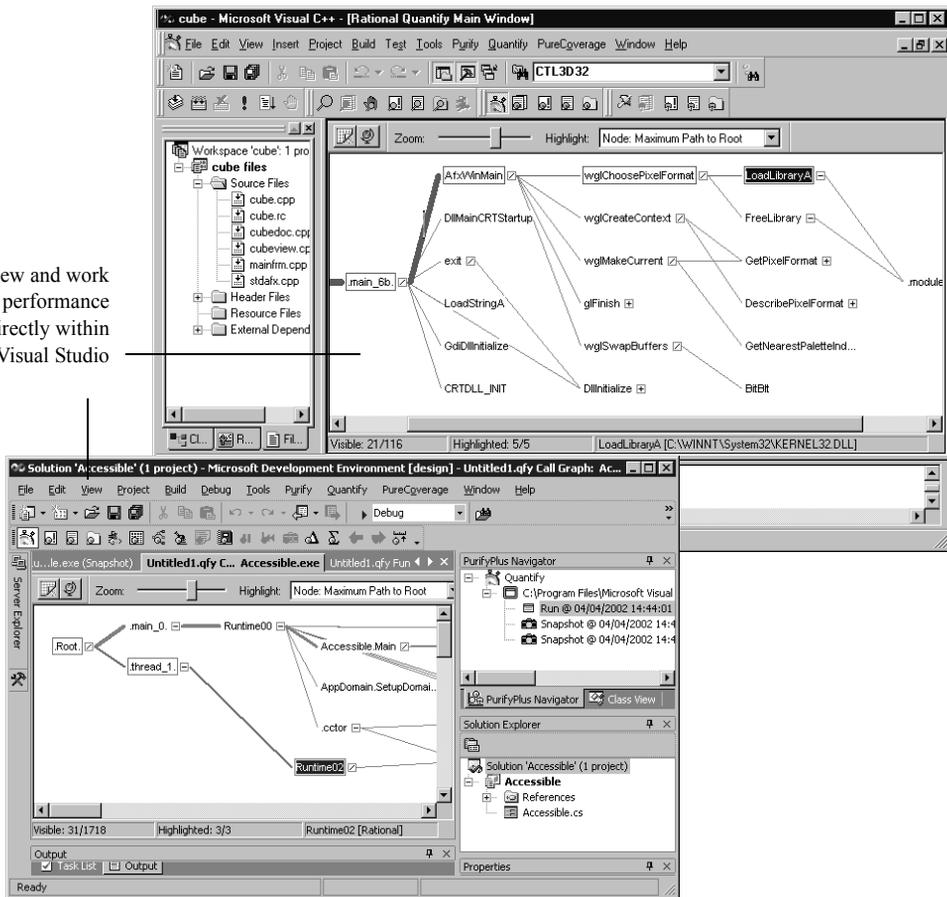
If you have Rational Robot installed, you can run a test script for a program and profile the program at the same time, without leaving Robot. With Rational ClearQuest, you can submit a performance defect, and attach a Quantify data file (.qfy), as soon as you find slow code, without leaving Quantify.

More information? Look up *integrating* in the Quantify online Help index.

Using selective instrumentation

If you are working in C/C++ or Visual Basic native-compiled code, Quantify offers you the option of selecting for instrumentation one or more modules or .dll's, rather than instrumenting all modules. This has the advantage of automatically focusing your profiling data on the code you're most concerned with, and it also saves time when you run your code under Quantify.

For example, assume you are working on a plug-in application that is to be loaded by Microsoft Internet Information Server (IIS). You don't need to instrument and profile all of IIS. All you need to do is instrument your plug-in, and then run it as usual under IIS. Quantify collects performance data as your plug-in runs, and presents this data to you when the plug-in exits.

To instrument your plug-in, select **Settings > Default settings** in Quantify to display the Settings dialog, and then in the dialog select **Modules to Instrument: Selected Modules**. Click on **Configure** to open the Module Instrumentation dialog for specifying the name of your plug-in.

Click to add a module to the list

Selected modules must have debug data avaliable; an .exe module also requires relocation data

With transient instrumentation, the instrumented version of the module is kept only for the duration of one program run, and is then replaced by the uninstrumented original

Module Instrumentation [Selective]

Modules:

Module	Measurement	Debug	Instrumentation State
c:\hello.exe	(Line)	Yes	Transient

Measurement
- ● Default
- ○ Line
- ○ Function
- ○ Time

Instrumentation State
- ● Transient
- ○ Permanent
- ○ None

☑ Show full path names

Restore All Modules   OK   Cancel   Help

Run your plug-in as usual. Quantify collects and displays profiling data.

More information? Look up *selective instrumentation* in the Quantify online Help index.

Controlling data recording interactively

As your program runs, you can monitor the performance of threads and fibers and view general information about the run using the Run Summary window.
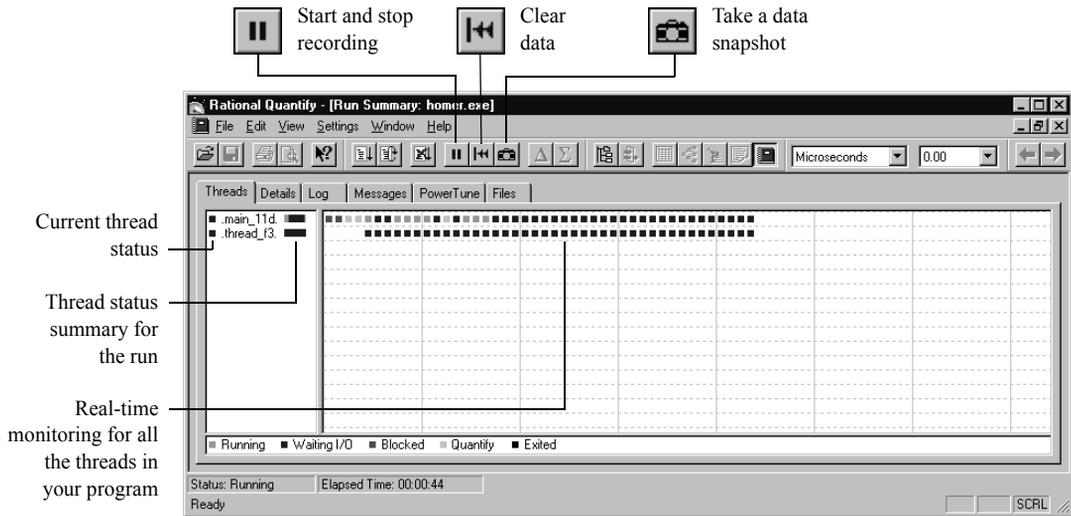
You can use the data recording tools to collect data for the entire program or for just a section of it, so you get exactly the performance data you want. For example, at any time you can stop recording, clear the data collected to that point, and then resume recording. You can also take a snapshot of the current data, enabling you to examine performance in stages.

You can also start and stop recording, clear data, and take snapshots automatically from within your program by incorporating Quantify's data recording API functions in your code.

More information? Look up *threads*, *recording data*, and *API functions* in the Quantify online Help index.

Highlighting functions that share key attributes

You can highlight functions in the call graph to display specific performance characteristics or to show calling relationships.

Click to display the Highlight list

Select **Functions with Source**, for example, to highlight functions that have annotated source
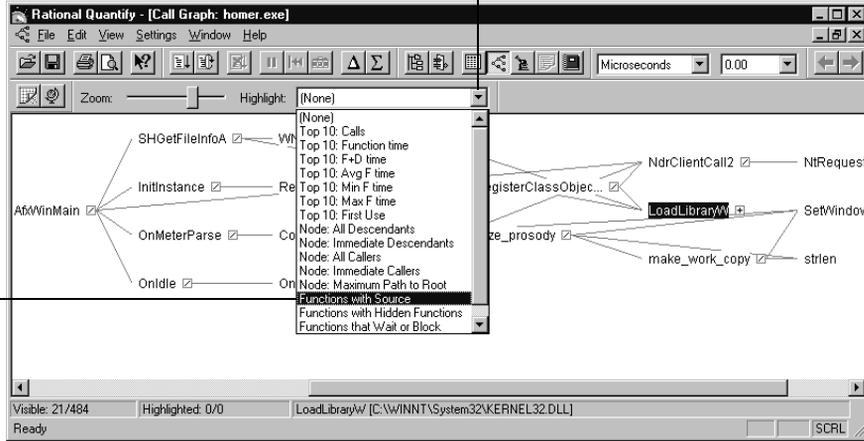
Functions with source code available are enclosed in rectangles

21 of the 484 functions in the current dataset are displayed in the Call Graph

8 of the 86 functions with source code are displayed in the Call Graph
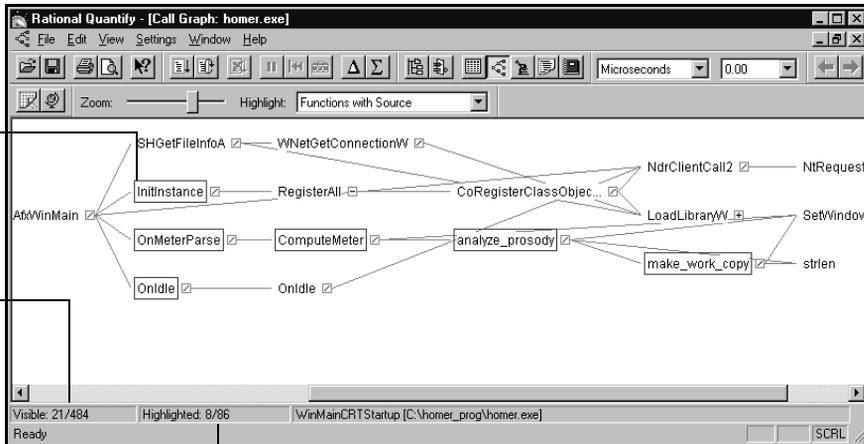
More information? Look up *highlighting* in the Quantify online Help index.

Focusing your data

Use Quantify's *filter* commands to remove a selected function, or all functions in a module, from the current dataset. Alternatively, use *subtree* commands to focus on or remove a specific function and all its descendants from the current dataset. Simply right-click a function in the Call Graph, Function List, or Function Detail window.

You can hide or delete individual functions, all functions in a module, or entire subtrees. Hide functions or subtrees to roll up their time to their callers; delete them to discard their time completely

The Filter Manager offers additional filtering options

Select **Focus on Subtree** to delete all functions except the subtree
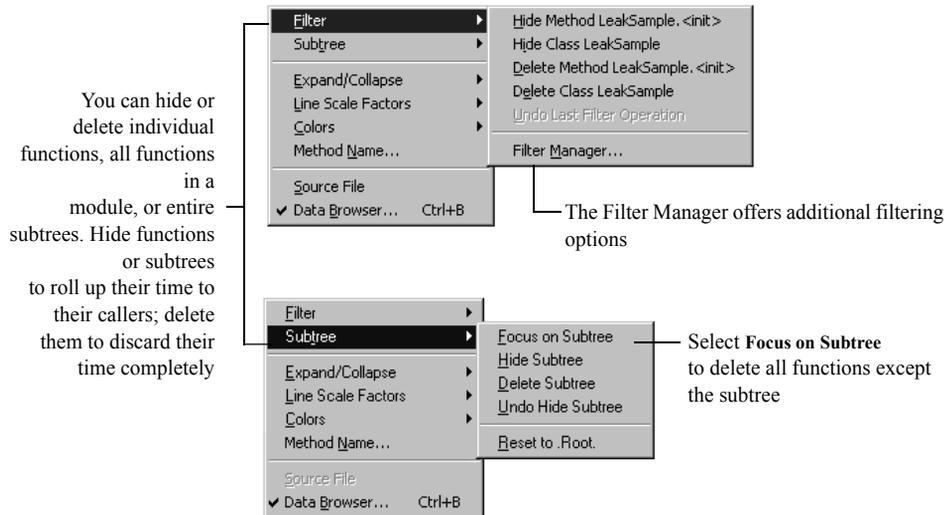
Quantify has undo capabilities for all filter and subtree commands, to easily return to any previous dataset configuration.

The Call Graph window also provides a series of expand and collapse commands that work with subtrees. Unlike the filter and subtree commands, however, these commands affect only the Call Graph display; they do not change the current dataset.

More information? Look up *filtering data* and *subtrees* in the Quantify online Help index.

Fine-tuning data collection

Using the Quantify PowerTune options, you can specify how you want Quantify to measure your program's performance. Quantify's default measurement levels are based on what is appropriate in most situations, but with PowerTune you can control how specific modules are measured.

Why is this useful? It allows you to *significantly* speed up the run-time performance during profiling. You can, for example, select **Time** as your default measurement level, and then select **Line** for the specific modules that you're currently investigating.

Select a default measurement level

Select one or more modules. . .

then set their measurement level

Click **Configure** to display the Module Instrumentation dialog and specify measurement levels for specific modules
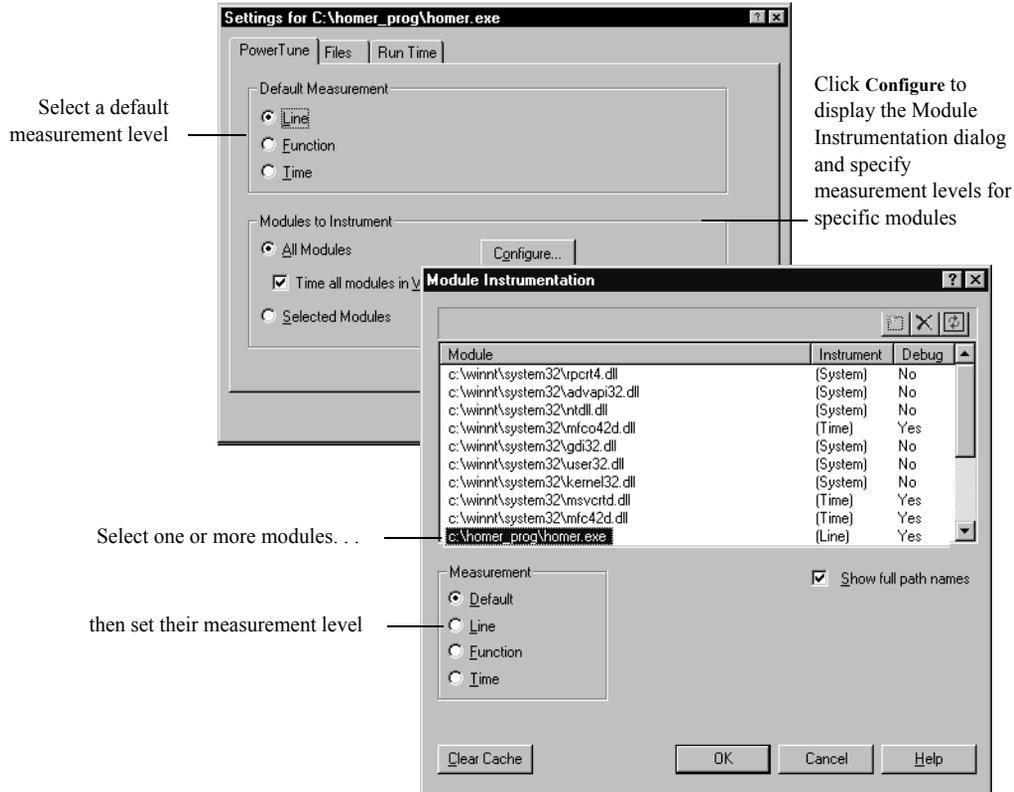
Quantify measures performance at several levels of detail:

- **Line**. At this level, Quantify counts the number of times each line executes during a run, then computes performance data based on the number of cycles needed for one execution. Line level, which requires debug line information, results in the most accurate and detailed data possible, but does take the most time to collect.

- **Function**. This level provides the same level of accuracy as line-level measurement, but less detail. Function level is useful when you don't need to know how individual lines perform, but still want precise, repeatable data for functions.

□ **Time**. Quantify collects data for timed functions by starting and stopping a timer when each function begins and ends. The data is accurate for the current run, but is influenced by microprocessor state and memory effects. The overhead for collecting timed data, however, is very low.

More information? Look up *measurement types* in the Quantify online Help index.

Integrating Quantify in your test environment

By integrating Quantify into your test environment, you have a tool that detects changes in performance in your nightly tests, giving you an immediate heads-up as soon as things start to go wrong.

You can easily run Quantify from an existing makefile, batch file, or Perl script by adding the command:

```
Quantify /SaveData Exename.exe
```

to run your program under Quantify. The `/SaveData` option generates performance data in a format for viewing and comparing with previous runs of the program in the Quantify graphical interface.

Note that the `/SelectModuleList` option is also available to help focus your testing. Refer to *Using selective instrumentation* on page 70.

NET managed code, and Visual Basic programmers: For managed code and Visual Basic p-code programs, the command line must include the /Net switch. For example, if you have a test script that runs managed code, change the line that runs it to:

```
Quantify /SaveData /Net Exename.exe
```

More information? For details, and additional command-line options, look up *command line* and *scripts* in the Quantify online Help index.

Now try out Quantify on your own code. Remember that Quantify's online Help contains detailed information to assist you.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

```
IBM Director of Licensing
IBM Corporation, North Castle Drive
Armonk, NY 10504-1785
U.S.A.
```

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

```
IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan
```

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION 'AS IS' WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

```
IBM Corporation
Department BCFB
20 Maguire Road
Lexington, MA 02421
U.S.A.
```

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming

interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBMís application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(c) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (c) Copyright IBM Corp. _enter the year or years_. All rights reserved.

Additional legal notices are described in the legal_information.html file that is included in your Rational software installation.

Trademarks

AIX, ClearCase, ClearCase Attache, ClearCase MultiSite, ClearDDTS, ClearGuide, ClearQuest, DB2, DB2 Universal Database, DDTS, Domino, IBM, Lotus Notes, MVS, Notes, OS/390, Passport Advantage, ProjectConsole, PureCoverage, Purify, PurifyPlus, Quantify, Rational, Rational Rose, Rational Suite, Rational Unified Process, RequisitePro, RUP, S/390, SoDA, SP1, SP2, Team Unifying Platform, WebSphere, XDE, and z/OS are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

Index