Rational Software Corporation®

Rational® PurifyPlus
Rational® Purify®
Rational® PureCoverage®
Rational® Quantify®

Getting Started

VERSION: 2002.05.20

PART NUMBER: 800-025734-000

WINDOWS

Rational®
the software development company

support@rational.com
http://www.rational.com

IMPORTANT NOTICE

COPYRIGHT

Copyright ©2001, 2002, Rational Software Corporation. All rights reserved.

Part Number: 800-025734-000

Version Number: 2002.05.20

PERMITTED USAGE

THIS DOCUMENT CONTAINS PROPRIETARY INFORMATION WHICH IS THE PROPERTY OF RATIONAL SOFTWARE CORPORATION (“RATIONAL”) AND IS FURNISHED FOR THE SOLE PURPOSE OF THE OPERATION AND THE MAINTENANCE OF PRODUCTS OF RATIONAL. NO PART OF THIS PUBLICATION IS TO BE USED FOR ANY OTHER PURPOSE, AND IS NOT TO BE REPRODUCED, COPIED, ADAPTED, DISCLOSED, DISTRIBUTED, TRANSMITTED, STORED IN A RETRIEVAL SYSTEM OR TRANSLATED INTO ANY HUMAN OR COMPUTER LANGUAGE, IN ANY FORM, BY ANY MEANS, IN WHOLE OR IN PART, WITHOUT THE PRIOR EXPRESS WRITTEN CONSENT OF RATIONAL.

TRADEMARKS

Rational, Rational Software Corporation, ClearQuest, PureCoverage, Purify, Purify'd, Quantify, and Rational Visual Test, among others, are either trademarks or registered trademarks of Rational Software Corporation in the United States and/or in other countries. All other names are used for identification purposes only, and are trademarks or registered trademarks of their respective companies.

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

PATENT

U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,835,701. Additional patents pending.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

GOVERNMENT RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Rational Software Corporation license agreement and as provided in DFARS 277.7202-1(a) and 277.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct. 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 227-14, as applicable.

WARRANTY DISCLAIMER

This document and its associated software may be used as stated in the underlying license agreement. Rational Software Corporation expressly disclaims all other

warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability or fitness for a particular purpose or arising from a course of dealing, usage, or trade practice.

Contents

Welcome to the Rational PurifyPlus Product Family 1

Rational PurifyPlus: What it is 1

Tips for development engineers 2

Tips for test engineers 3

Other PurifyPlus resources 4

Contacting Rational technical support 5

Contacting Rational technical publications 5

Getting Started: Rational Purify 7

Purify for Visual C/C++ developers and testers 7

Purify for Visual C/C++: What it does 7

Purify for Visual C/C++: The basic steps 9

Purify for Visual C/C++: Advanced features 20

Purify for Java developers and testers 26

Purify for Java: What it does. 26

Purify for Java: The basic steps 28

Purify for Java: Advanced features. 36

Purify for .NET managed code developers and testers 40

Purify for .NET managed code: What it does 40

Purify for .NET managed code: The basic steps 41

Purify for .NET managed code: Advanced features 50

Getting Started: Rational PureCoverage 55

PureCoverage: What it does. 55

PureCoverage: The basic steps 57

PureCoverage: Advanced features. 64

Getting Started: Rational Quantify 73

Quantify: What it does. 73

Quantify: The basic steps 74

Quantify: Advanced features. 85

Index 93

Welcome to the Rational PurifyPlus Product Family

Rational PurifyPlus: What it is

Rational® PurifyPlus brings together three essential tools that help you you develop high-quality applications more efficiently:

- **Rational Purify®** An automatic error detection tool for finding runtime errors and memory leaks in every component of your program.
- **Rational Quantify®** A performance analysis tool for resolving performance bottlenecks so your program can run faster.
- **Rational 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 Visual C/C++, Visual Basic, Java, or managed code in any language that Microsoft Visual Studio .NET supports.

If you're developing code in Visual Studio, start the PurifyPlus tools from the Visual Studio menus or toolbars. 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 the PurifyPlus tools into existing test scripts and harnesses to automate error detection, memory profiling, 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.

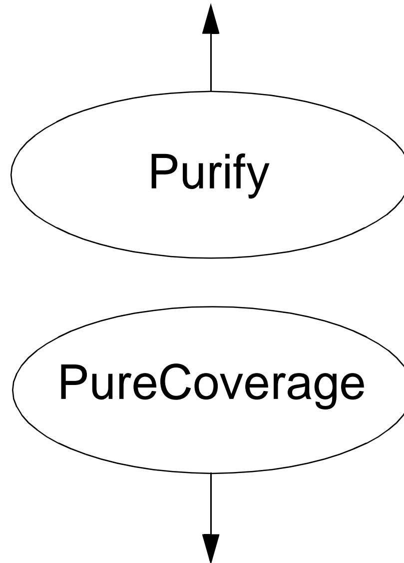
Do yourself a favor. Don't waste days looking for problems that PurifyPlus can pinpoint in seconds. And don't release a product with hidden bugs that these tools can detect easily. Consistent use of the PurifyPlus tools, from the time you start development until you ship, will provide solid benefits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



Improve code coverage

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

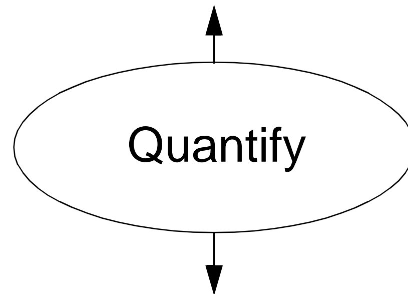
Use PureCoverage to make sure you're exercising all your code during pre-checkin testing.

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

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.



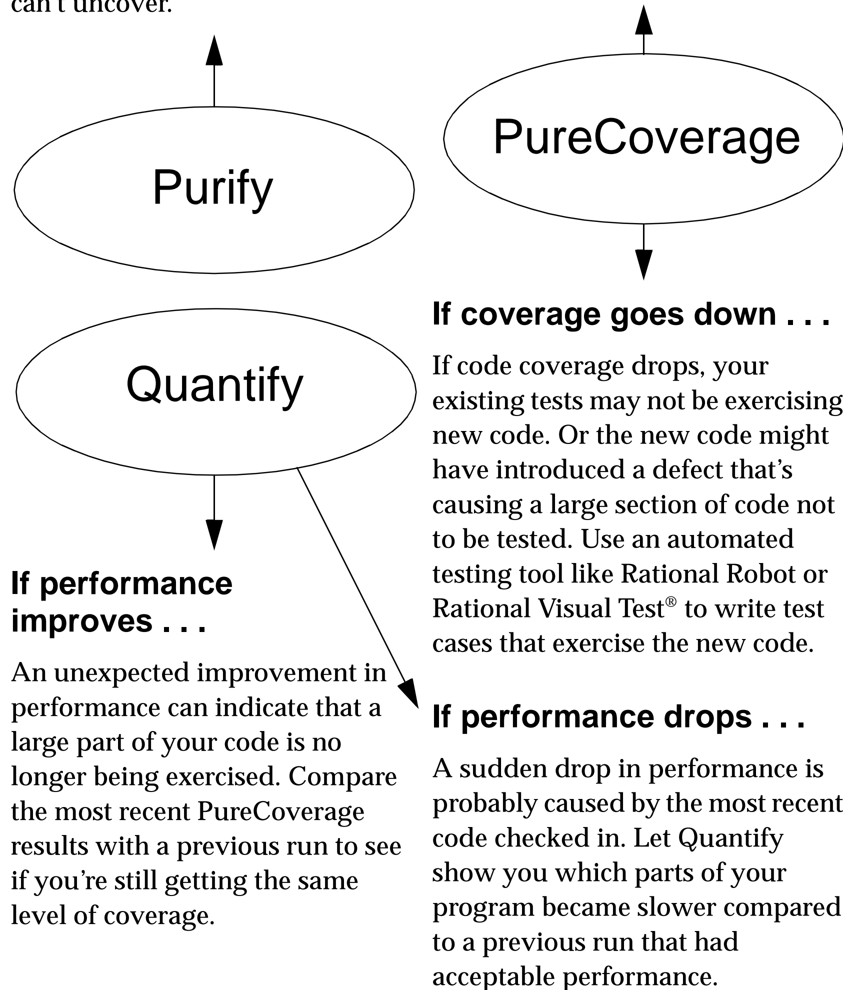
Analyze code structure

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.

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.

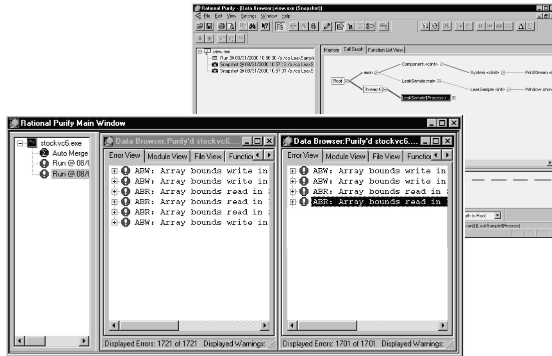
If performance drops . . .

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.

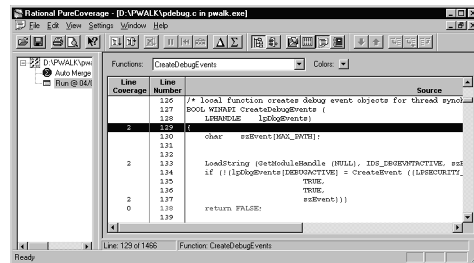
Other PurifyPlus resources

Additional information is available for all the PurifyPlus tools:

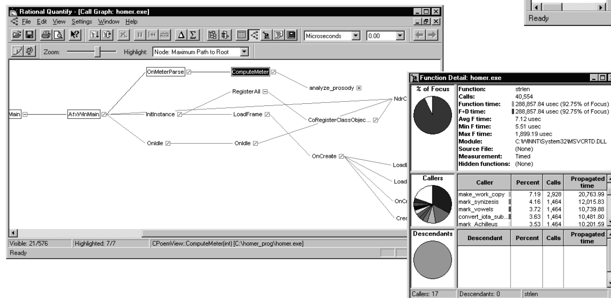
To learn how to pinpoint hard-to-find bugs in C/C++, Java, and managed code, go to *Getting Started: Rational Purify* on page 7



To learn how you can avoid shipping untested code, go to *Getting Started: Rational PureCoverage* on page 55



To learn how to highlight performance bottlenecks, go to *Getting Started: Rational Quantify* on page 73



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 Rational Software and other Rational products, go to http://www.rational.com.

Contacting Rational technical support

You can contact Rational technical support by email at support@rational.com.

You can also reach Rational technical support over the Internet or by telephone. For contact information, as well as for answers to common questions about Purify, Quantify, and PureCoverage, go to http://www.rational.com/support.

Contacting Rational technical publications

To order copies of Rational publications, go to the Rational Press at http://www.rational.com/support/documentation/index.jsp#press.

Please send any feedback about Rational documentation to the Rational technical publications department at techpubs@rational.com.

Getting Started: Rational Purify

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

Purify for Visual C/C++ developers and testers

Purify for Visual C/C++: 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 Visual 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 Visual 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
- Visual C/C++ components embedded within Visual Basic applications, Internet Explorer, Netscape Navigator, 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.

Purify for Visual C/C++: 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.

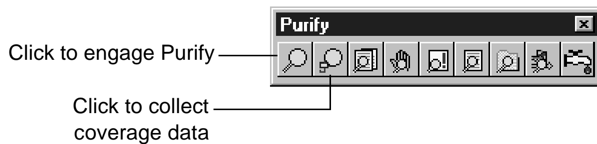
The following pages show you how to use Purify integrated with Microsoft Visual Studio 6, but you can also use Purify in other ways. Read the following:

- *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 toolbar.

If you have installed Rational PureCoverage, set Purify to collect coverage data in addition to checking for errors and memory leaks.

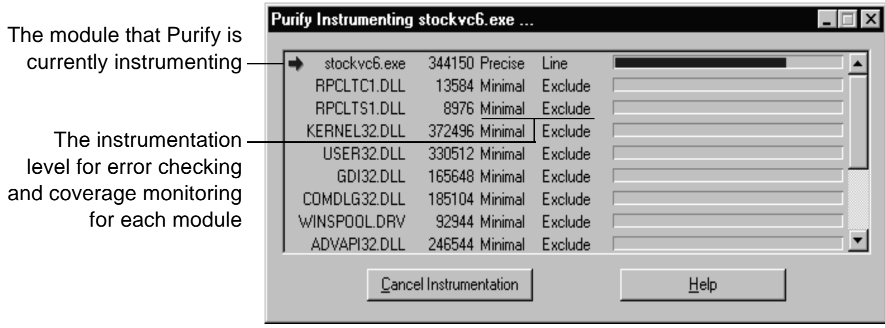


Build and execute your program using commands from the Visual Studio **Build** menu. To get the maximum level of detail in Purify error and coverage data, build your program with debug and relocation data.

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.

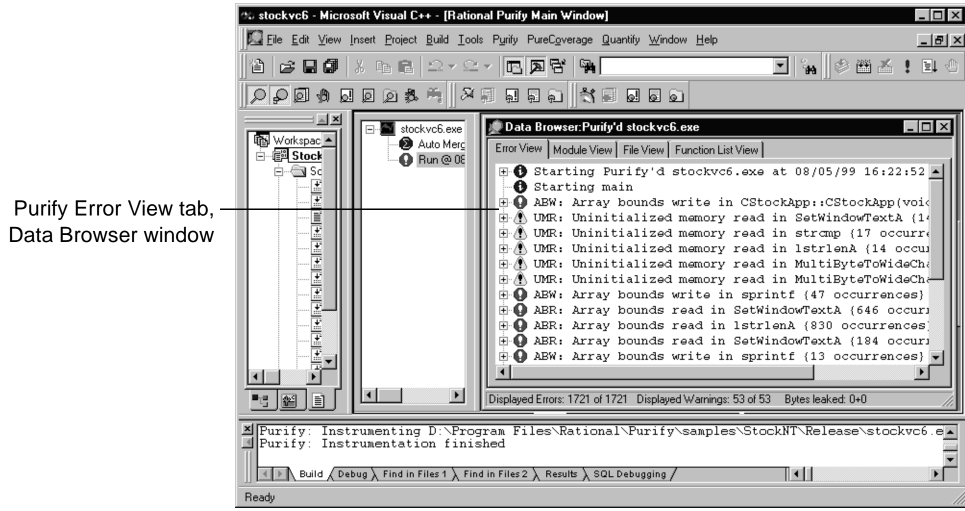


Purify instruments each module at a default instrumentation level. If you want to focus on a specific part of your program, you can override the default and customize the instrumentation level.

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.

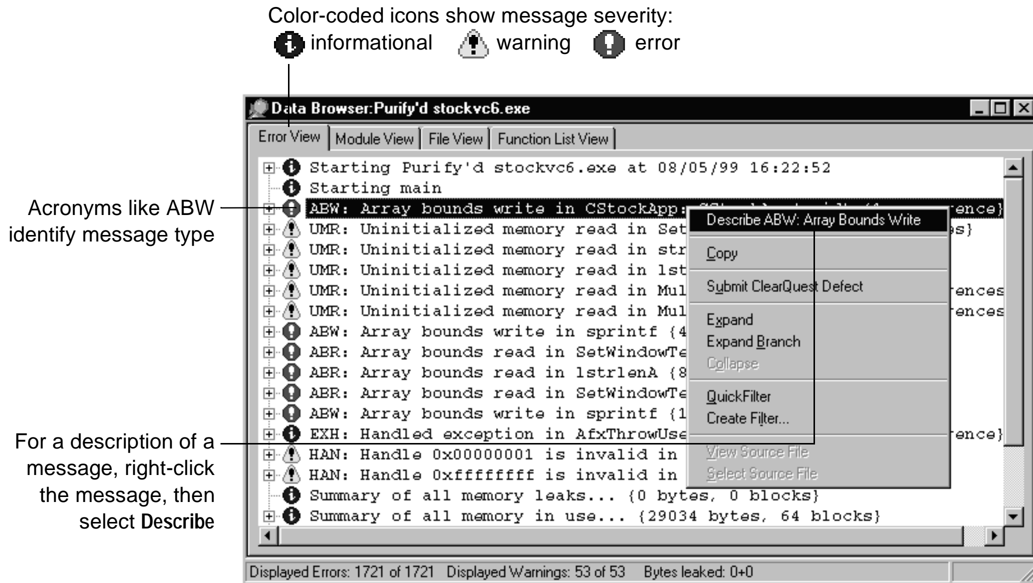


More information? Look up *error view* in the Purify online Help index.

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.



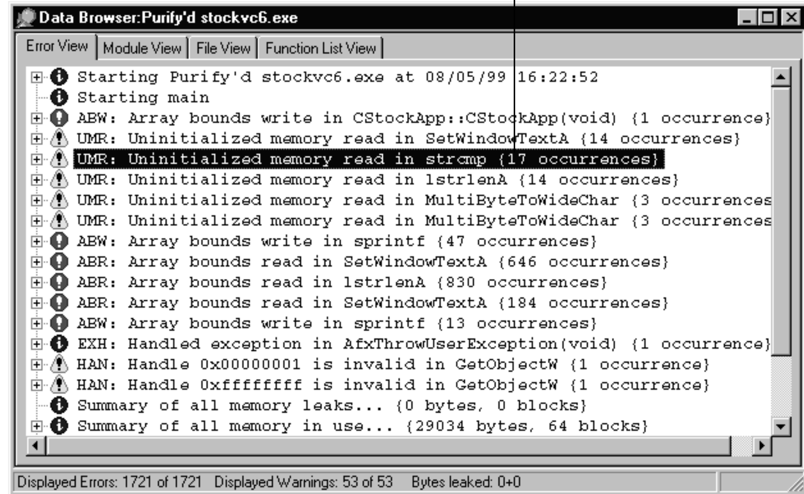
When you exit the program, Purify reports memory leaks. In addition to memory leaks, you can set Purify to report memory in use at exit and handles in use at exit.

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.

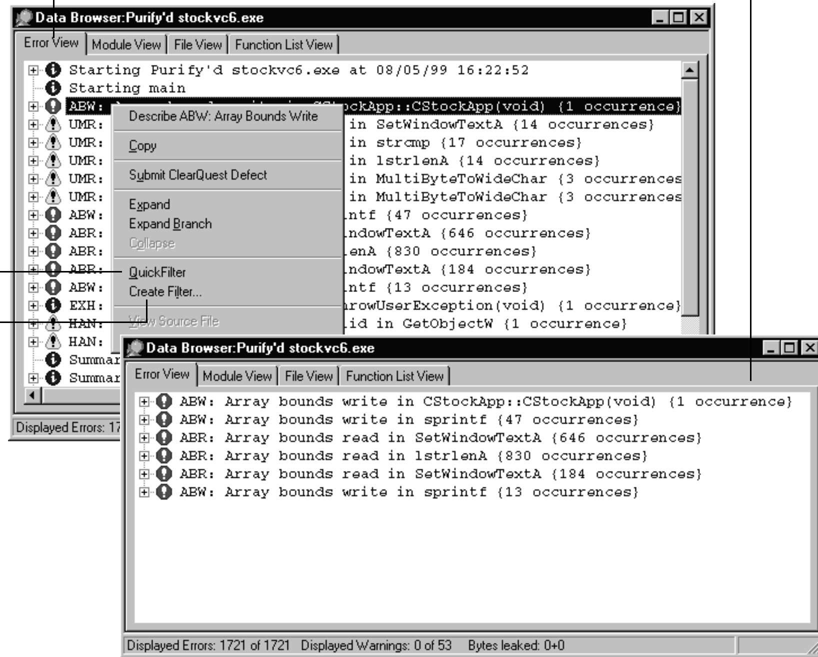
You can filter messages individually, or you can filter them based on their type and source. Consider hiding all informational messages, for example, or all messages originating from a specific file.

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

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

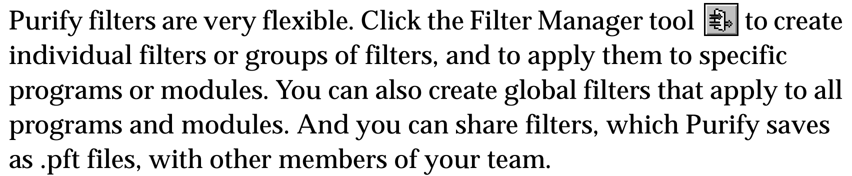
Right-click a message and select **QuickFilter** to hide the message immediately

Or select **Create Filter** to define a set of filtering criteria

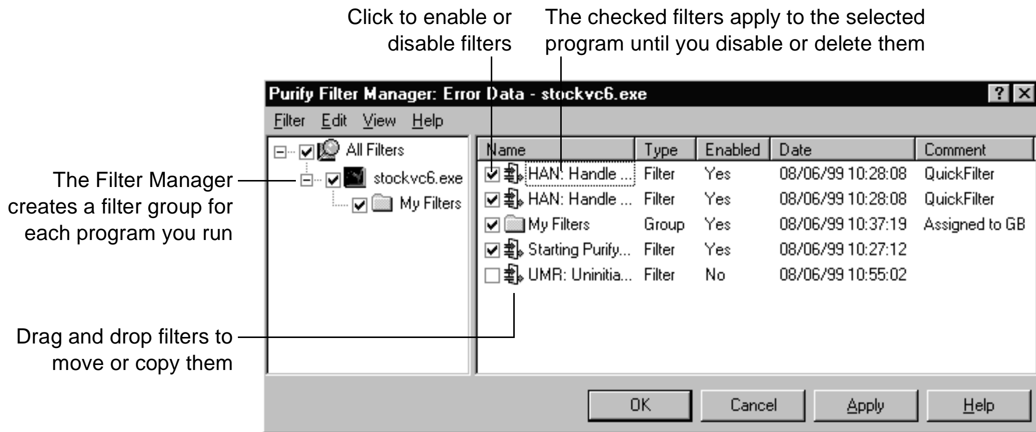


Once created, error filters apply to the current run and to all future runs of the program until you disable them. Disabling a filter causes hidden messages to be redisplayed in the error view.

Working with error data 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.



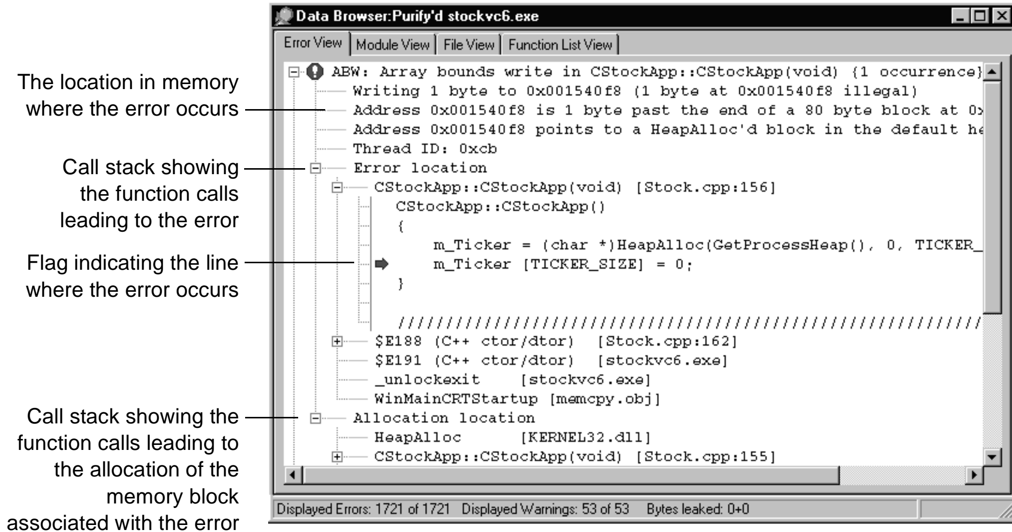
More information? Purify provides filters for coverage data as well as for error data. Look up *filtering data* in the Purify online Help index.

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:



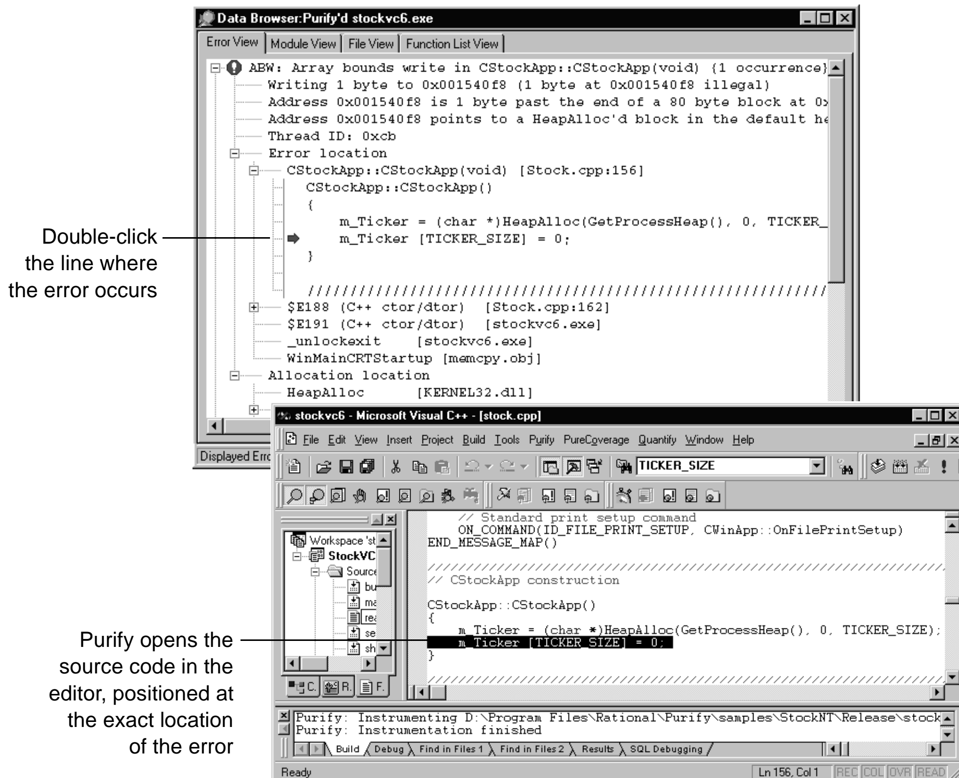
The level of detail provided in call stacks depends on the availability of debug and relocation data. Even if you build your program in release mode, you can still get the highest possible level of detail. For more information, look up *debug data (C/C++), release version* in the Purify online Help index.

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.



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

Data Browser:Purify'd stockvc6.exe

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

Coverage Item	Calls	Functions Missed	Functions Hit	% Functions Hit	Lines Missed	Lines Hit	% Lines Hit
CDialog::HandleInitDialog(UI..	1		hit		11	8	42.11
CDialog::HandleSetFont(UIN..	1		hit		0	3	100.00
CDialog::InitModalIndirect(DI..	0	missed			5	0	0.00
CDialog::InitModalIndirect(vc..	0	missed			5	0	0.00
CDialog::OnCancel(void)	0	missed			2	0	0.00
CDialog::OnCmdMsg(UINT,i..	12		hit		7	7	50.00
CDialog::OnCommandHelp(L..	0	missed			9	0	0.00
CDialog::OnCtlColor(CDC *,C..	26		hit		0	2	100.00
CDialog::OnHelpHitTest(UIN..	0	missed			4	0	0.00
CDialog::OnInitDialog(void)	1		hit		5	10	66.67
CDialog::OnOK(void)	1		hit		0	4	100.00
CDialog::OnSetFont(CFont *)	1		hit		0	1	100.00
CDialog::PostModal(void)	1		hit		1	9	90.00
CDialog::PreInitDialog(void)	1		hit		0	1	100.00

Coverage Item: Ascending order | Function: CDialog::OnInitDialog(void)

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

D:\Program Files\Microsoft Visual Studio\VC98\MFC\SRC\dlgcore.cpp in stockvc6.exe

Functions: CDialog::OnInitDialog(void) Colors:

Line Coverage	Line Number	Source
	660	BOOL CDialog::OnInitDialog()
1	661	{
	662	// execute dialog RT_DLGINIT resource
	663	BOOL bDlgInit;
1	664	if (m_lpDialogInit != NULL)
0	665	bDlgInit = ExecuteDlgInit(m_lpDialogInit);
0	666	else
1	667	bDlgInit = ExecuteDlgInit(m_lpszTemplateName);
	668	
1	669	if (!bDlgInit)
	670	{
	671	TRACE0("Warning: ExecuteDlgInit failed during dial
	672	EndDialog(-1);
1	673	return FALSE;
	674	}

Line: 660 of 892 | Function: CDialog::OnInitDialog(void)

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.

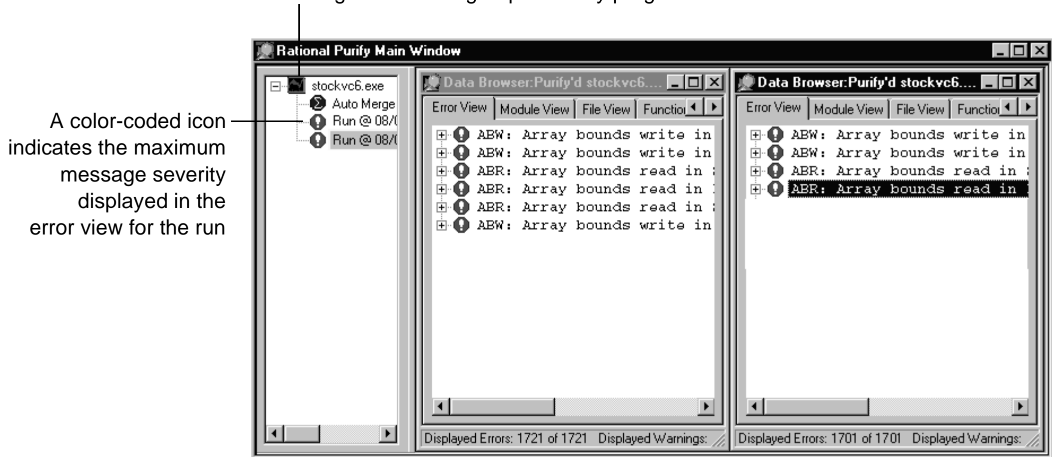
More information? Look up *coverage data (C/C++)* in the Purify online Help index.

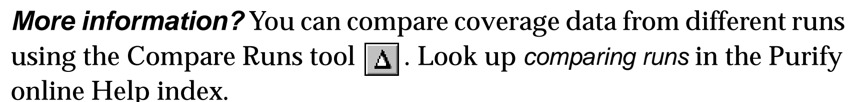
Comparing program runs

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





More information? You can compare coverage data from different runs using the Compare Runs tool [Δ]. 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.

Purify for Visual C/C++: Advanced features

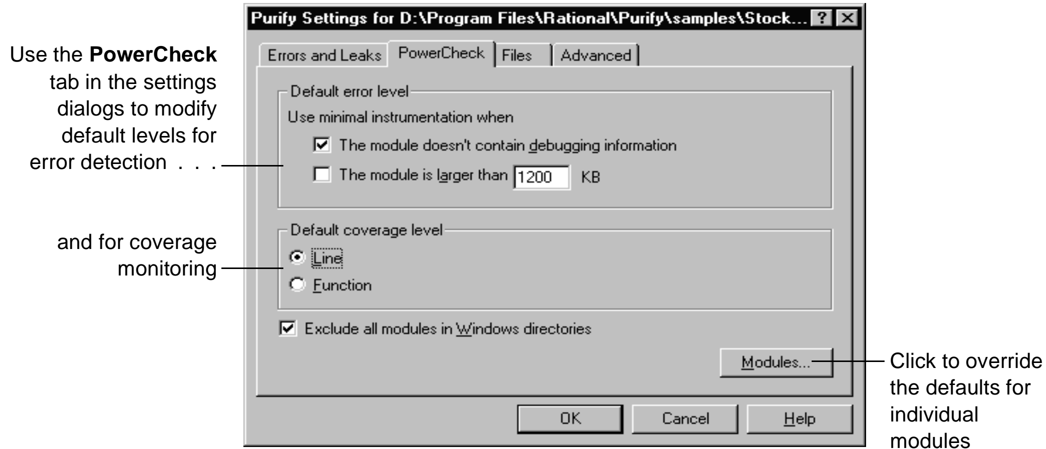
Customizing instrumentation

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:

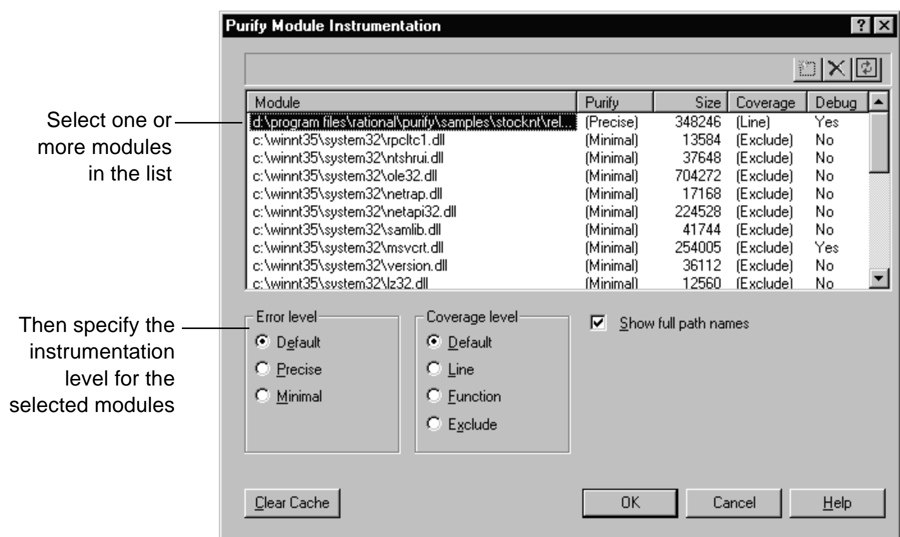
- *Precise* instrumentation, which provides full run-time error detection to pinpoint problems in any part of your program
- *Minimal* instrumentation, which improves Purify's performance while providing a basic level of error detection

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



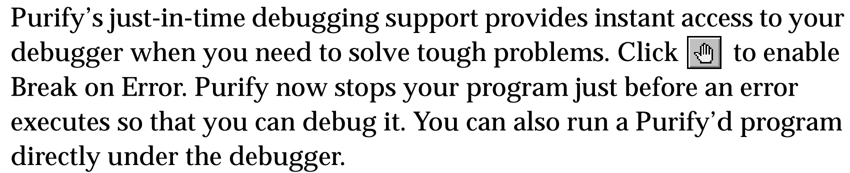
You can override the default and specify the level for each module to meet your own requirements.



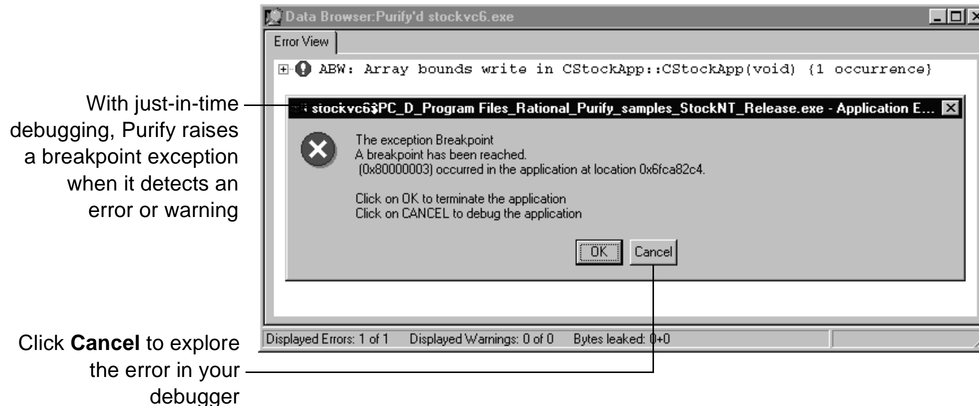
Try using the Precise error level for the most critical modules in your program and the Minimal level for the others. Later, you can change the Minimal level to Precise for a thorough check of the other modules.

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 [icon] 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.



To quickly debug *only* the most critical errors in your program, use Break on Error together with Purify error 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.

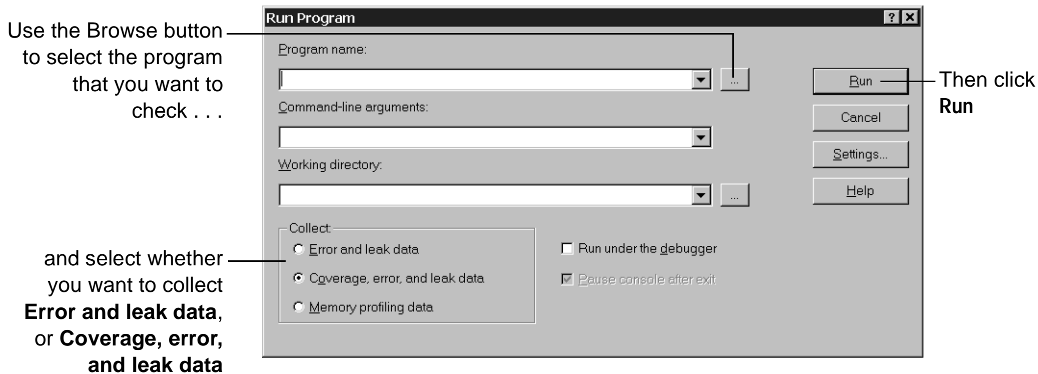
More information? Look up *break on error tool (C/C++)* in the Purify online Help index.

Using Purify standalone

When you don't need all of the Microsoft Visual Studio 6 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.



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 [icon] 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
```

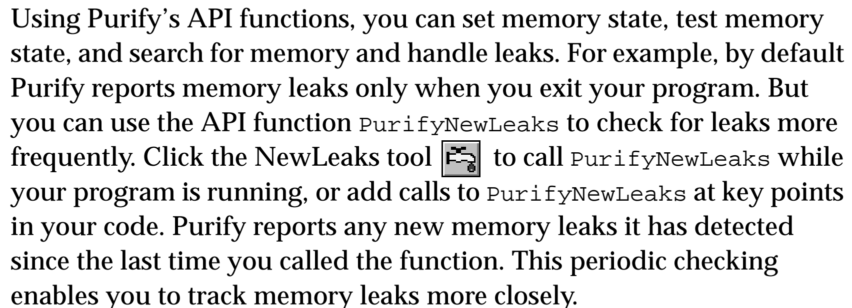
You can run Purify without the graphical interface by using the `/SaveTextData` option. This option saves Purify's diagnostic

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 6 debugger, as well as by including them in your code.

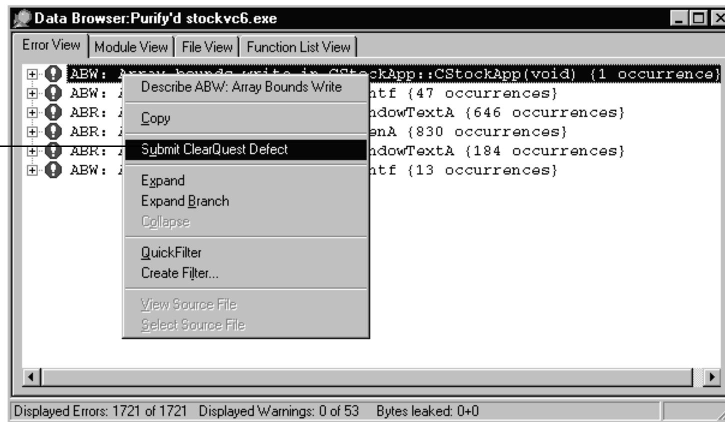
More information? For the complete listing of Purify API functions, including functions related to coverage monitoring, look up *api function list*. For instructions on using the functions, look up *api functions, using* in the Purify online Help index.

Using Rational Software integrations

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

Using Purify with ClearQuest

If you have 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.

If you have Visual Test installed, you can run Purify on the program that Visual Test is exercising within Visual Studio. If you are using a test harness to run Visual Test scripts, you can easily modify it to run Purify automatically as it exercises the program.

More information? Look up *clearquest*, *robot*, and *visual test* in the Purify online Help index, and refer to the ClearQuest, Robot, and Visual Test 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 Java developers and testers

Purify for Java: What it does

Java memory leaks?

Yes, there *are* Java memory leaks, and they can be serious.

The Java virtual machine (JVM) garbage collector automatically removes from memory objects that your program no longer needs, and so avoids most of the memory leaks that occur in other programming contexts. But Java applications can still consume more and more memory over time. The causes for this can be extremely difficult to track down. Purify makes it much easier to find and fix them.

There are two major categories of leaks in Java: object references that are no longer needed, and system resources that are not freed.

Object references that are no longer needed

Very often, Java code retains references to memory that it no longer needs, and this prevents the memory from being garbage collected. Java 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 called, 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

Java methods can also allocate heap memory that exists outside of Java instances, such as resources for windows and bitmaps. Java often allocates these resources by calling C or C++ routines using Java Native Interface (JNI) calls.

How Purify can help

Purify helps you find these Java 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. To free system resources, check your Java toolkit for help. For example, the `dispose()` method in Sun Microsystem's Abstract Windowing Toolkit (AWT) frees the system resources used by the Frame, Dialog, and Graphics classes.

You can gather memory profiling data any time your program runs. If you want to test a new feature before you check in your code, run the code from Purify's graphical user interface; see *Purify for Java: The basic steps* on page 28. To gather data automatically from your test harness, use Purify's command-line interface in your test scripts and insert Purify API function calls in your code to control the data collection; see *Integrating Purify into your Java test environment* on page 38.

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: Rational PureCoverage* on page 55 and *Getting Started: Rational Quantify* on page 73.

Purify for Java: The basic steps

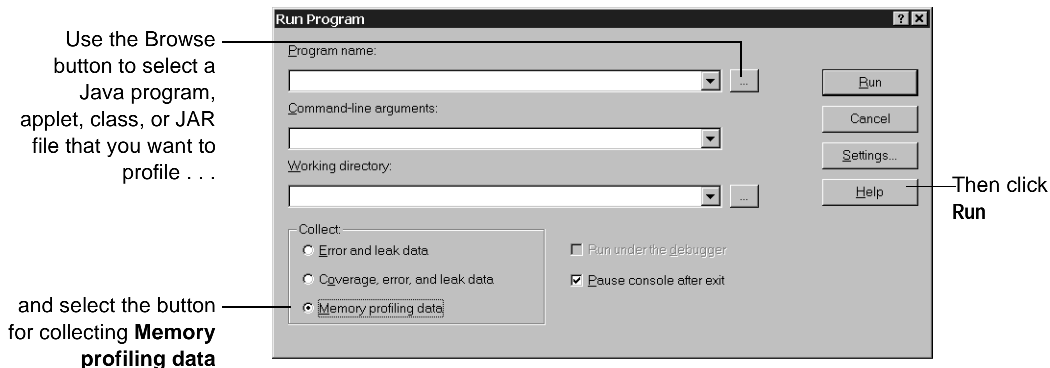
Java applications can consume a lot of memory over time if a forgotten reference to an object unintentionally prevents it from being garbage collected. With Rational Purify, you can determine how much memory your Java program is using, and detect exactly which objects are responsible for these "memory leaks." You can also identify places where forcing a garbage collection would improve your code's performance.

To use Purify to profile Java memory usage:

1. Run your Java 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.

Running your Java program with Purify

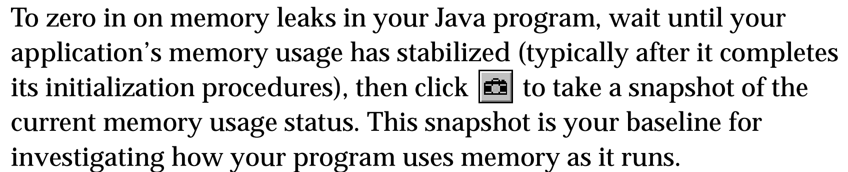
To Purify your Java program, start Purify and click **Run** in the Welcome Screen to display the Run Program dialog.



More information? Look up *specifying a JVM (Java)* and *running programs* in the Purify online Help index.

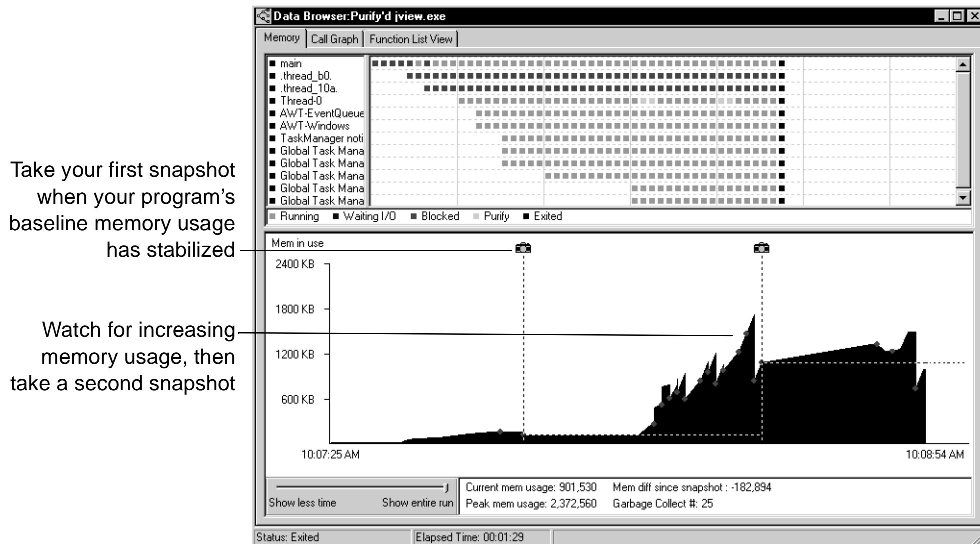
As your program runs, Purify intercepts and tabulates messages related to memory usage from the JVM. Based on these messages, Purify keeps track of how much memory your program has allocated to each method and object at any given time.

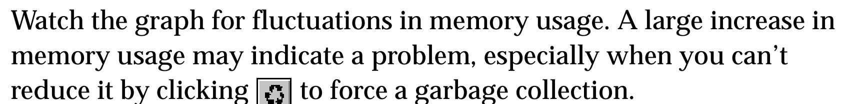
Taking snapshots of memory use



To zero in on memory leaks in your Java program, wait until your application's memory usage has stabilized (typically after it completes its initialization procedures), then click 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 is 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.





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 clicking to force a garbage collection.

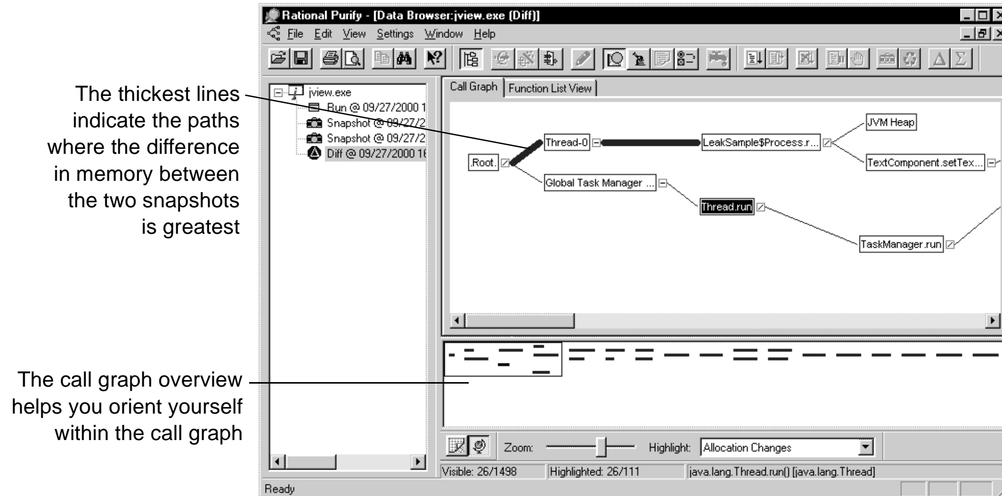
Now take another snapshot so that you have a "before" and "after" record of what's going on, and exit your program.

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 click to compare the second snapshot with the first.

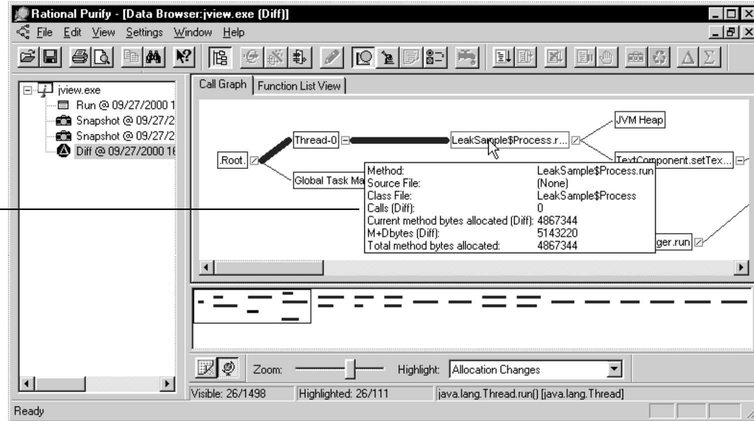
Purify now displays a call graph showing the methods that are responsible for allocating the largest amounts of memory during the interval between the first and second snapshots.



The 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.

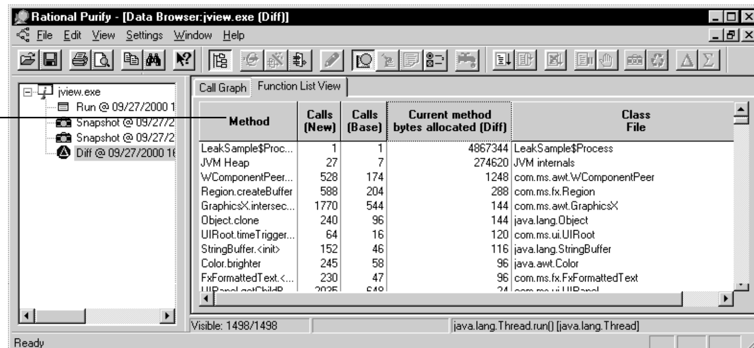
To view your code from within Purify, right-click a method for which source is available, then select **Source File**.

More information? Look up *diff'ing 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. You can do full-program sorts in the Function List View to find the biggest memory-consuming methods in your entire program.

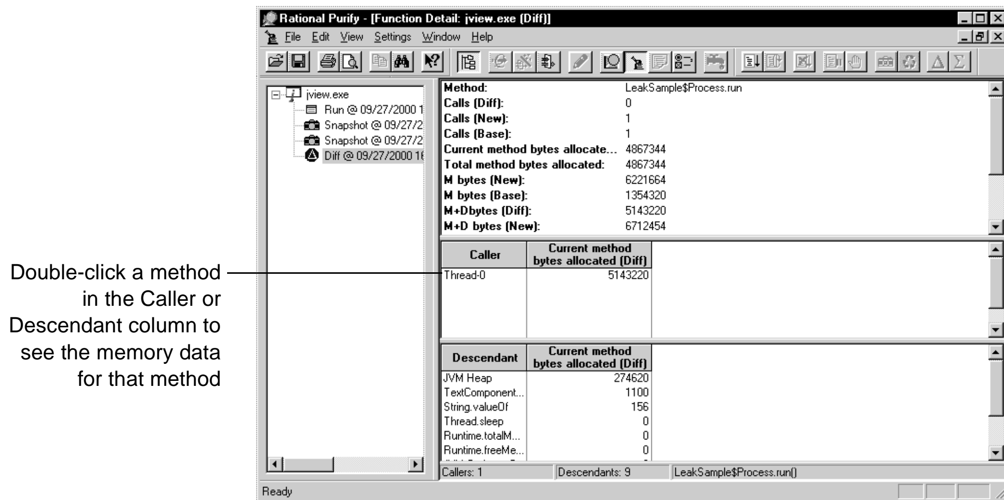
Click a column header to sort the memory profiling data



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.



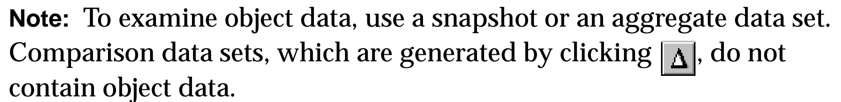
More information? Look up *function detail window* in the Purify online Help index.

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.

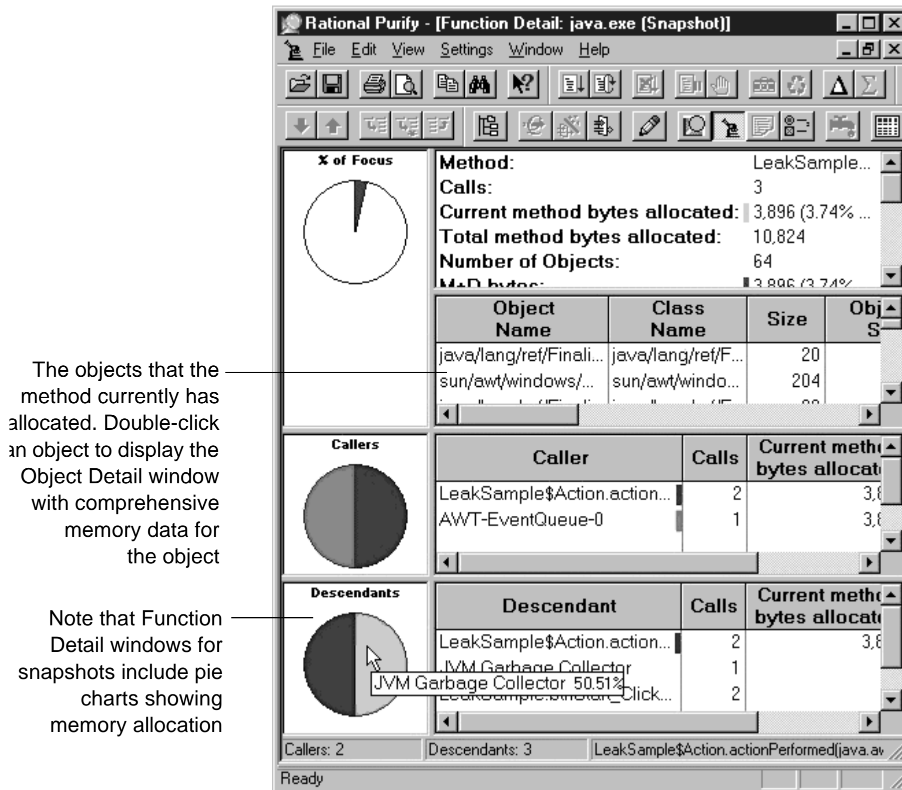
Looking for unneeded objects

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: To examine object data, use a snapshot or an aggregate data set. Comparison data sets, which are generated by clicking , 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.



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 so that you can 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

Pause the mouse over an object for detailed memory information

Choose a criterion for highlighting objects in the reference graph

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

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

Rational Purify - [Data Browser:java.exe (Snapshot)]

File Edit View Settings Window Help

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

Object Name	Class Name	Method Name	Size	O+R Size	GCs Survived
char [8194] 2113D...	char []	JVM Garbage Colle...	16,388	16,388	0
char [8194] 21137...	char []	LeakSample.<init>(j...	16,388	16,388	0
byte [8196] 211806...	byte []	LeakSample$Proce...	8,196	8,196	4
byte [8196] 2117E...	byte []	LeakSample$Proce...	8,196	8,196	4
byte [8196] 2113B...	byte []	LeakSample.<init>(j...	8,196	8,196	0
byte [8196] 21135...	byte []	LeakSample.<init>(j...	8,196	8,196	0
short [4034] 21131...	short []	LeakSample.<init>(j...	8,068	8,068	0
char [4034] 2112F0...	char []	LeakSample.<init>(j...	8,068	8,068	0
<Unknown Class> ...	<Unknown...	LeakSample.<init>(j...	3,692	33,564	0
char [1434] 2112D...	char []	LeakSample.<init>(j...	2,868	2,868	0
char [1338] 2112E...	char []	LeakSample.<init>(j...	2,676	2,676	0
int [633] 21133D68	int []	LeakSample.<init>(j...	2,532	2,532	0
char [1266] 21130F...	char []	LeakSample.<init>(j...	2,532	2,532	0
char [1178] 21134...	char []	LeakSample.<init>(j...	2,356	2,356	0

Object: 2/8354 | char [8194] 21137AD0 | LeakSample.<init>(java.lang.String)

Ready

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 Java data in the following formats:

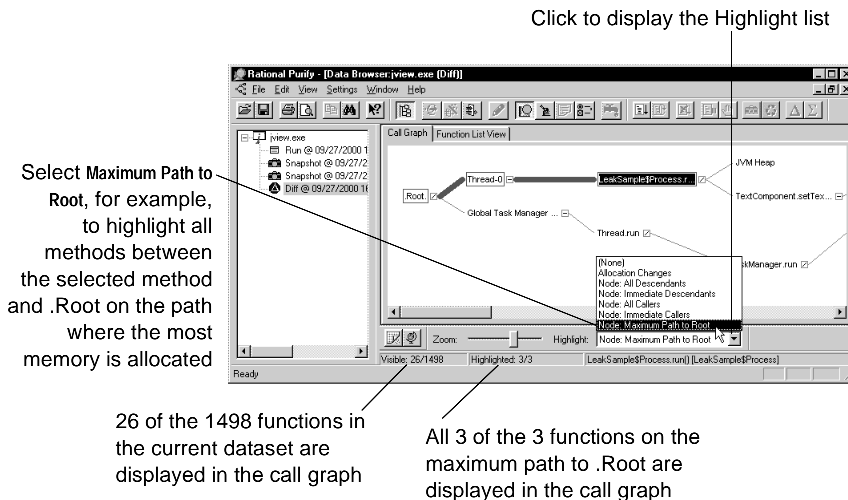
- Purify memory profiling files (.pmy). You can open these files and view them in Purify, just as you would any run, snapshot, or other dataset.
- ASCII text files (.txt). You can process this data with scripts or use it in spreadsheet and word-processing applications.

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

Purify for Java: Advanced features

Highlighting methods that share key attributes

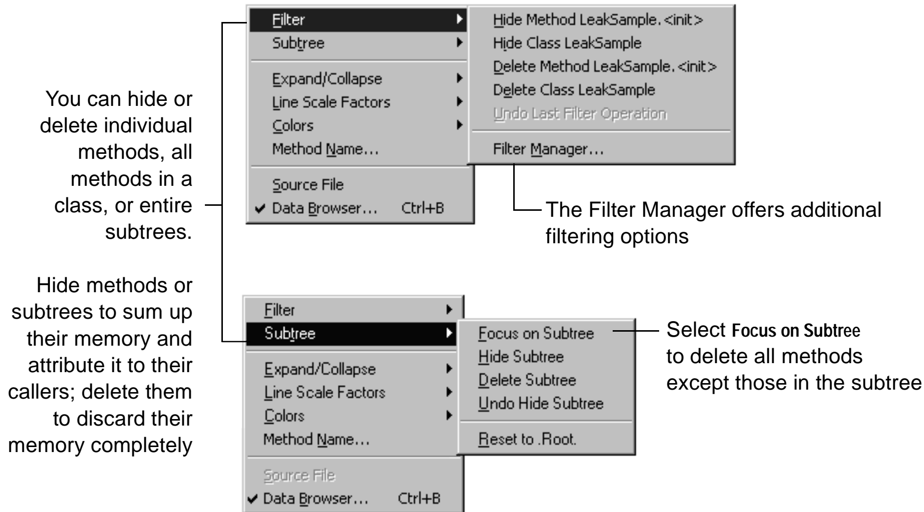
You can highlight methods in the call graph to display specific memory-related characteristics or to show calling relationships.



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

Focusing your data

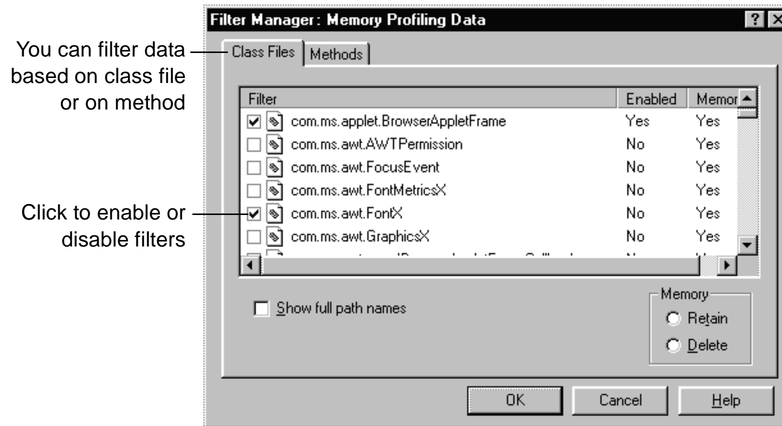
Use Purify’s *filter* commands to remove a selected method, or all methods in a class file, from the set of data that Purify has collected. Alternatively, use *subtree* commands to focus on or remove a specific method and all its descendants from the dataset. Right-click a method in the call graph, function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



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.



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

Integrating Purify into your Java 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 Java class file and are using Sun Microsystem's Java viewer, change the line that runs the class file to:

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

This command runs your class file 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 Java memory profiling with the Purify API*, immediately following.

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

Controlling Java 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 Java code. Remember that Purify's online Help contains detailed information to assist you.

Purify for .NET managed code developers and testers

Purify for .NET managed code: What it does

Purify finds and reports memory leaks in .NET managed code (assemblies, .exe's, .dll's, OLE/ActiveX controls, and COM objects) just as it does in Java. If you've used Purify for Java, you'll find the information in the following sections familiar.

Memory leaks in managed code

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

The .NET garbage collector automatically removes from 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. The causes for this can be extremely difficult to track down. Purify makes it much easier to find and fix them.

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 called, 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 C or 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: Rational PureCoverage* on page 55, and *Getting Started: Rational Quantify* on page 73.

Purify for .NET managed code: The basic steps

Managed code applications can consume a lot of memory over time if a forgotten reference to an object unintentionally prevents it from being garbage collected. With Rational® Purify®, you can determine how much memory your managed code program is using, and detect

exactly which objects are responsible for these “memory leaks.” You can also identify places where forcing a garbage collection would improve your code’s performance.

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.

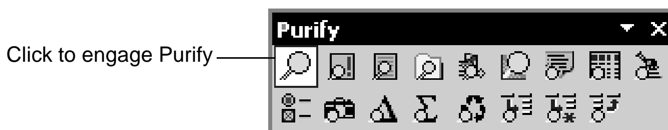
The following pages show you how to use Purify integrated with Microsoft Visual Studio .NET, but you can also use Purify in other ways. Read the following:

- *Using Purify standalone* on page 52
- *Integrating Purify into your managed code test environment* on page 53.

Running your managed code program with Purify

The first time you use Purify in Visual Studio .NET, display the Purify toolbar by selecting **Toolbars > Purify** from the Visual Studio **View** menu. The instructions in this section refer to the Purify toolbar, but if you prefer you can use the corresponding commands from the **Purify** menu instead.

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

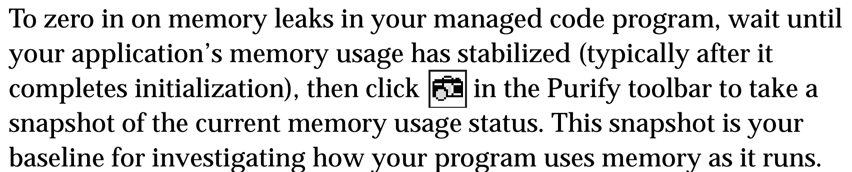


Build and execute your program as usual, using commands from the Visual Studio menu. To get the maximum level of detail in Purify memory profiling data, build your program with debug data.

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

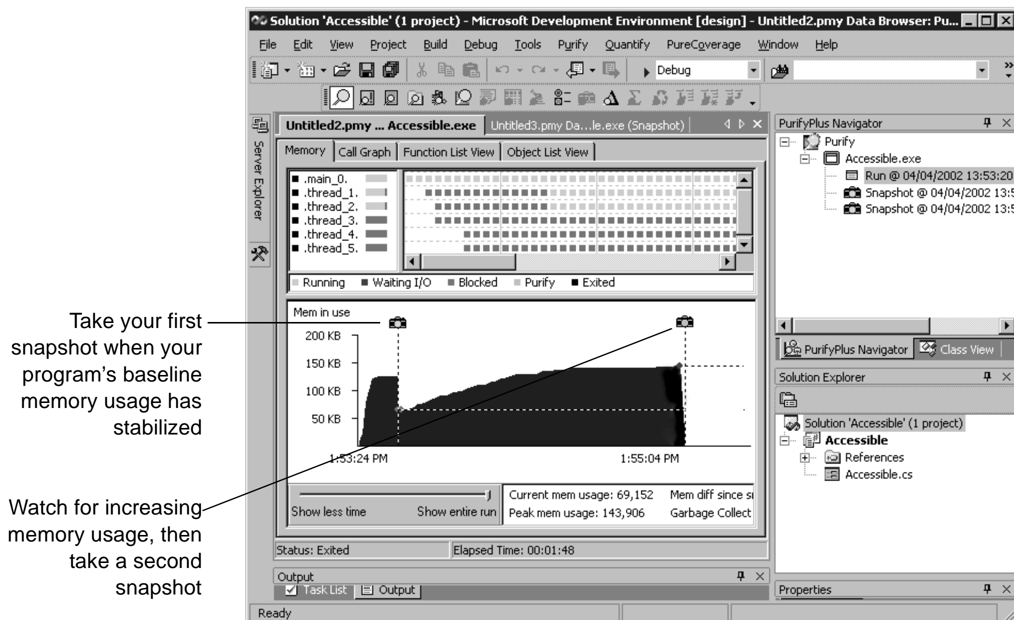
As your program runs, Purify intercepts and tabulates messages related to memory usage from the .NET runtime environment. Based on these messages, Purify keeps track of how much memory your program has allocated to each method and object at any given time.

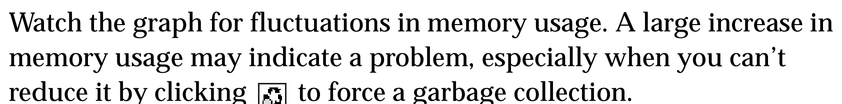
Taking snapshots of memory use



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 click [icon] in the Purify toolbar 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.





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 clicking [icon] to force a garbage collection.

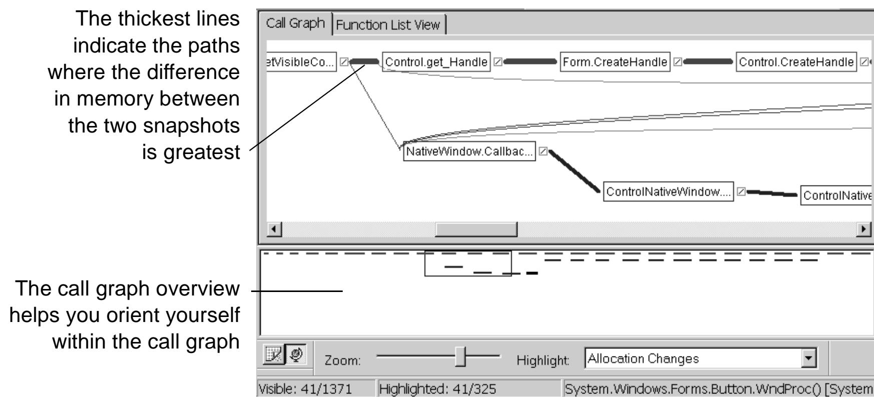
Now take another snapshot so that you have a “before” and “after” record of what’s going on, and exit your program.

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 click in the Purify toolbar to compare the second snapshot with the first.

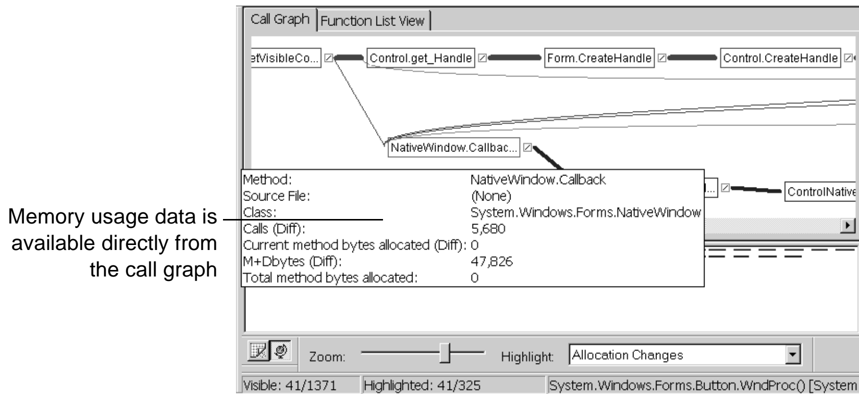
Purify now displays a call graph showing the methods that are responsible for allocating the largest amounts of memory during the interval between the first and second snapshots.



The 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.

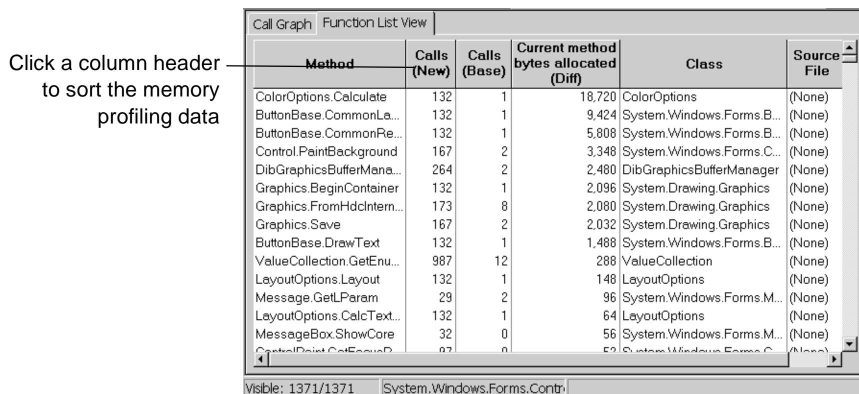
To view your code from within Purify, right-click a method for which source is available, then select **Source File**.

More information? Look up *diff'ing 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. 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



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)

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

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()

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

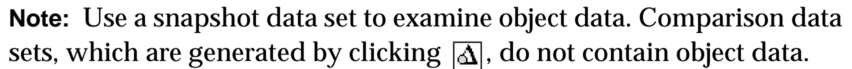
More information? Look up *function detail window* in the Purify online Help index.

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.

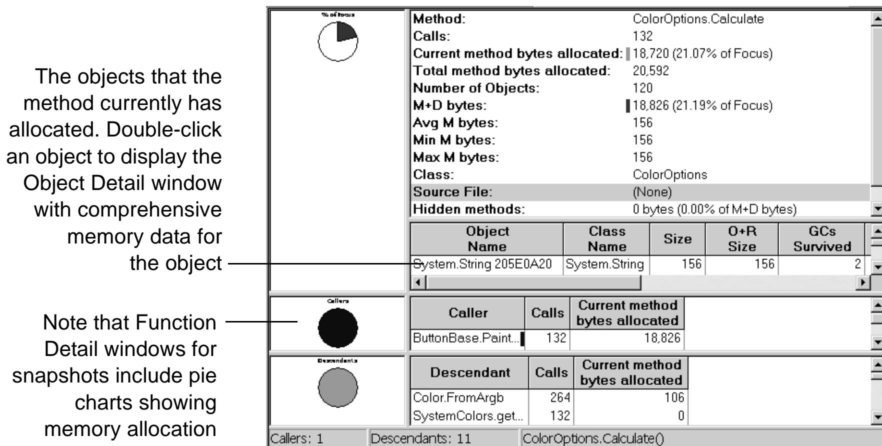
Looking for unneeded objects

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 clicking , 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.

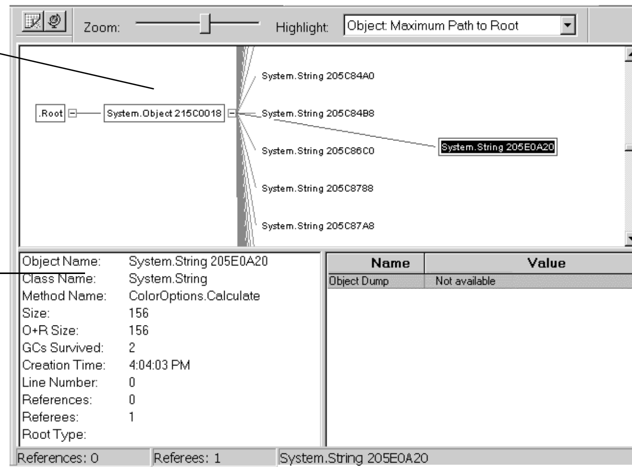


Examining object details

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 so that you can 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

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



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.

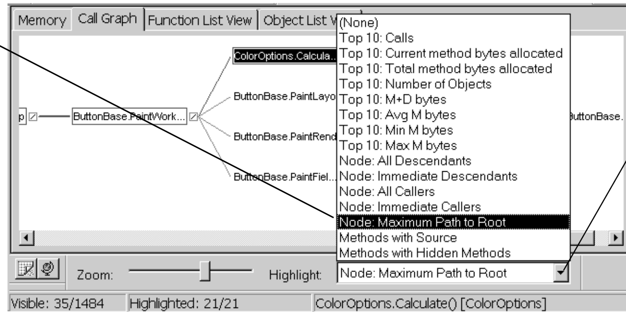
More information? Look up *saving data* in the Purify online Help index.

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

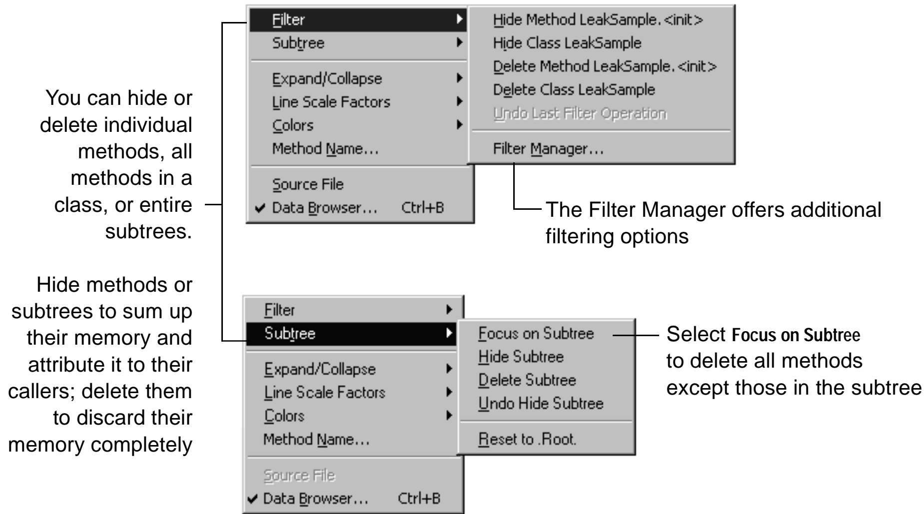
All 21 of the 21 functions on the maximum path to .Root are displayed in the call graph

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

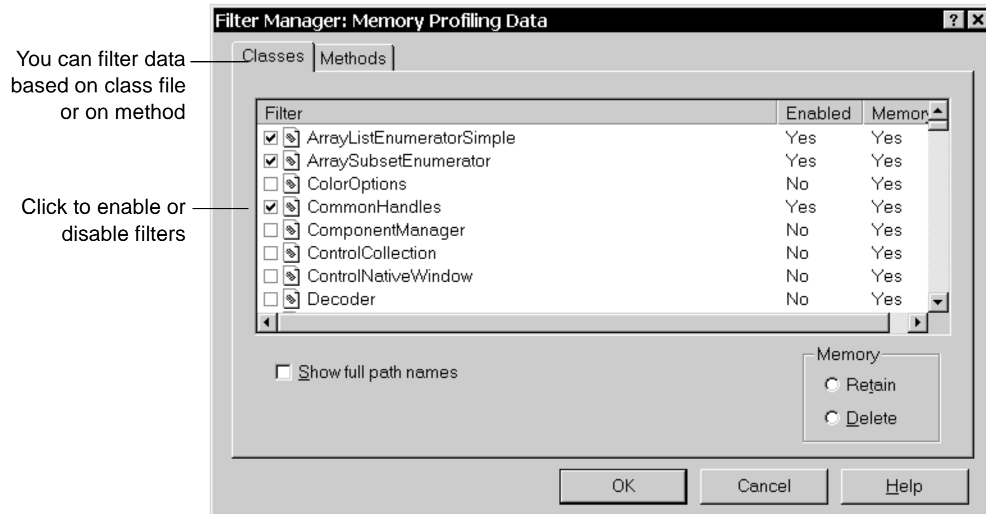
method and all its descendants from the dataset. Right-click a method in the call graph, function list view, or function detail to perform these operations.



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.

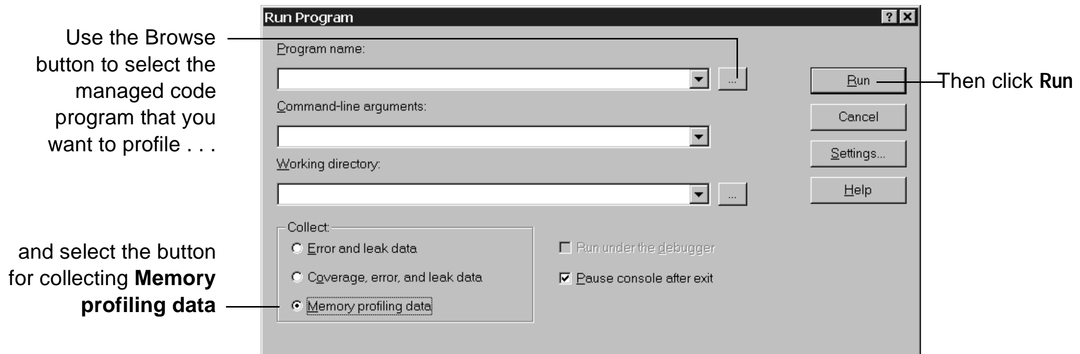


More information? Look up *filtering data* and *subtrees* in the Purify online Help index.

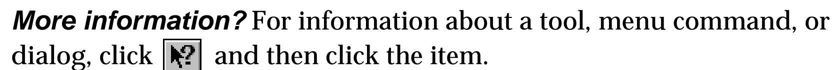
Using Purify standalone

When you don't need all of the Microsoft Visual Studio .NET 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.



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: Rational PureCoverage

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 Rational® 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 Visual 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 and Microsoft Visual Basic

Check every component in your program

PureCoverage analyzes coverage for every component in:

- Visual C/C++ code in .exe's, .dll's, OLE/ActiveX controls, and COM objects
- Visual Basic projects and 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 .NET.
- Components launched from container programs such as Microsoft Internet Explorer, the Microsoft Transaction Server, jexegen'd executables, Jview.exe, Tstcon32.exe, Netscape Navigator, 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 and Microsoft Visual Basic, so you can use it without changing the way you work if you're developing code in these environments.

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.

PureCoverage: The basic steps

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

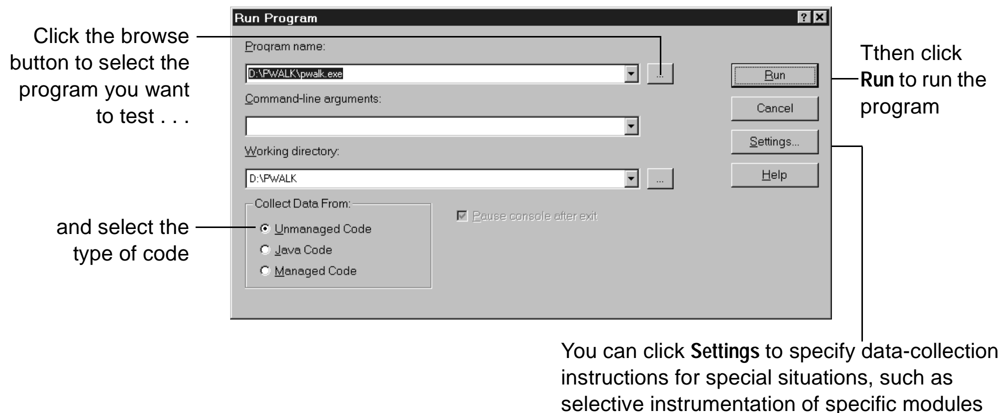
1. Run a program with PureCoverage.
2. Get the big picture with the Coverage Browser and Function List windows. Use PureCoverage filters to focus on the areas that concern you most.
3. Identify unexercised lines in the Annotated Source window.
4. Modify your test run to cover missed lines, conditions, 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 Microsoft Visual Basic, or when you incorporate it into your test harness. For more information, read *Integrating PureCoverage with your development desktop* on page 65 and *Integrating PureCoverage in your test environment* on page 70.

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.

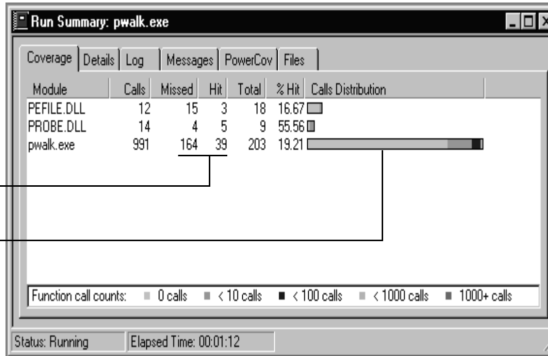


Your program begins to execute. As it runs, PureCoverage collects comprehensive information about what lines and functions have been exercised.

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.

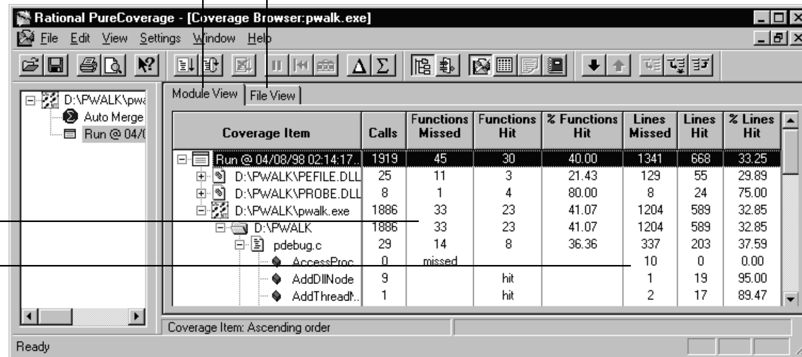
The Coverage Browser window provides coverage data organized hierarchically according to source file.

The Module View tab groups data by file within modules

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

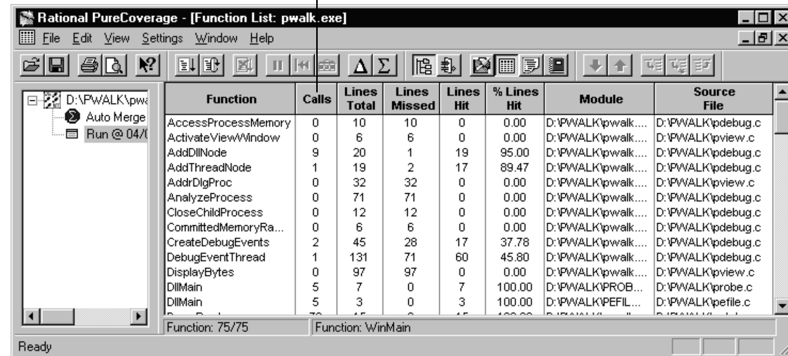
The Coverage Browser window shows hierarchical coverage information for functions, procedures, or methods . . .

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

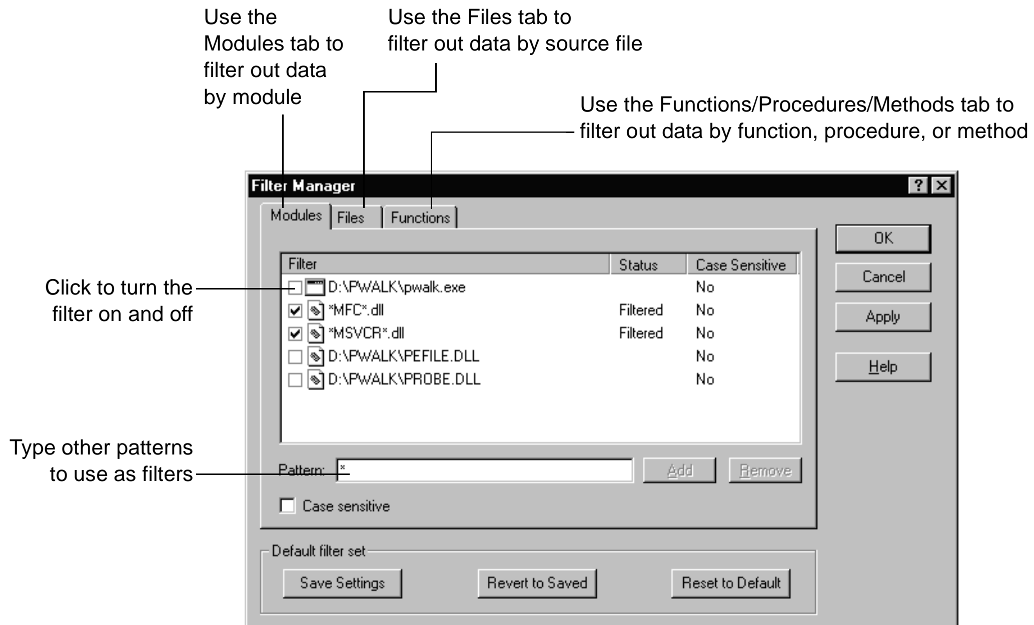


More information? To learn how to customize the data display, look up *coverage browser window* and *function list window* in the PureCoverage online Help index.

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.

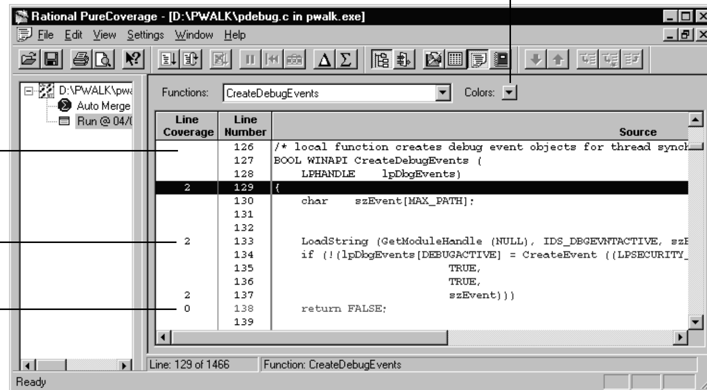
To see the data that PureCoverage has filtered out, or to change the filtering to display other information that concerns you, click the Filter Manager tool to open the Filter Manager dialog.



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.



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 or by using an Immediate window in Visual Basic. 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 or the Immediate window, see your Visual Studio or Visual Basic 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

Rational PureCoverage - [Coverage Browser:pwalk.exe (Auto Merge)]

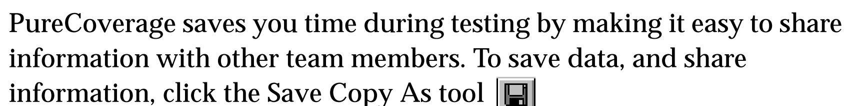
Coverage Item	Calls	Functions Missed	Functions Hit	% Functions Hit	Lines Missed	Lines Hit	% Lines Hit
Auto Merge @ 08/12/98 07:56...	2315	43	32	42.67	1318	707	34.91
D:\PWALK\PEFILE.DLL	68	11	3	21.43	129	55	29.89
D:\PWALK\PROBE.DLL	20	1	4	80.00	8	24	75.00
D:\PWALK\pwalk.exe	2227	31	25	44.64	1181	628	34.72

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 70.

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.

PureCoverage: 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 6, Microsoft Visual Studio .NET, Microsoft Visual Basic, Rational Visual Test®, Rational Robot, and Rational ClearQuest™.

During installation, a PureCoverage menu and toolbar are automatically added to Visual Studio 6 and Visual Basic so you can monitor your code at any time during development, without leaving your development environment. In Visual Studio .NET, select **Toolbars > PureCoverage** from the **View** menu to display the toolbar.

Click the Engage PureCoverage Integration tool in the PureCoverage toolbar, then run your program

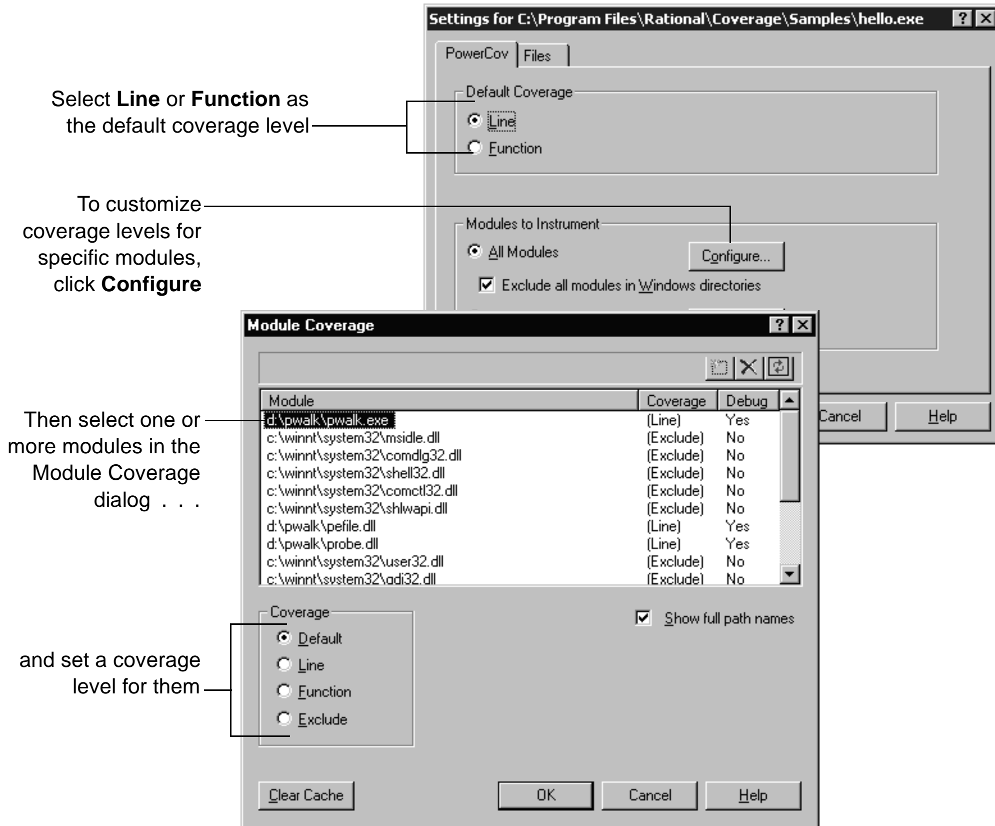
View and work with coverage data directly within Visual Studio 6 or Visual Studio .NET

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



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.

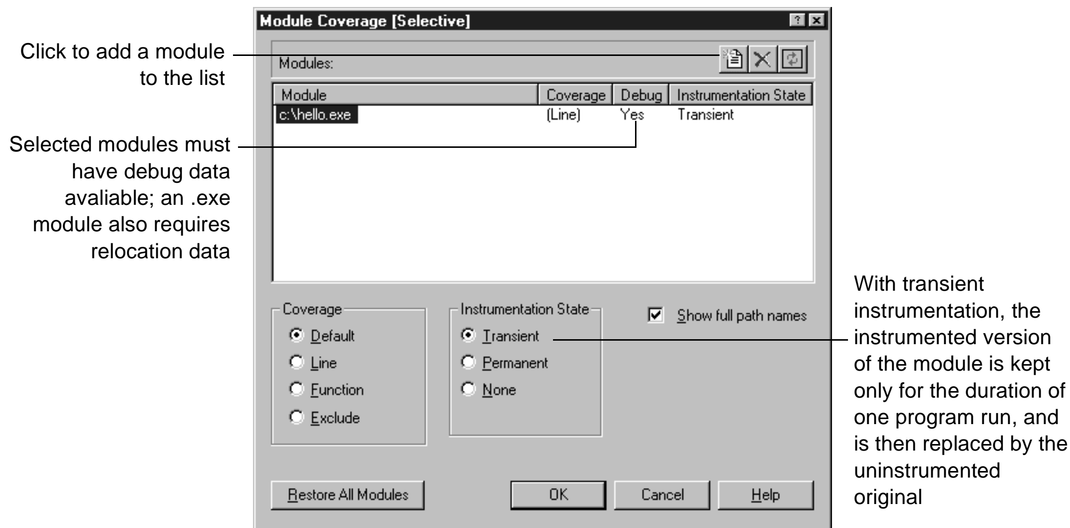
More information? Look up *settings, overview* and *coverage levels, overview* in the PureCoverage online Help index.

Using Selective instrumentation

If you are working in Visual 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 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 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.



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

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

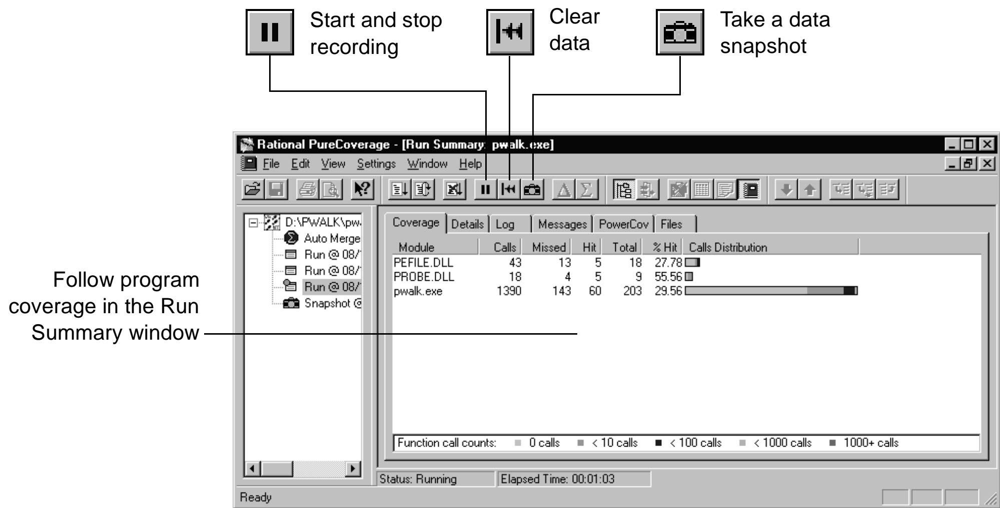
Zeroing in on key program areas

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.



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

More information? Look up *visual test* and *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: Rational Quantify

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?

Rational® 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:

- Visual C/C++ code in .exe's, .dll's, OLE/ActiveX controls, and COM objects
- Visual Basic projects and 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 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, `jexegen`'d executables, Jview.exe, Tstcon32.exe, Netscape Navigator, 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 Visual 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 6, Microsoft Visual Studio .NET, and Microsoft Visual Basic, so you can use Quantify without changing the way you work if you're developing code in these environments.

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 Java methods and class files, and to Visual Basic procedures and object libraries.

Quantify: 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:

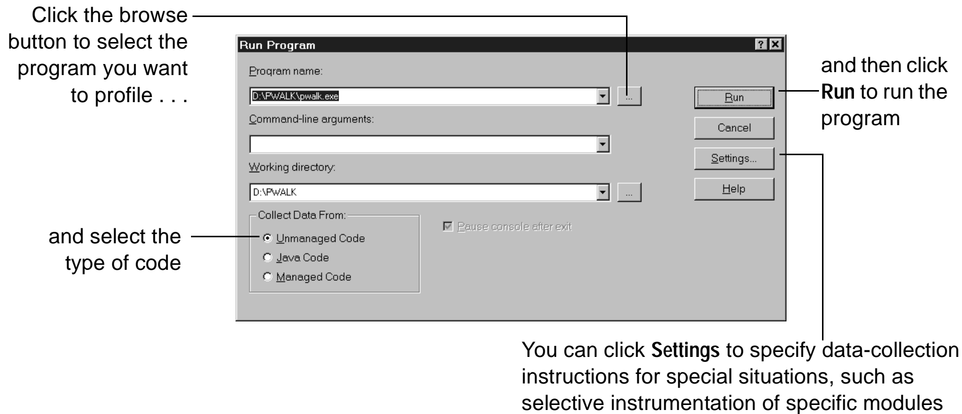
1. Run the program with Quantify to collect performance data.
2. Use the Quantify data windows to analyze the performance data and find bottlenecks.
3. Modify your code to reduce or eliminate bottlenecks.
4. Rerun the program and use the Compare Runs tool to verify performance improvements

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 Microsoft Visual Basic, or

when you incorporate Quantify into your test harness. For more information, read *Integrating Quantify with your development desktop* on page 85 and *Integrating Quantify in your test environment* on page 92.

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.

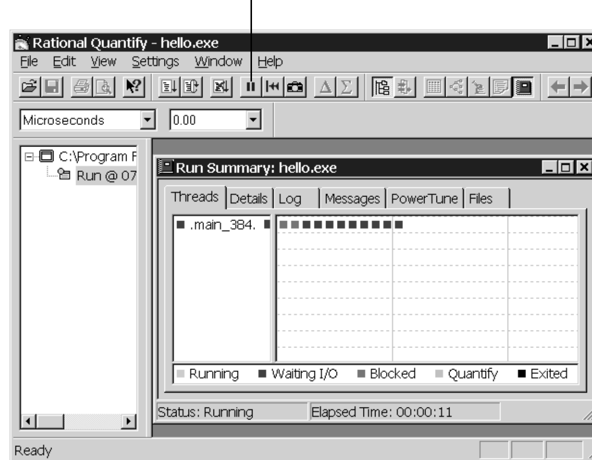


You can click **Settings** to specify data-collection instructions for special situations, such as selective instrumentation of specific modules

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

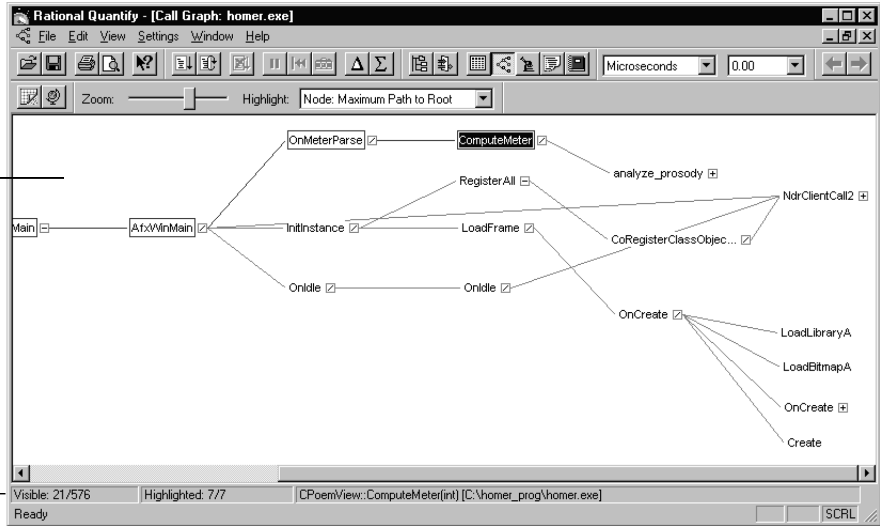
The second step in improving your program's performance is to analyze the performance data that Quantify has collected.

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

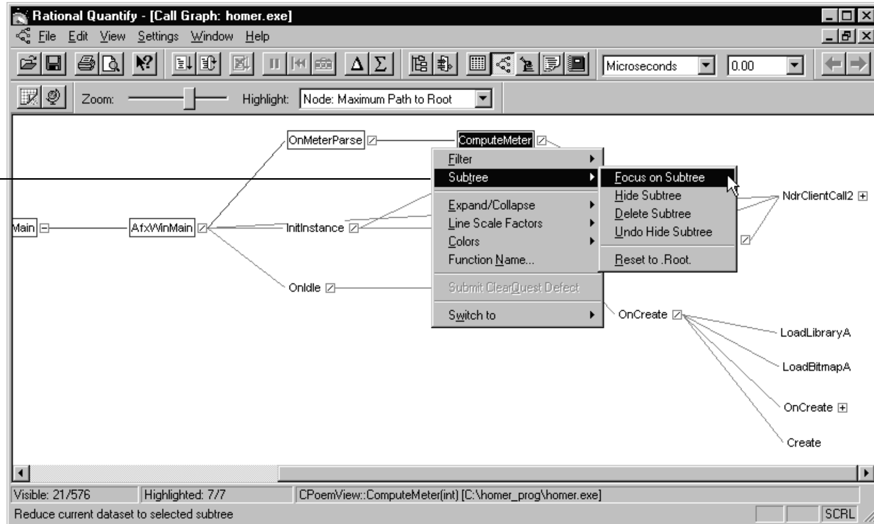


The call graph initially highlights the most expensive path. You can choose instead to highlight functions based on various criteria, including performance, calling relationships, and possible causes for bottlenecks. You can also show additional functions, hide functions, and grab and move functions 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.

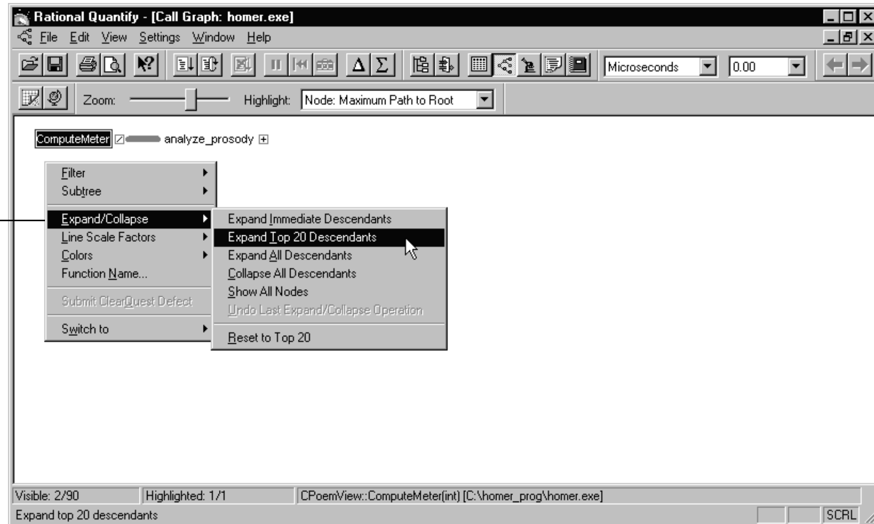
Having located a suspicious function, you can isolate it to examine where it spends its time

The Subtree commands adjust the focus of the dataset



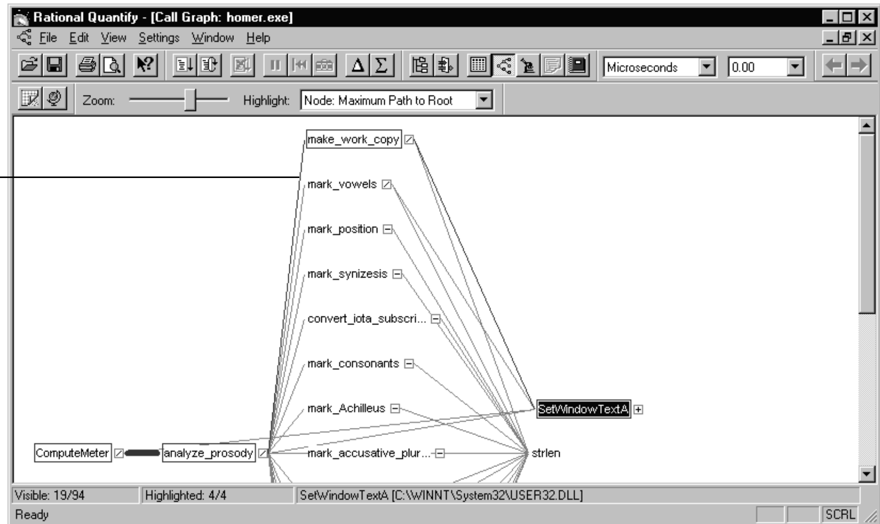
Quantify adjusts the dataset so it contains only `ComputeMeter` and its descendants. You can now can expand the `ComputeMeter` subtree to see what's going on downstream.

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

Click the Function List tool to display 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.

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 List: homer.exe

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

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.

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

Function List: homer.exe

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

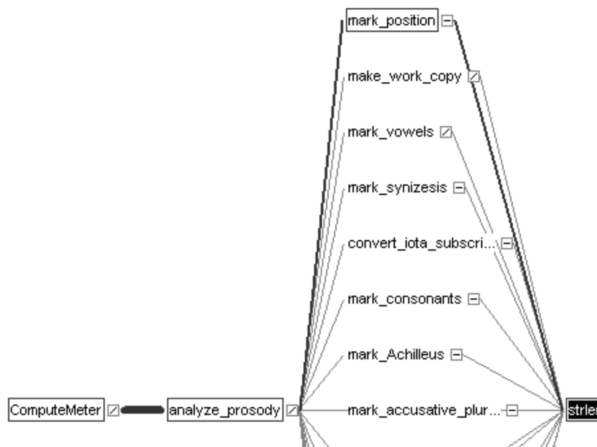
Visible: 93/93 CPoemView::ComputeMeter(int)

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

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

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

and by a function (to descendants)

Function Detail: homer.exe

% of Focus

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)

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

Descendants

Descendant	Percent	Calls	Propagated time

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

C:\homer_prog\prosody.cpp

Functions: mark_consonants(void)

Line time	L+D time	Percent of Function time	Percent of F+D time	Line number	Source
					**
					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

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

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.

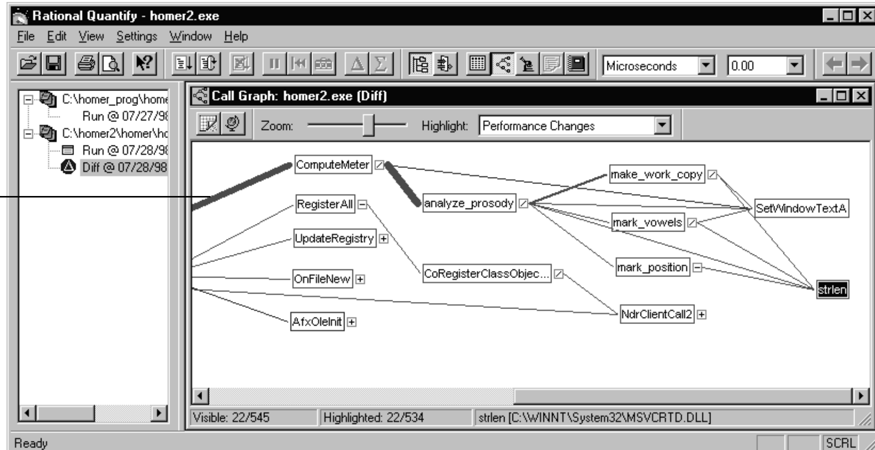
Compare the modified program's performance

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 List: homer2.exe (Diff)

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

Visible: 545/545 CPoemView::ComputeMeter(int)

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.

Quantify: 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.

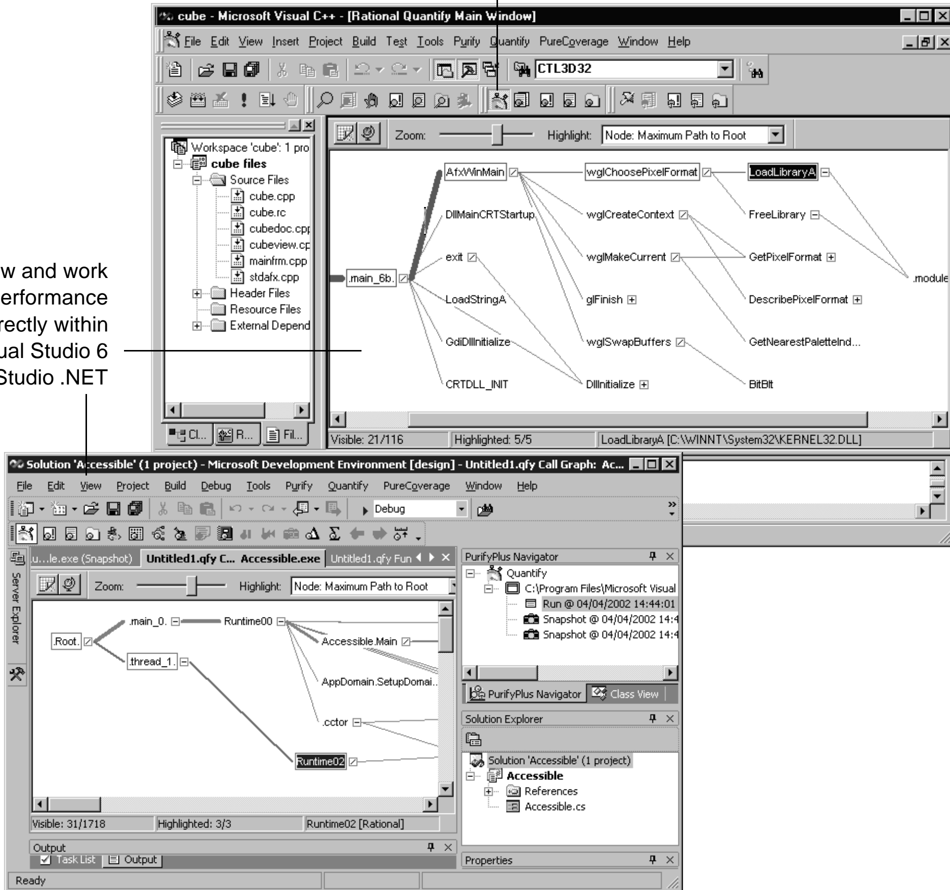
Integrating Quantify with your development desktop

Quantify's integration—for example, with Microsoft Visual Studio, Microsoft Visual Basic, Rational Visual Test®, 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 and toolbar are automatically added to Visual Studio 6 and Visual Basic so you can profile your code at any time during development, without leaving your development environment. The first time you use Quantify in Visual Studio .NET, display the Quantify toolbar by selecting **Toolbars > Quantify** from the Visual Studio **View** menu.

Click the Engage Quantify Integration tool in the Quantify toolbar, then run your program

View and work with performance data directly within Visual Studio 6 and Visual Studio .NET



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

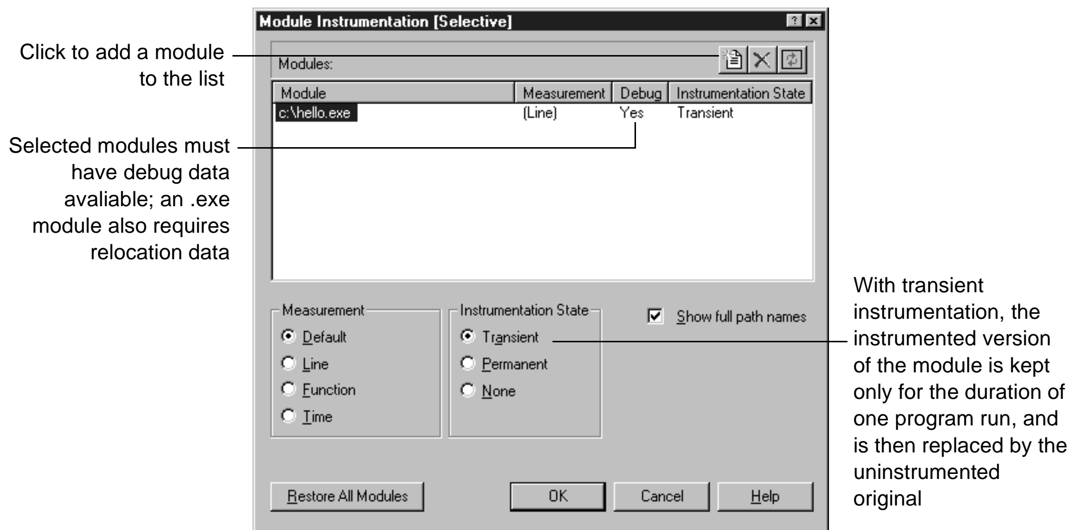
More information? Look up *integrating* in the Quantify online Help index.

Using selective instrumentation

If you are working in Visual 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.

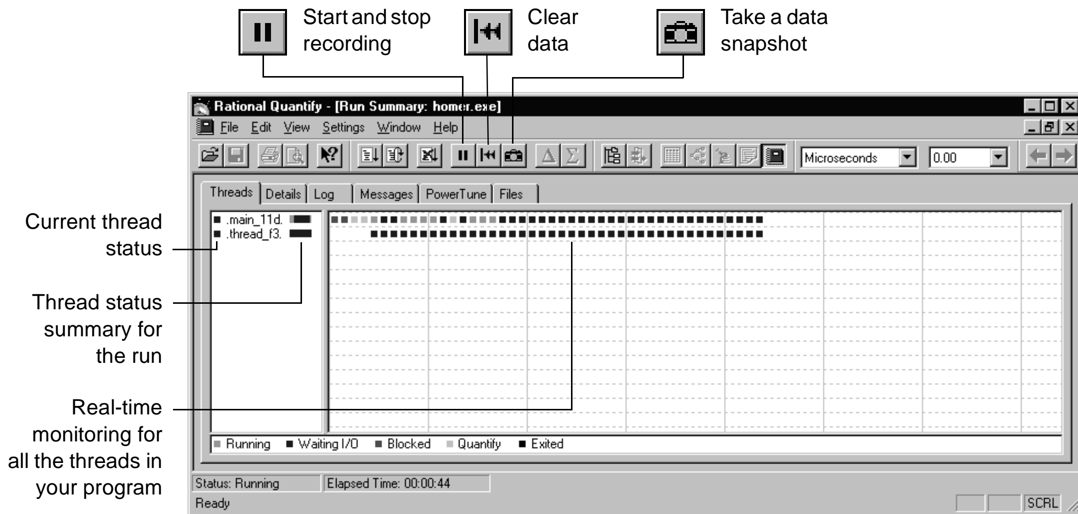


Run your plug-in as usual. Quantify collects and displays profiling data.

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.

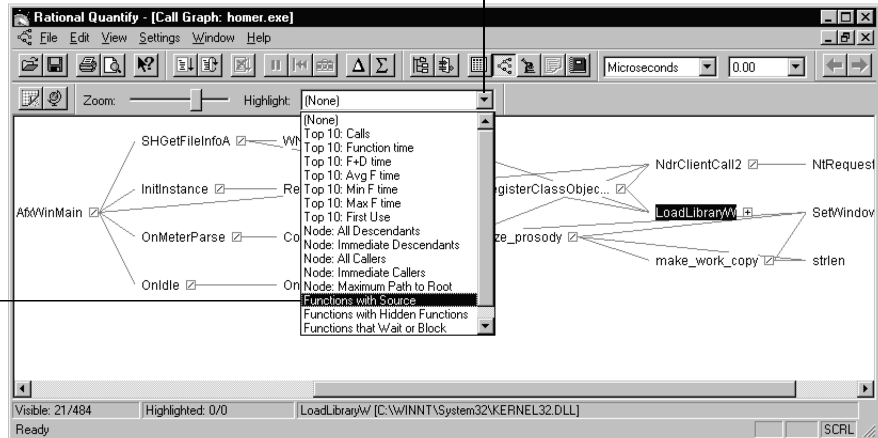
More information? Look up *threads*, *recording data*, and *API functions* in the Quantify online Help index.

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

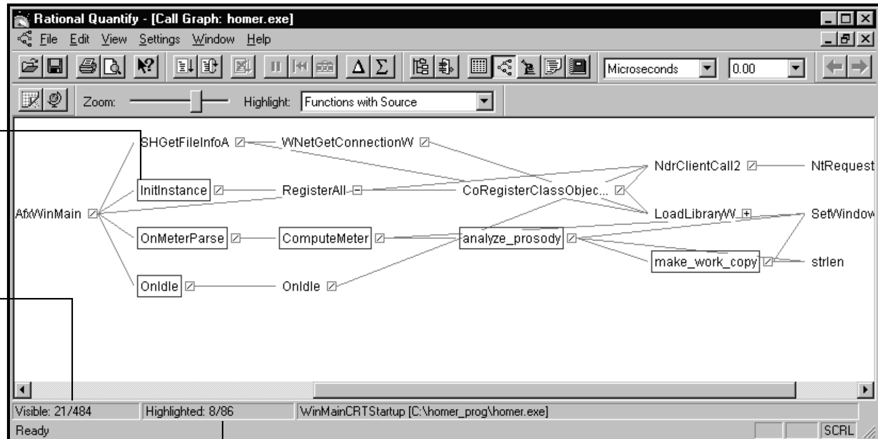
Select **Functions with Source**, for example, to highlight functions that have annotated source



Functions with source code available are enclosed in rectangles

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

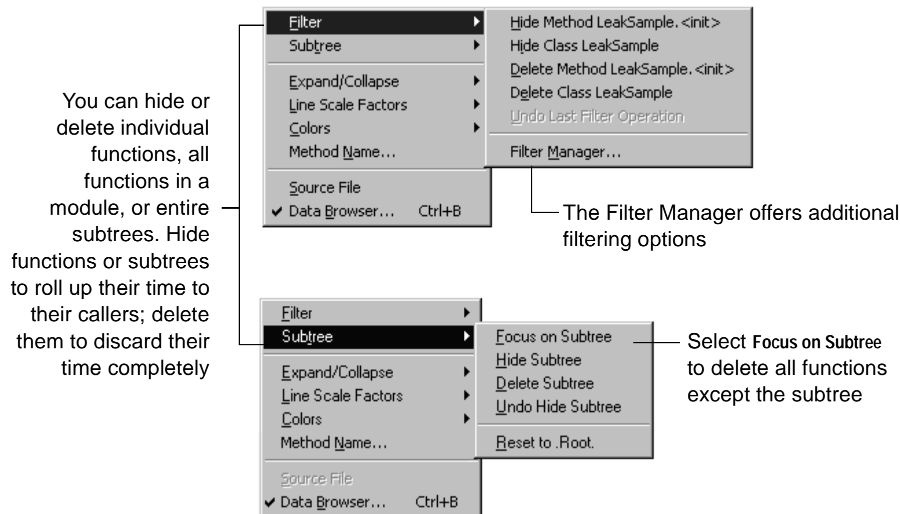
8 of the 86 functions with source code are displayed in the Call Graph



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.



Quantify has undo capabilities for all filter and subtree commands, to easily return to any previous dataset configuration.

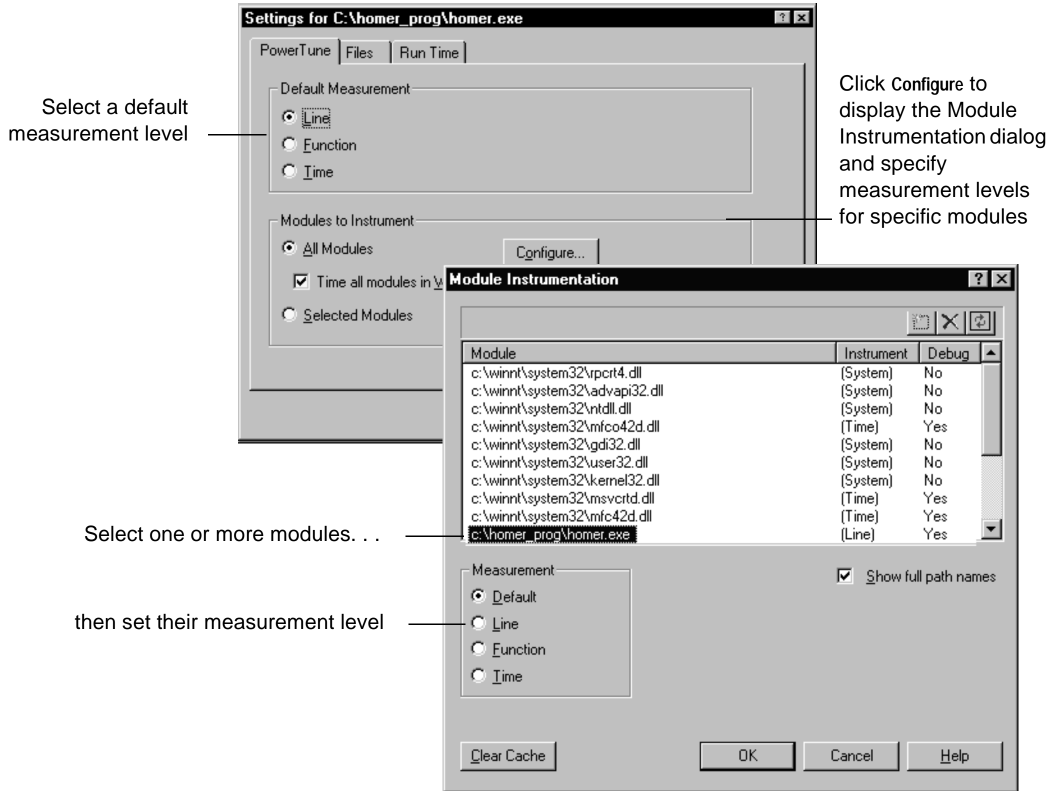
The Call Graph window also provides a series of expand and collapse commands that work with subtrees. Unlike the filter and subtree commands, however, these commands affect only the Call Graph display; they do not change the current dataset.

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.



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 87.

Java, .NET managed code, and Visual Basic programmers: For Java code, the command line must include the /Java switch. 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 a Java class file, change the line that runs it to:

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

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

```
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.

Index

A

ABW error (Purify, C/C++) 15
Annotated Source window
  PureCoverage 62
  Purify (coverage data) 18
  Quantify 83
API functions
  PureCoverage 69
  Purify, C/C++ 24
  Purify, Java 39
  Purify, managed code 54
  Quantify 88
array bounds write error (Purify, C/C++) 15
ASCII text files (.txt)
  PureCoverage 64, 70
  Purify, C/C++ 20
  Purify, Java 38
  Purify, managed code 53
  Quantify 85
Auto Merge (PureCoverage) 63

B

basic steps
  improving code coverage 57
  improving program performance 74
  Purify'ing C/C++ code 9
  Purify'ing Java code 28
  Purify'ing managed code 42
batch files for automated testing
  PureCoverage 70
  Purify, C/C++ 23
  Purify, Java 38
  Purify, managed code 53
  Quantify 92
Break on Error tool (Purify, C/C++) 22

C

cache files
  Purify, C/C++ 11
call graph (Purify, Java)
  filter commands 37
  highlighting related methods 36
  overview 30
  subtree commands 37
call graph (Purify, managed code)
  filter commands 50
  highlighting related methods 50
  overview 44
  subtree commands 50, 51
call graph (Quantify)
  filter commands 90
  for comparing runs 84
  highlighting related functions 89
  initial display 76
  line width 79
  subtree commands 77, 90
call stack (Purify, C/C++) 15, 16
callers of a function, listed (Quantify) 82
calling paths, call graph (Quantify) 79
C/C++ code
  monitoring coverage 58
  profiling performance 73, 75
  Purify'ing 9
.cfy files
  PureCoverage 64
  Purify, coverage data 20
ClearQuest integration
  PureCoverage 66
  Purify, C/C++ 24
  Quantify 86
code
  editing (Purify, C/C++) 17
  editing (Purify, Java) 31
  editing (Purify, managed code) 45

collapsing call graph subtrees
- Purify, Java 37
- Purify, managed code 51
- Quantify 90

colors
- in annotated source (PureCoverage) 62
- in annotated source (Purify, coverage data) 18
- in call graph (Quantify) 84

command-line interface
- Purify, C/C++ 23
- Purify, Java 38
- Purify, managed code 53

commands (Purify, C/C++)
- Embed Data Browsers 22

commands (Purify, Java)
- Expand/Collapse 37
- filter commands 37
- subtree commands 37
- undoing 37

commands (Purify, managed code)
- Expand/Collapse 51
- filter commands 50
- subtree commands 51
- undoing 51

commands (Quantify)
- Expand/Collapse 78, 90
- filter commands 90
- subtree commands 78, 90
- undoing 90

comparing
- program runs (Purify, C/C++) 19
- program runs (Purify, Java) 30
- program runs (Purify, managed code) 44
- program runs (Quantify) 84
- snapshots (Purify, Java) 30
- snapshots (Purify, managed code) 44

Coverage Browser window (PureCoverage) 59

coverage data (PureCoverage)
- controlling with API functions 69
- filtering 61
- limiting collection 68
- merged for multiple runs 63
- saving from the command line 70
- saving from the user interface 63
- sharing 64

coverage data (Purify, C/C++)
- collecting 10, 23
- saving 20

coverage data files (.cfy)
- PureCoverage 64
- Purify 20

coverage levels
- customizing (PureCoverage) 66
- setting default levels (PureCoverage) 66

coverage monitoring (Purify, C/C++)
- /Coverage option 23
- description 8
- saving coverage data 20
- turning on 10
- using coverage data 17–19

Create Filter command (Purify, C/C++) 14

customizing
- coverage levels (PureCoverage) 66
- data collection level (Quantify) 90
- data display (PureCoverage) 60

D

data, *see* coverage data, error data, memory profiling data, *and* performance data

Data Browser window (Purify)
- coverage data (C/C++) 18
- error data (C/C++) 11–19
- memory profiling data (Java) 29–31
- memory profiling data (managed code) 43–46
- object list (Java) 34
- object list (managed code) 48

data recording
- changing default level (Quantify) 90
- controlling (Quantify) 88
- controlling programatically (Purify, Java) 39
- controlling programatically (Purify, managed code) 54

debug data
- and instrumentation (Purify, C/C++) 10, 20, 42

and line-level coverage (PureCoverage) 58
and line-level profiling (Quantify) 75
debugging, just-in-time (Purify, C/C++) 22
default instrumentation levels, setting
PureCoverage 66
Purify, C/C++ 20
Quantify 90
deleting call graph subtrees
Purify, Java 37
Purify, managed code 51
Quantify 80, 90
descendants of a function, listed (Quantify) 82
diff call graph (Quantify) 84
diff function list (Quantify) 84
diff'ing snapshots
equivalent results with API (Purify, Java) 39
equivalent results with API (Purify, managed code) 54
Purify, Java 30
Purify, managed code 44
displaying filtered messages (Purify, C/C++) 15
dispose() method (Purify, Java) 27

E

editing source code
Purify, C/C++ 17
Purify, Java 31
Purify, managed code 45
Embed Data Browsers command (Purify, C/C++) 22
Error View tab, Data Browser window (Purify, C/C++) 11
errors (Purify, C/C++)
analyzing 15
breaking on errors 22
correcting 17
saving error data 20
See also messages (Purify, C/C++)
excluding modules (PureCoverage) 67
exit messages (Purify, C/C++) 12
expanding call graph subtrees
Purify, Java 37
Purify, managed code 51
Quantify 78, 90

F

F+D (Function + Descendants) time (Quantify) 80
File View tab (Purify, coverage data) 18
files
caching after instrumentation (Purify, C/C++) 11
.cfy (PureCoverage) 64
.cfy (Purify, C/C++) 20
.pcy (Purify, C/C++) 20
.pft (Purify, C/C++) 15
.pfy (Purify, C/C++) 20
.pmy (Purify, Java) 36
.pmy (Purify, managed code) 49
.txt (PureCoverage) 64
.txt (Purify, C/C++) 20
.txt (Purify, Java) 36
.txt (Purify, managed code) 49
filters
filter groups (Purify, C/C++) 15
Filter Manager (PureCoverage} 61
Filter Manager (Purify, C/C++) 15
Filter Manager (Purify, Java) 38
Filter Manager (Purify, managed code) 52
Filter Manager (Quantify) 90
overview (Purify, C/C++) 36
overview (Purify, Java) 50
overview (Purify, managed code) 13
saved in .pft files (Purify, C/C++) 15
sharing (Purify, C/C++) 15
undoing filter commands (Purify, Java) 37
undoing filter commands (Purify, managed code) 51
undoing filter commands (Quantify) 90
focusing on subtrees
Purify, Java 51
Purify, managed code 90

Function Detail window
Purify, Java 32
Purify, managed code 46
Quantify 82
Function level profiling (Quantify) 91
function list view
Purify, coverage data 18
Purify, Java 31
Purify, managed code 45
Function List window
for a single run (Quantify) 80
for comparing runs (Quantify) 84
sorting data (Quantify) 80
using (PureCoverage) 60
function time (Quantify) 80
function-level coverage
described (PureCoverage) 59
setting (PureCoverage) 66
function-level instrumentation (Purify) 20
functions
PureCoverage API 69
Purify API (C/C++) 24
Purify API (Java) 39
Purify API (managed code) 54
Quantify API 88

G

garbage collector
Purify, Java 26, 29
Purify, managed code 40, 43
graphs
call graph (Purify, Java) 30
call graph (Purify, managed code) 44
call graph (Quantify) 76, 84, 89
memory usage graph (Purify, Java) 29
memory usage graph (Purify, managed code) 43
object reference (Purify, Java) 34
object reference (Purify, managed code) 47
green highlighting in call graph (Quantify) 84
groups, filter (Purify, C/C++) 15

H

handles in use at exit (Purify, C/C++) 12
hiding call graph subtrees
Purify, Java 37
Purify, managed code 51
Quantify 90
hiding Purify C/C++ error messages
See filters
highlighting
green in call graphs (Quantify) 84
performance improvements (Quantify) 84
related functions (Quantify) 89
related methods (Purify, Java) 36
related methods (Purify, managed code) 50

I

instrumentation
customizing (PureCoverage) 66
customizing (Purify, C/C++) 21
default levels (Purify, C/C++) 20
described (PureCoverage) 58
described (Purify, C/C++) 10
selective (PureCoverage) 67
selective (Quantify) 87
integration
Microsoft Visual Basic (PureCoverage) 65
Microsoft Visual Basic (Quantify) 85
Microsoft Visual Studio 6 (PureCoverage) 65
Microsoft Visual Studio 6 (Purify, C/C++) 9–20
Microsoft Visual Studio 6 (Quantify) 85
Microsoft Visual Studio .NET (PureCoverage) 65
Microsoft Visual Studio .NET (Purify) 42
Microsoft Visual Studio .NET (Quantify) 85
Rational ClearQuest (PureCoverage) 65
Rational ClearQuest (Purify, C/C++) 24
Rational ClearQuest (Quantify) 85
Rational Robot (PureCoverage) 65
Rational Robot (Purify, C/C++) 24–25
Rational Robot (Quantify) 85
Rational Visual Test (PureCoverage) 65

Rational Visual Test (Purify, C/C++) 24– 25
Rational Visual Test (Quantify) 85
interactive snapshots (PureCoverage) 69

J

Java (PureCoverage)
running from the command line 70
supported languages 55
Java (Purify)
examining objects 33– 35
filtering memory profiling data 37
memory leaks 26, 28
memory usage graph 29
Purify'ing Java code 28
saving memory profiling data 36
Java (Quantify)
running from the command line 92
supported languages 73
/Java option
PureCoverage 70
Purify 38
Quantify 92
just-in-time debugging (Purify, C/C++) 22

L

L+D (Line + Descendants) time (Quantify) 83
languages and applications supported
PureCoverage 56
Purify, C/C++ 8
Quantify 73
leaks (Purify)
Java 26
managed code 40
See also memory leaks (Purify)
levels of measurement (Quantify) 91
limiting coverage data collection (PureCoverage) 68
line colors
in annotated source (PureCoverage) 62
in annotated source (Purify coverage data) 18
Line level measurement (Quantify) 91
Line time (Quantify) 83
line width, in call graph (Quantify) 79
line-level coverage (PureCoverage)
annotated source 62
described 59
setting 66
line-level instrumentation (Purify, C/C++) 20

M

makefiles for automated testing
PureCoverage 70
Purify, C/C++ 23
Purify, Java 38
Purify, managed code 53
Quantify 92
managed code (PureCoverage)
running from the command line 70
supported languages 55
managed code (Purify)
examining objects 47– 49
filtering memory profiling data 50
memory leaks 40, 42
memory usage graph 43
Purify'ing managed code 42
saving memory profiling data 49
managed code (Quantify)
supported languages 73
measurement levels (Quantify) 91
memory leaks (Purify)
C/C++ leaks reported at exit 12
Java memory leaks 26, 28
managed code memory leaks 40, 42
PurifyNewLeaks API function (C/C++) 24
memory profiling data (Purify)
filtering (Java) 37
filtering (managed code) 50, 52
saving (Java) 36
saving (managed code) 49
memory usage graph
Purify, Java 29
Purify, managed code 43
menu, shortcut (Purify, C/C++) 12

merging data from multiple runs (PureCoverage) 63
messages (Purify, C/C++)
analyzing 15
expanding 15
filtering 13
redisplaying filtered 15
See also errors (Purify, C/C++)
method-level coverage (PureCoverage), *see* function-level coverage (PureCoverage)
methods, highlighting by category
Purify, Java 36
Purify, managed code 50
Microsoft Visual Studio 6 integration
PureCoverage 65
Purify 9
Quantify 85
Microsoft Visual Studio .NET integration
PureCoverage 65
Purify 42
Quantify 85
minimal instrumentation (Purify, C/C++) 20
Module View tab (Purify coverage data) 18
modules
controlling coverage levels (PureCoverage) 66
controlling instrumentation (Purify, C/C++) 21
controlling instrumentation level (Quantify) 90
excluding from coverage (PureCoverage) 67
filtering by module (Purify, Java) 37
filtering by module (Purify, managed code) 50
filtering by module (Quantify) 90
monitoring program performance (Quantify) 88
monitoring program runs (Quantify) 88

N

Navigator
PureCoverage 63
Purify, C/C++ 19
Purify, Java 30
Purify, managed code 44
negative values in function list (Quantify) 84
.NET managed code, *see* managed code
/Net option
PureCoverage 70
Purify 53
Quantify 92

O

Object Detail window
Purify, Java 34
Purify, managed code 47
Object List View tab
Purify, Java 34
Purify, managed code 48
object reference graph
Purify, Java 47
Purify, managed code 34
object references
and Java memory leaks 26
and managed code memory leaks 40
objects, examining
Purify, Java 33– 35
Purify, managed code 47– 49

P

.pcy files (Purify, C/C++) 20
performance data (Quantify)
comparing runs 84
controlling recording 88
filtering 90
for all dataset functions 80
for individual lines 83
for single functions 82
improvements highlighted 84
saving from the command line 92
Perl scripts for automated testing
PureCoverage 70
Purify, C/C++ 23
Purify, Java 38
Purify, managed code 53
Quantify 92

.pft files (Purify, C/C++) 15
.pfy files (Purify, C/C++) 20
pie charts, Function Detail window
  Purify, Java 33
  Purify, managed code 47
.pmy files
  Purify, Java 36
  Purify, managed code 49
PowerCheck tab (Purify, C/C++) 20
PowerCov options (PureCoverage) 66
PowerTune (Quantify) 90
precise instrumentation (Purify, C/C++) 20
problems
  Java code 26
  managed code 40
procedure-level coverage (PureCoverage), *see* function-level coverage (PureCoverage)
profiling program performance (Quantify) 88
programming languages and components supported
  PureCoverage 56
  Purify 7
  Purify, C/C++ 8
  Quantify 73
programs
  instrumenting (PureCoverage) 58
  profiling performance (Quantify) 88
  rerunning (Purify) 19
  running from Microsoft Visual Studio 6 (PureCoverage) 65
  running from Microsoft Visual Studio 6 (Purify, C/C++) 10
  running from Microsoft Visual Studio 6 (Quantify) 85
  running from Microsoft Visual Studio .NET (PureCoverage) 65
  running from Microsoft Visual Studio .NET (Purify, managed code) 42
  running from Microsoft Visual Studio .NET (Quantify) 85
  running Java programs (Purify) 28
  running managed code programs (Purify) 42
  running under debugger (Purify, C/C++) 22
PureCoverage
  in PurifyPlus 1
  tips for developers 2
  tips for testers 3
  using 57
Purify
  in PurifyPlus 1
  tips for developers 2
  tips for testers 3
  using (C/C++) 9
  using (Java) 28
  using (managed code) 42
Purify data files
  C/C++ 20
  Java 36
  managed code 49
Purify'ing
  C+C++ code 9
  Java code 28
  managed code 42
PurifyPlus, described 1

Q

Quantify
  in PurifyPlus 1
  tips for developers 2
  tips for testers 3
  using 74
QuickFilter command (Purify, C/C++) 14

R

Rational ClearQuest integration
  PureCoverage 66
  Purify 24
  Quantify 86
Rational PureCoverage
  in PurifyPlus 1
  tips for developers 2
  tips for testers 3
  using 57

Rational Purify
- in PurifyPlus 1
- tips for developers 2
- tips for testers 3
- using (C/C++) 9
- using (Java) 28
- using (managed code) 42

Rational PurifyPlus, described 1
Rational Quantify
- in PurifyPlus 1
- tips for developers 2
- tips for testers 3
- using 74

Rational Robot integration
- PureCoverage 66, 71
- Purify, C/C++ 24–25
- Quantify 86

Rational Software technical publications, contacting 5
Rational Software technical support, contacting 5
Rational Visual Test integration
- PureCoverage 66, 71
- Purify, C/C++ 24–25
- Quantify 86

recording data, controlling
- PureCoverage 68
- Quantify 88

relocation data, and instrumentation
- Purify, C/C++ 10, 20, 42

Robot integration
- PureCoverage 66, 71
- Purify, C/C++ 24–25
- Quantify 86

Run Control toolbar (Quantify) 88
Run Summary window
- PureCoverage 59
- Quantify 88

running programs
- from the command line (Purify) 53
- from the command line (Purify, C/C++) 23
- from the command line (Purify, Java) 38
- from Visual Studio 6 (PureCoverage) 65
- from Visual Studio 6 (Purify, C/C++) 10
- from Visual Studio 6(Quantify) 85
- from Visual Studio .NET (PureCoverage) 65
- from Visual Studio .NET (Purify, managed code) 42
- from Visual Studio .NET (Quantify) 85
- in the Purify standalone interface (Purify) 52
- in the Purify standalone interface (Purify, C/C++) 22
- PureCoverage 58
- Purify, Java 28
- Purify, managed code 42
- Quantify 75
- rerunning (Purify, C/C++) 19

runs, comparing
- Purify, C/C++ 19
- Purify, Java 30
- Purify, managed code 44
- Quantify 84

S

/Save* options
- PureCoverage 70
- Purify, C/C++ 23
- Purify, Java 38
- Purify, managed code 53
- Quantify 92

saving data
- from the command line (PureCoverage) 70, 92
- from the command line (Quantify) 92
- from the command line(Purify, C/C++) 23, 38, 53
- from the user interface (PureCoverage) 63
- from the user interface (Purify, C/C++) 20
- from the user interface(Purify, Java) 36
- from the user interface(Purify, managed code) 49

scaling of line widths, in Quantify call graph 79
scripts for automated testing
- PureCoverage 70
- Purify, C/C++ 23
- Purify, Java 38
- Purify, managed code 53
- Quantify 92

selective instrumentation
- PureCoverage 67
- Quantify 87

settings for data collection
- PureCoverage 58
- Quantify 75

sharing
- data files (PureCoverage) 64
- filters (Purify, C/C++) 15

shortcut menu
- Purify, C/C++ 37
- Purify, Java 12
- Purify, managed code 51

snapshots
- coverage dava (PureCoverage) 69
- memory use (Purify, Java) 29
- memory use (Purify, managed code) 43

sorting data
- PureCoverage 60
- Quantify 80

source code
- displaying (PureCoverage) 62
- displaying (Quantify) 83
- editing (Purify, C/C++) 17
- editing (Purify, Java) 31
- editing (Purify, managed code) 45

stack, call (Purify, C/C++) 16

standalone Purify interface (C/C++) 22

standalone Purify interface (managed code) 52

starting
- PureCoverage 58
- Purify, C/C++ 10, 22, 52
- Purify, Java 28
- Purify, managed code 42
- Quantify 75

status line, Quantify windows 89

strategies for using Rational PureCoverage 56

subtrees (Purify, Java)
- deleting 37
- expanding and collapsing 37
- focusing on 37
- undoing subtree commands 37

subtrees (Purify, managed code)
- deleting 51
- expanding and collapsing 51
- focusing on 51
- undoing subtree commands 51

subtrees (Quantify call graph)
- deleting 80, 90
- expanding and collapsing 90
- focusing on 90
- undoing subtree commands 90

supported languages and components
- PureCoverage 56
- Purify, C/C++ 8
- Quantify 73

system resources and memory leaks
- Purify, Java 27
- Purify, managed code 41

T

technical publications, contacting 5

technical support, contacting 5

tests
- using PureCoverage in automated tests 70
- using PureCoverage in unit tests 3
- using Purify in automated tests 23, 25
- using Purify in automated tests (Java) 38
- using Purify in unit tests 3
- using Quantify in automated tests 92
- using Quantify in unit tests 3

text files (.txt)
- PureCoverage 64, 70
- Purify, C/C++ 20
- Purify, Java 36
- Purify, managed code 49
- Quantify 85

thread status, monitoring (Quantify) 88

Time measurement (Quantify) 92

tool tips, call graph
- Purify, Java 31
- Purify, managed code 45

.txt files
- PureCoverage 64, 70
- Purify. Java 36
- Purify, C/C++ 20
- Purify, managed code 49
- Quantify 85

U

undoing filter and subtree commands
  Purify, Java 37
  Purify, managed code 51
  Quantify 90
unembedding Purify (C/C++) 22

V

Visual Basic
  integration (PureCoverage) 65
  integration (Quantify) 85
Visual C/C++, running programs
  PureCoverage 58
  Purify 9
  Quantify 75
Visual Studio 6 integration
  PureCoverage 65
  Purify, C/C++ 9
  Quantify 85
Visual Studio .NET integration
  PureCoverage 65
  Purify 42
  Quantify 85
Visual Test integration
  PureCoverage 66, 71
  Purify, C/C++ 24– 25
  Quantify 86

W

what-ifs, in Quantify call graph 80
windows and tabs
  Annotated Source (PureCoverage) 62
  Annotated Source (Quantify) 83
  Call Graph (Purify, Java) 30, 36
  Call Graph (Purify, managed code) 44, 50
  Call Graph (Quantify) 76, 84, 89
  Coverage Browser (PureCoverage) 59
  Data Browser (Purify, C/C++) 11– 14, 15– 18
  Data Browser (Purify, Java) 29– 31, 34
  Data Browser (Purify, managed code) 43– 46, 48
  Diff Call Graph (Quantify) 84
  Diff Function List (Quantify) 84
  File View (Purify, coverage data) 18
  Function Detail (Purify, Java) 32, 33
  Function Detail (Purify, managed code) 46, 47
  Function Detail (Quantify) 82
  Function List (PureCoverage) 60
  Function List (Quantify) 80, 84
  Function List View (Purify Coverage data) 18
  Function List View (Purify, Java) 31
  Function List View (Purify, managed code) 45
  Module View (Purify, coverage data) 18
  Navigator (Purify, C/C++) 19
  Navigator (Purify, Java) 30
  Navigator (Purify, managed code) 44
  Object Detail (Purify, Java) 34
  Object Detail (Purify, managed code) 47
  Object List View (Purify, Java) 34
  Object List View (Purify, managed code) 48
  Run Summary (PureCoverage) 59
  Run Summary (Quantify) 88