Rational the e-development company™

WINDOWS

Getting Started with Rational® PurifyPlus

PART NUMBER: 800-024651-000

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

Rational®
the e-development company™

COPYRIGHT NOTICE

U.S. GOVERNMENT RIGHTS NOTICE

TRADEMARK NOTICE

U.S. PATENT NOTICE

Printed in the U.S.A.

Contents

Welcome to Rational PurifyPlus

Rational PurifyPlus: What it does

Rational® PurifyPlus brings together three essential tools that help 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 pinpointing 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 any language that Microsoft Visual Studio.NET supports.

If you're developing code in Visual Studio, invoke the PurifyPlus tools from the Visual Studio menus. 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, 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 start of 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.

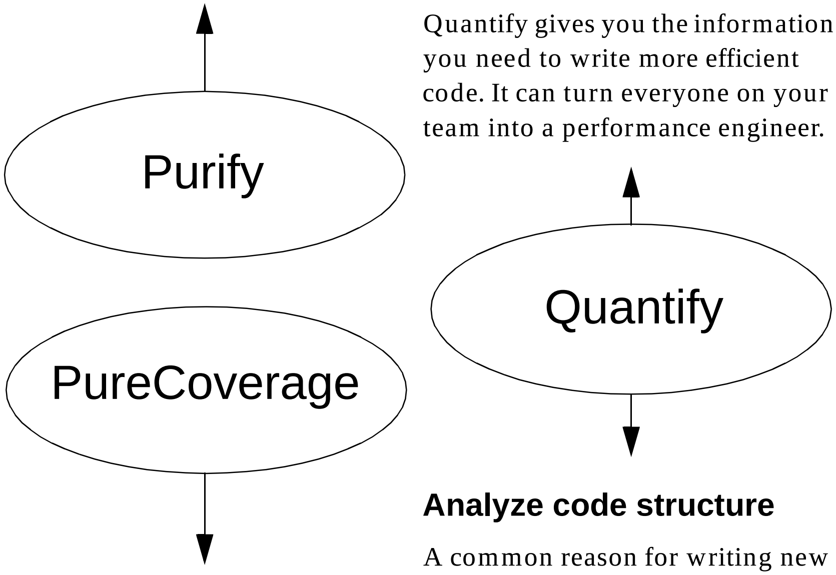
Find memory errors early

Use Purify with Visual Studio to pinpoint hard-to-find bugs. Memory errors don’t always show up right away, but they’re the ones that will make your program crash someday.

Prevent performance bottlenecks

Whenever you write new code or modify existing code, use Quantify right away 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.



Improve code coverage

You haven’t Purify’d® code you haven’t run. Use PureCoverage from within Purify to make sure you're exercising all your code during pre-checkin testing—just click **Coverage, error, and leak data** in Purify’s Run Program dialog.

PureCoverage can tell you if your tests are covering your code sufficiently for Purify to find all the memory errors.

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?

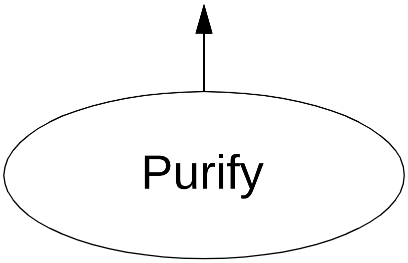
Use Quantify not only to find performance bottlenecks, but also to learn more about how your code is structured. It will help you to make effective performance improvements.

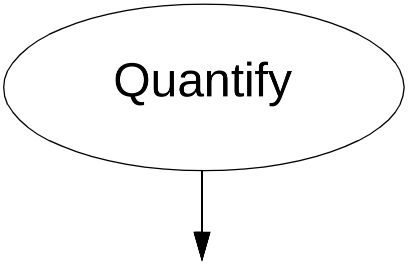
Tips for test engineers

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 errors that your external functionality tests can't uncover.



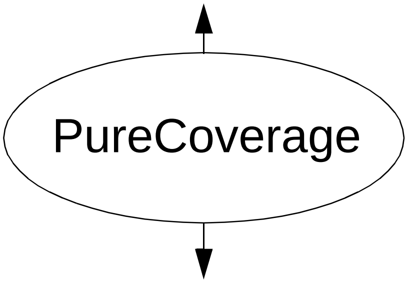


If performance improves . . .

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.

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 all your 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.

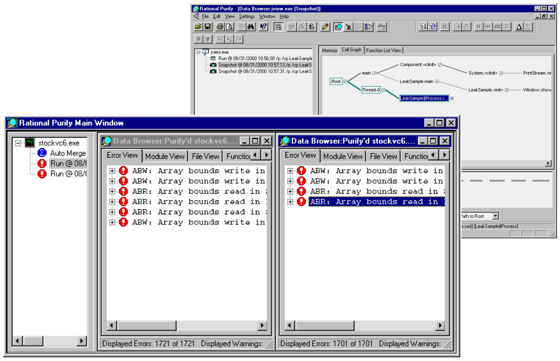
If performance drops . . .

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 use Purify to automatically pinpoint hard-to-find bugs in C/C++ and Java code, read the rest of this manual



Rational Purify Main Window

Data Browser:Purify'd stockvc6....

Error View | Module View | File View | Functio

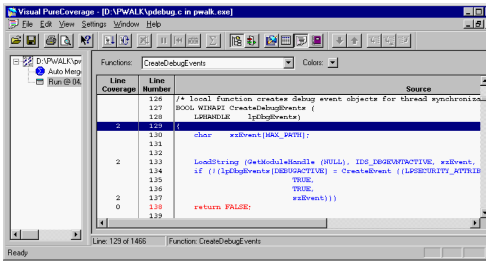
```
ABW: Array bounds write in
ABW: Array bounds write in
ABR: Array bounds read in
ABR: Array bounds read in
ABR: Array bounds read in
ABW: Array bounds write in
```

Displayed Errors: 1721 of 1721 Displayed Warnings:

```
ABW: Array bounds write in
ABW: Array bounds write in
ABR: Array bounds read in
ABR: Array bounds read in
```

Displayed Errors: 1701 of 1701 Displayed Warnings:

To avoid shipping untested code, read *Getting Started with Rational PureCoverage*



Visual PureCoverage - [D:\PWALK\pdebug.c in pwalk.exe]

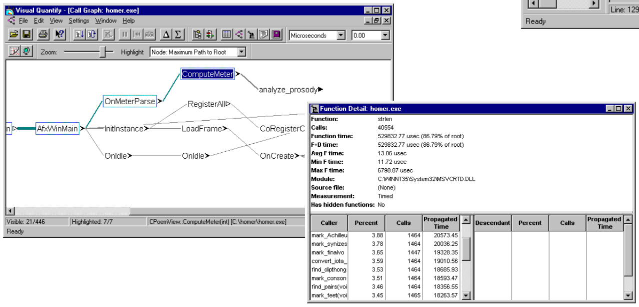
Functions: CreateDebugEvents

Line Coverage	Line Number	Source
	126	`/* local function creates debug event objects for thread synchroniza`
	127	`BOOL WINAPI CreateDebugEvents (`
	128	`    LPHANDLE    lpDbgEvents)`
2	129	`{`
	130	`    char    szEvent[MAX_PATH];`
	131	
	132	
2	133	`    LoadString (GetModuleHandle (NULL), IDS_DBGEVNTACTIVE, szEvent,`
	134	`    if (!(lpDbgEvents[DEBUGACTIVE] = CreateEvent ((LPSECURITY_ATTRIB`
	135	`                                TRUE,`
	136	`                                TRUE,`
2	137	`                                szEvent)))`
0	138	`    return FALSE;`
	139	

Line: 129 of 1466 Function: CreateDebugEvents

Ready

To highlight performance bottlenecks, read *Getting Started with Rational Quantify*



Visual Quantify - [Call Graph: homer.exe]

Function Detail: homer.exe

Function: strlen
Calls: 40554
Function time: 529832.77 usec (86.79% of root)
F+D time: 529832.77 usec (86.79% of root)
Avg F time: 13.06 usec
Min F time: 11.72 usec
Max F time: 6798.87 usec
Module: C:\WINNT35\System32\MSVCRTD.DLL
Source file: (None)
Measurement: Timed
Has hidden functions: No

Caller	Percent	Calls	Propagated Time
mark_Achilleu	3.88	1464	20573.45
mark_synizes	3.78	1464	20036.25
mark_finalvo	3.65	1447	19328.35
convert_iota_	3.59	1464	19010.56
find_dipthong	3.53	1464	18685.93
mark_conson	3.51	1464	18593.47
find_pairs(voi	3.46	1464	18356.55
mark_feet(voi	3.45	1465	18263.57

Descendant	Percent	Calls	Propagated Time

Visible: 21/446 Highlighted: 7/7 CPoemView::ComputeMeter(int) [C:\homer\homer.exe]

Ready

The online Help for Purify, Quantify, and PureCoverage contains 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.

Introducing Rational Purify

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

For Visual C/C++ developers and testers

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 pinpoints untested code

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, 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:

- DLLs, including Windows DLLs and Microsoft Foundation Class Library DLLs
- 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. With PurifyPlus, 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 be 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 early and often

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 early and often, you'll release clean, reliable products—on time.

More information? PurifyPlus tools help you improve not only your application's reliability, but also its performance. Using Quantify, you can pinpoint and eliminate the bottlenecks that prevent your application from operating at its greatest potential speed. Read *Getting Started with Rational Quantify* for an introduction to Quantify's features.

For Java developers and testers

Java memory leaks?

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

The Java 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 occur, for example, when you:

- add objects to arrays and forget about them.
- do not release 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.

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, Quantify and PureCoverage. Quantify can help you find the bottlenecks that slow down your code, and PureCoverage can show you the areas in your code that your tests are not reaching. Read *Getting Started with Rational Quantify* and *Getting Started with Rational PureCoverage* to see how these tools can help you.

Getting started with Purify: C/C++ code

The basic steps

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

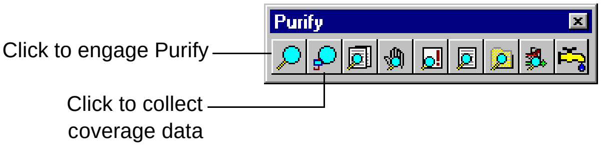
1. Run your program with Purify to collect:
   - Error data
   - Code coverage data
2. Analyze the error data and correct your source code.
3. If you've collected coverage data, analyze it to find any parts of your code that you have not Purify'd®.
4. Rerun your program with Purify.

This chapter shows you how to use Purify in Microsoft Visual Studio. But you can also use Purify independently of Visual Studio. Read "Using Purify standalone" on page 31 of this guide, and "Testing with Purify's command-line interface" on page 32.

Running a C/C++ program with Purify

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

Set Purify to collect coverage data, as well as checking for errors and memory leaks.

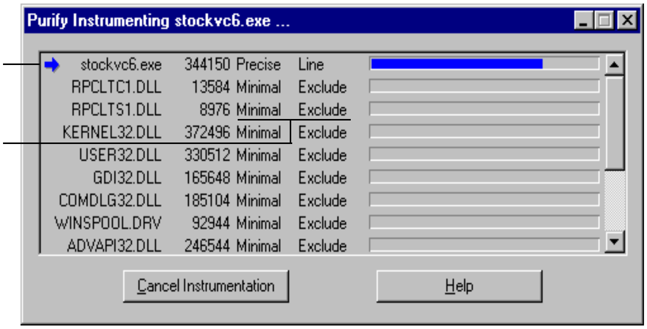


Build and execute your program using commands from the Visual Studio **Build** menu. To get the maximum level of detail in Purify error and coverage data, build your program with debug and relocation data.

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

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

Purify reports its progress as it instruments each module.



The module that Purify is currently instrumenting

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

Purify Instrumenting stockvc6.exe ...

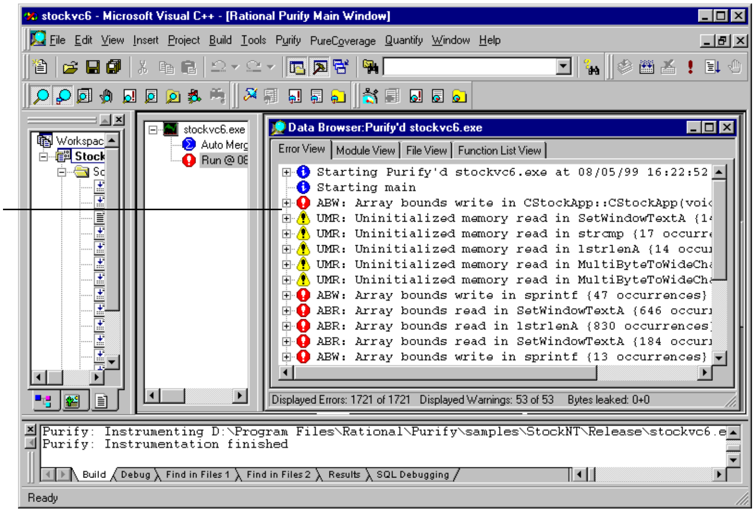
➔ stockvc6.exe	344150	Precise	Line
RPCLTC1.DLL	13584	Minimal	Exclude
RPCLTS1.DLL	8976	Minimal	Exclude
KERNEL32.DLL	372496	Minimal	Exclude
USER32.DLL	330512	Minimal	Exclude
GDI32.DLL	165648	Minimal	Exclude
COMDLG32.DLL	185104	Minimal	Exclude
WINSPOOL.DRV	92944	Minimal	Exclude
ADVAPI32.DLL	246544	Minimal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

Purify instruments each module at a default instrumentation level, but you can customize the instrumentation level to provide more or less detail for special cases.

More information? For an explanation of instrumentation levels and how to use them, read "Customizing instrumentation" on page 29 of this guide. For more detail, look up *instrumenting* in the Purify online Help index.

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

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



Purify Error View tab, Data Browser window

stockvc6 - Microsoft Visual C++ - [Rational Purify Main Window]
File Edit View Insert Project Build Tools Purify PureCoverage Quantify Window Help
Workspac
Stock
Sc
stockvc6.exe
Auto Merg
Run @ 08
Data Browser:Purify'd stockvc6.exe
Error View
Module View
File View
Function List View
Build
Debug
Find in Files 1
Find in Files 2
Results
SQL Debugging
Ready

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

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

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

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 31 of this guide.

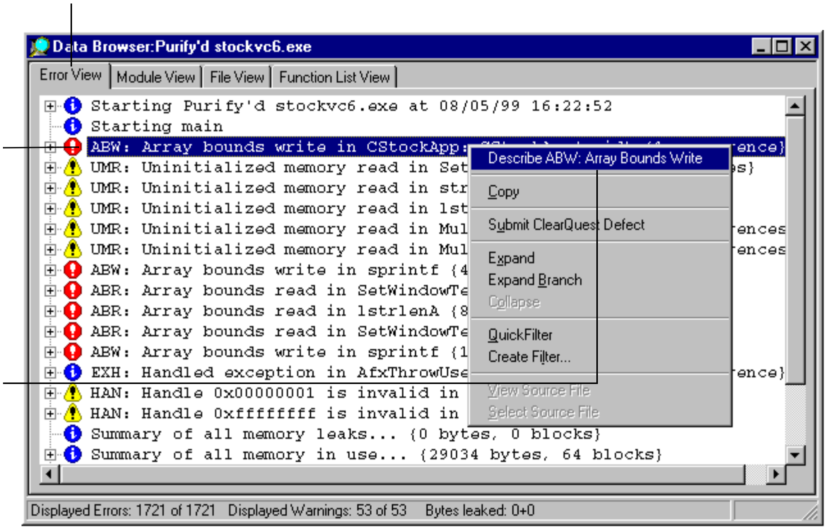
Seeing all your errors at a glance

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

Color-coded icons show message severity:
ⓘ informational ⚠ warning ❗ error

Acronyms like ABW identify message type

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



```
Starting Purify'd stockvc6.exe at 08/05/99 16:22:52
Starting main
ABW: Array bounds write in CStockApp:                   ence}
UMR: Uninitialized memory read in Set                   s}
UMR: Uninitialized memory read in str
UMR: Uninitialized memory read in lst
UMR: Uninitialized memory read in Mul                   ences
UMR: Uninitialized memory read in Mul                   ences
ABW: Array bounds write in sprintf {4
ABR: Array bounds read in SetWindowTe
ABR: Array bounds read in lstrlenA {8
ABR: Array bounds read in SetWindowTe
ABW: Array bounds write in sprintf {1
EXH: Handled exception in AfxThrowUse                   ence}
HAN: Handle 0x00000001 is invalid in
HAN: Handle 0xffffffff is invalid in
Summary of all memory leaks... {0 bytes, 0 blocks}
Summary of all memory in use... {29034 bytes, 64 blocks}
```

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

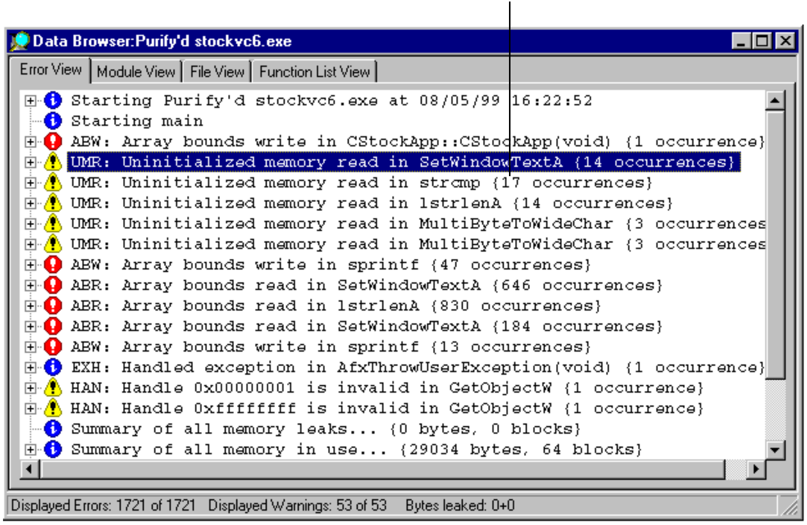
When you exit the program, Purify reports memory leaks. In addition to memory leaks, you can set Purify to report memory in use at exit and handles in use at exit.

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

When identical errors repeat

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

This uninitialized memory
read (UMR) occurred 17 times



```
Starting Purify'd stockvc6.exe at 08/05/99 16:22:52
Starting main
ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}
UMR: Uninitialized memory read in SetWindowTextA {14 occurrences}
UMR: Uninitialized memory read in strcmp {17 occurrences}
UMR: Uninitialized memory read in lstrlenA {14 occurrences}
UMR: Uninitialized memory read in MultiByteToWideChar {3 occurrences
UMR: Uninitialized memory read in MultiByteToWideChar {3 occurrences
ABW: Array bounds write in sprintf {47 occurrences}
ABR: Array bounds read in SetWindowTextA {646 occurrences}
ABR: Array bounds read in lstrlenA {830 occurrences}
ABR: Array bounds read in SetWindowTextA {184 occurrences}
ABW: Array bounds write in sprintf {13 occurrences}
EXH: Handled exception in AfxThrowUserException(void) {1 occurrence}
HAN: Handle 0x00000001 is invalid in GetObjectW {1 occurrence}
HAN: Handle 0xffffffff is invalid in GetObjectW {1 occurrence}
Summary of all memory leaks... {0 bytes, 0 blocks}
Summary of all memory in use... {29034 bytes, 64 blocks}
```

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

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.

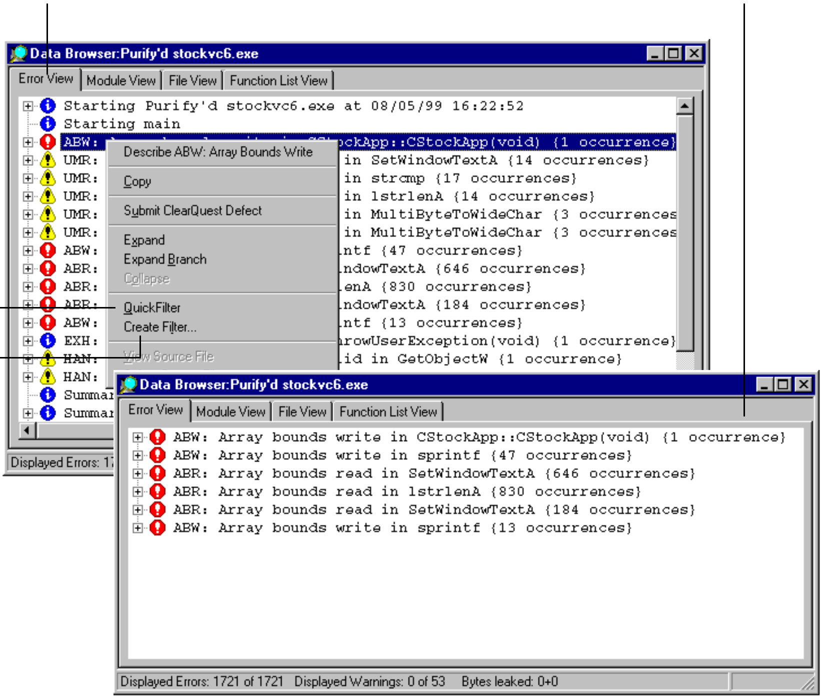
You can filter messages individually, or you can filter them based on their type and source. Consider hiding all informational messages, for example, or all messages originating from a specific file.

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



```
Starting Purify'd stockvc6.exe at 08/05/99 16:22:52
Starting main
ABW:                          ockApp::CStockApp(void) {1 occurrence}
UMR:                          in SetWindowTextA {14 occurrences}
UMR:                          in strcmp {17 occurrences}
UMR:                          in lstrlenA {14 occurrences}
UMR:                          in MultiByteToWideChar {3 occurrences
UMR:                          in MultiByteToWideChar {3 occurrences
ABW:                          ntf {47 occurrences}
ABR:                          ndowTextA {646 occurrences}
ABR:                          enA {830 occurrences}
ABR:                          ndowTextA {184 occurrences}
ABW:                          ntf {13 occurrences}
EXH:                          rowUserException(void) {1 occurrence}
HAN:                          id in GetObjectW {1 occurrence}
HAN:
Summar
Summar
```

Displayed Errors: 17

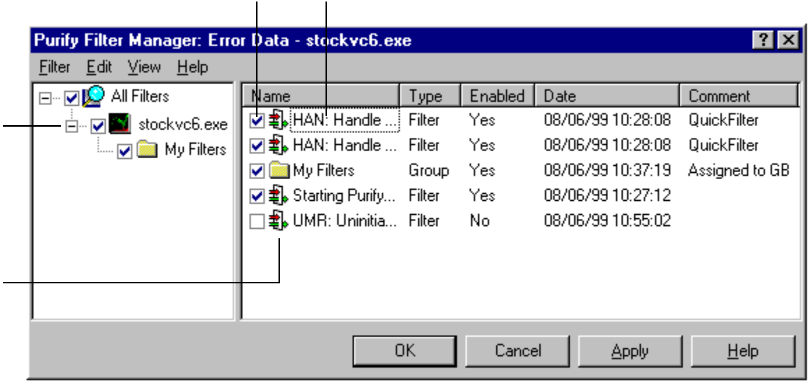
```
ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}
ABW: Array bounds write in sprintf {47 occurrences}
ABR: Array bounds read in SetWindowTextA {646 occurrences}
ABR: Array bounds read in lstrlenA {830 occurrences}
ABR: Array bounds read in SetWindowTextA {184 occurrences}
ABW: Array bounds write in sprintf {13 occurrences}
```

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

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.



Click to enable or disable filters

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

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

Drag and drop filters to move or copy them

Purify Filter Manager: Error Data - stockvc6.exe

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

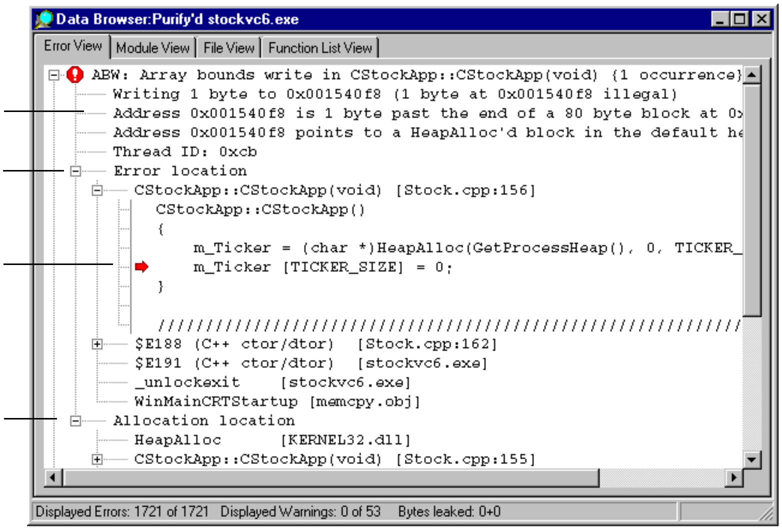
More information? Purify provides filters for coverage data as well as for error data. Look up *filtering data* in the Purify online Help index.

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 29 of this guide.

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:



```
ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}
    Writing 1 byte to 0x001540f8 (1 byte at 0x001540f8 illegal)
    Address 0x001540f8 is 1 byte past the end of a 80 byte block at 0x
    Address 0x001540f8 points to a HeapAlloc'd block in the default he
    Thread ID: 0xcb
    Error location
        CStockApp::CStockApp(void) [Stock.cpp:156]
            CStockApp::CStockApp()
            {
                m_Ticker = (char *)HeapAlloc(GetProcessHeap(), 0, TICKER_
                m_Ticker [TICKER_SIZE] = 0;
            }

            ///////////////////////////////////////////////////////////////
        $E188 (C++ ctor/dtor)  [Stock.cpp:162]
        $E191 (C++ ctor/dtor)  [stockvc6.exe]
        _unlockexit    [stockvc6.exe]
        WinMainCRTStartup [memcpy.obj]
    Allocation location
        HeapAlloc      [KERNEL32.dll]
        CStockApp::CStockApp(void) [Stock.cpp:155]
```

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

The location in memory where the error occurs

Call stack showing the function calls leading to the error

Flag indicating the line where the error occurs

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

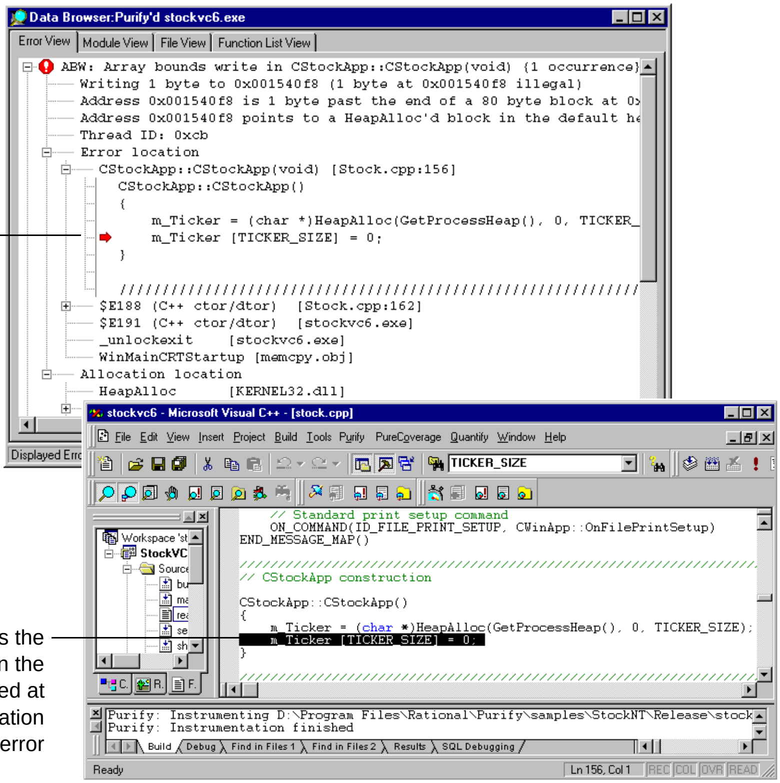
The level of detail provided in call stacks depends on the availability of debug and relocation data. Even if you build your program in release mode, you can still get the highest possible level of detail. For more information, look up *debug data, release builds* in the Purify online Help index.

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* in the Purify online Help index.

Correcting errors

Purify makes it easy to correct errors.



```
ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}
    Writing 1 byte to 0x001540f8 (1 byte at 0x001540f8 illegal)
    Address 0x001540f8 is 1 byte past the end of a 80 byte block at 0x
    Address 0x001540f8 points to a HeapAlloc'd block in the default he
    Thread ID: 0xcb
    Error location
        CStockApp::CStockApp(void) [Stock.cpp:156]
            CStockApp::CStockApp()
            {
                m_Ticker = (char *)HeapAlloc(GetProcessHeap(), 0, TICKER_
                m_Ticker [TICKER_SIZE] = 0;
            }

            ///////////////////////////////////////////////////////////////
        $E188 (C++ ctor/dtor)  [Stock.cpp:162]
        $E191 (C++ ctor/dtor)  [stockvc6.exe]
        _unlockexit     [stockvc6.exe]
        WinMainCRTStartup [memcpy.obj]
    Allocation location
        HeapAlloc       [KERNEL32.dll]
```

Double-click the line where the error occurs

```
    // Standard print setup command
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CStockApp construction

CStockApp::CStockApp()
{
    m_Ticker = (char *)HeapAlloc(GetProcessHeap(), 0, TICKER_SIZE);
    m_Ticker [TICKER_SIZE] = 0;
}

/////////////////////////////////////////////////////////////////////////////
```

```
Purify: Instrumenting D:\Program Files\Rational\Purify\samples\StockNT\Release\stock
Purify: Instrumentation finished
```

Purify opens the source code in the editor, positioned at the exact location of the error

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

Checking code coverage

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.

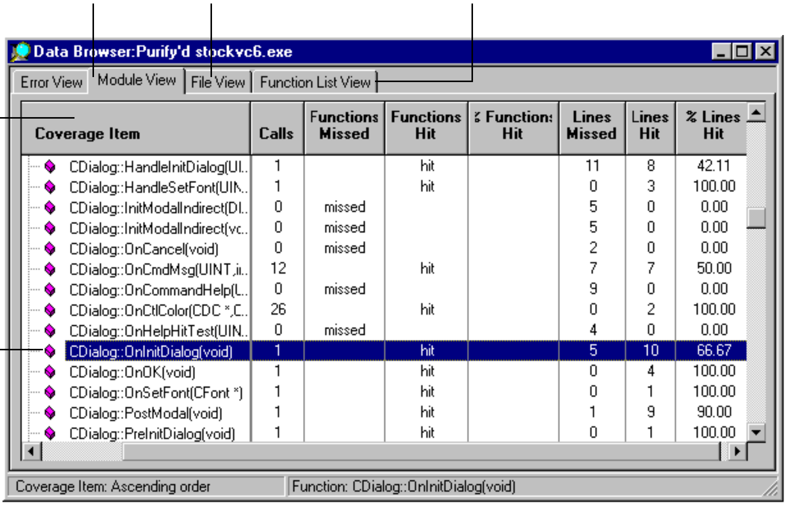
The Module View tab groups functions based on module

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



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)

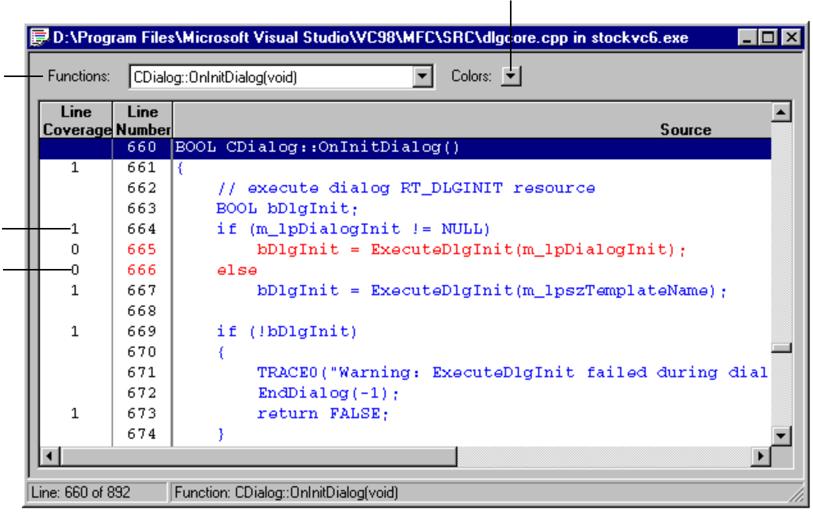
Purify can also display line-by-line coverage information marked directly on a copy of your code in an Annotated Source window. The color of each line of code indicates whether it is tested, untested, or partially tested, so that you can tell at a glance where you need to tighten up your testing.

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

Click to display information about color coding

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:
Colors:

Functions: CDialog::OnInitDialog(void)

```
Line      Line
Coverage  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* in the Purify online Help index.

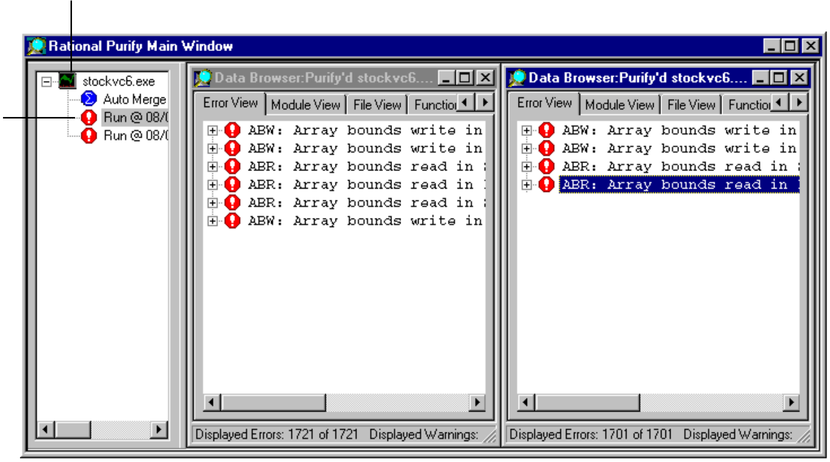
Comparing program runs

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

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

The Navigator window groups runs by program

A color-coded icon indicates the maximum message severity displayed in the error view for the run



```
ABW: Array bounds write in
ABW: Array bounds write in
ABR: Array bounds read in :
ABR: Array bounds read in :
ABR: Array bounds read in :
ABW: Array bounds write in
```

Displayed Errors: 1721 of 1721   Displayed Warnings:

```
ABW: Array bounds write in
ABW: Array bounds write in
ABR: Array bounds read in :
ABR: Array bounds read in
```

Displayed Errors: 1701 of 1701   Displayed Warnings:

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.

Customizing instrumentation

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

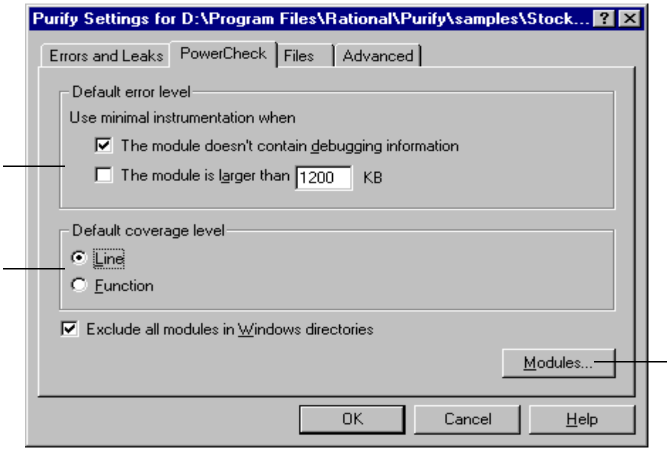
- *Precise* instrumentation, which provides full run-time error detection to pinpoint problems in any part of your program
- *Minimal* instrumentation, which improves Purify's performance while providing a basic level of error detection

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

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

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

and for coverage monitoring



Purify Settings for D:\Program Files\Rational\Purify\samples\Stock...

Default error level

Use minimal instrumentation when

☑ The module doesn't contain debugging information

☐ The module is larger than 1200 KB

Default coverage level

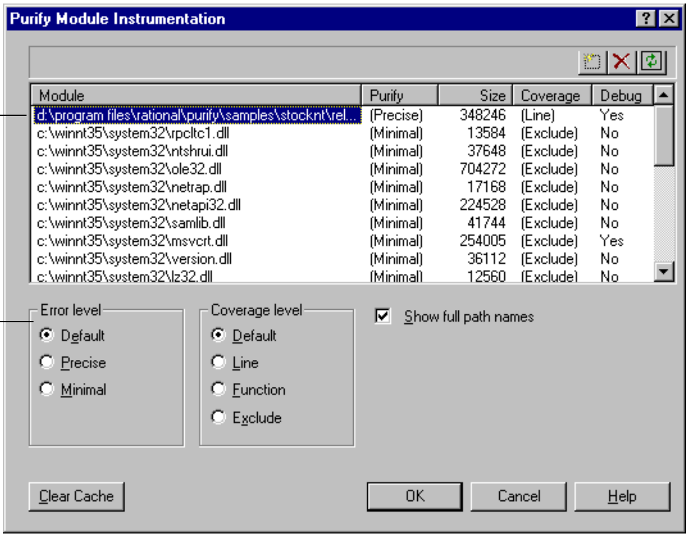
● Line

○ Function

☑ Exclude all modules in Windows directories

Click to override the defaults for individual modules

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



Select one or more modules in the list

Then specify the instrumentation level for the selected modules

Purify Module Instrumentation

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

Error level
● Default
○ Precise
○ Minimal

Coverage level
● Default
○ Line
○ Function
○ Exclude

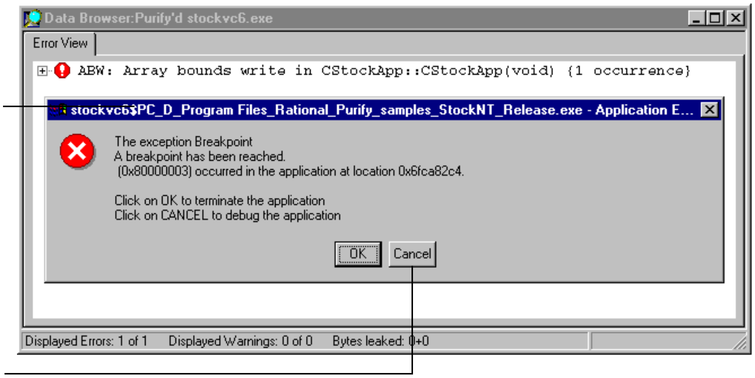
☑ Show full path names

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* and *powercheck* in the Purify online Help index.

Using just-in-time debugging

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


Data Browser:Purify'd stockvc6.exe
Error View
stockvc6$PC_D_Program Files_Rational_Purify_samples_StockNT_Release.exe - Application E...
OK
Cancel

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

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

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

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

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

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

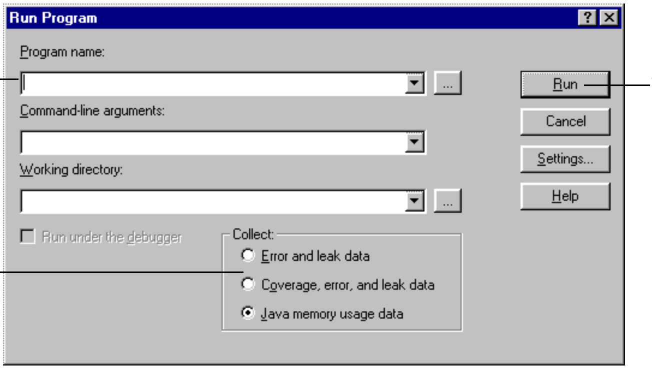
More information? Look up *break on error tool* in the Purify online Help index.

Using Purify standalone

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



First, specify the program you want to check

Second, specify whether to collect error and leak data, or coverage, error, and leak data

Third, click **Run**

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

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

Testing with Purify's command-line interface

Using Purify's command-line interface, you can use Purify with existing makefiles, batch files, and 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 and 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 executes. You can also call them from the QuickWatch dialog in the Visual Studio debugger, as well as by including them in your code.

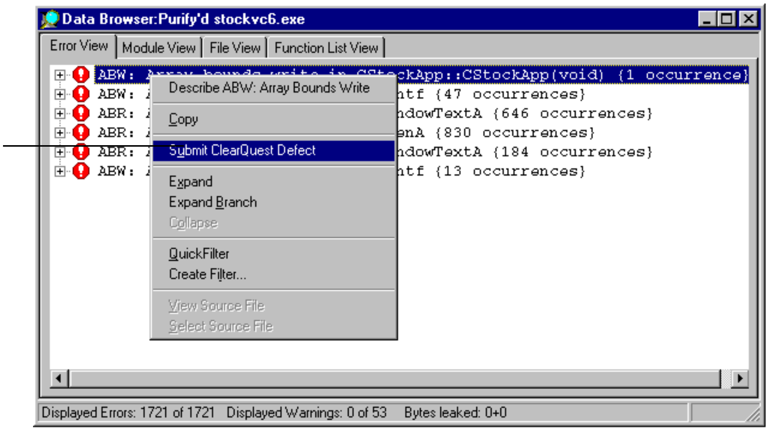
More information? For the complete listing of Purify API functions, including functions related to coverage monitoring, look up *api function list*. For instructions on using the functions, look up *api functions, using* in the Purify online Help index.

Using Purify in an integrated environment

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**

Data Browser:Purify'd stockvc6.exe

```
ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}
ABW: A                                      ntf {47 occurrences}
ABR: A                                      ndowTextA {646 occurrences}
ABR: A                                      enA {830 occurrences}
ABR: A                                      ndowTextA {184 occurrences}
ABW: A                                      ntf {13 occurrences}
```

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

Purify automatically supplies entries for a number of fields in the submit 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.

Getting started with Purify: Java code

The basic steps

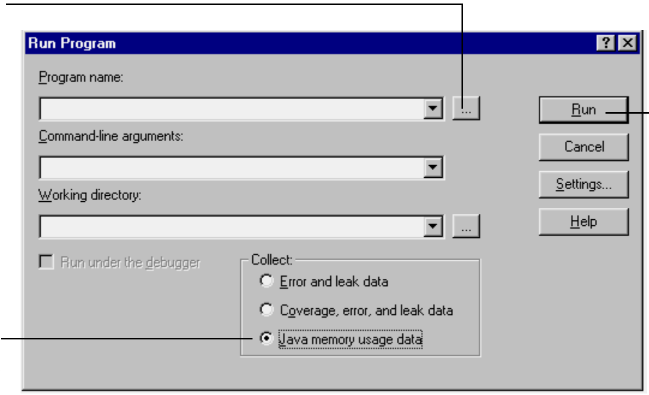
Java applications can consume a lot of memory over time if a forgotten reference to an object unintentionally prevents it from being garbage collected. With Rational® Purify®, you can determine how much memory your Java program is using, and detect exactly which objects are responsible for these “memory 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.



First, use the Browse button to select the Java program, applet, class, or JAR file that you want to profile

Second, select the button for collecting Java memory usage data

Third, click **Run**

Note: When you select a Java program (or applet, class, or JAR file) using the Browse button, Purify enters the name of the Purify Java helper program, `pstart.exe`, in the **Program name** field. The name of the Java program itself, along with the name of your specified Java virtual machine's Java viewer and any necessary options, is automatically entered in the **Command-line arguments** field. You must do the same if you enter information into these fields manually.

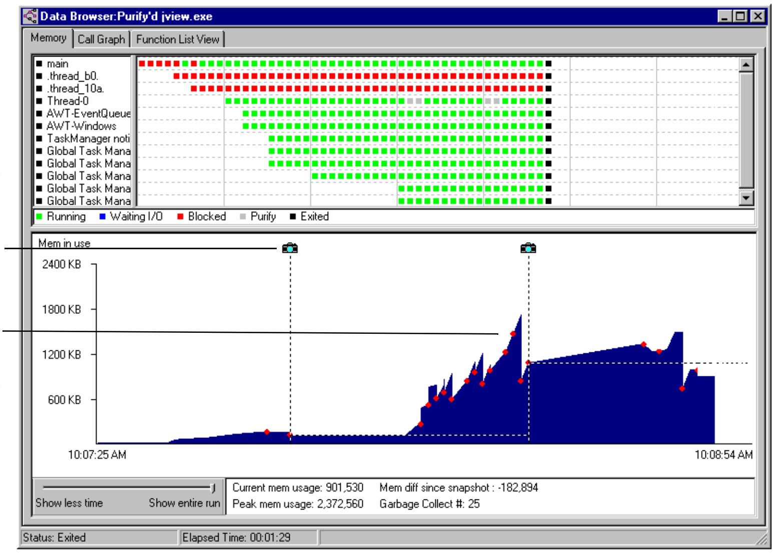
More information? Look up *specifying a JVM* and *running Java programs* in the Purify online Help index.

As your program runs, Purify intercepts and tabulates messages related to memory usage from the Java virtual machine. 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 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.


Data Browser:Purify'd jview.exe

Memory | Call Graph | Function List View

Status: Exited | Elapsed Time: 00:01:29

Current mem usage: 901,530
Peak mem usage: 2,372,560
Mem diff since snapshot : -182,894
Garbage Collect #: 25

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

Watch for increasing memory usage, then take a second snapshot

Watch the graph for fluctuations in memory usage. A large increase in memory usage may indicate a problem, especially when you can't reduce it by clicking ♻ to force a garbage collection.

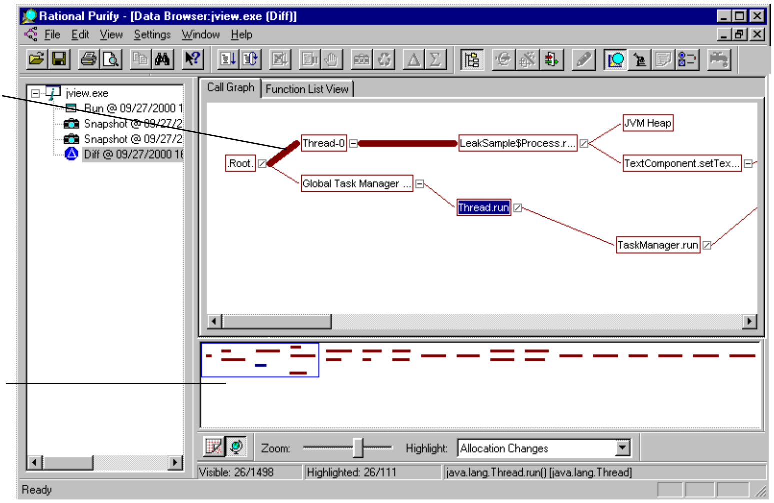
Now take another snapshot so that you have a "before" and "after" record of what's going on, and exit your program.

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

Comparing snapshots to identify problem methods

Select your second snapshot in the Navigator and click Δ 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.



Highlight: Allocation Changes

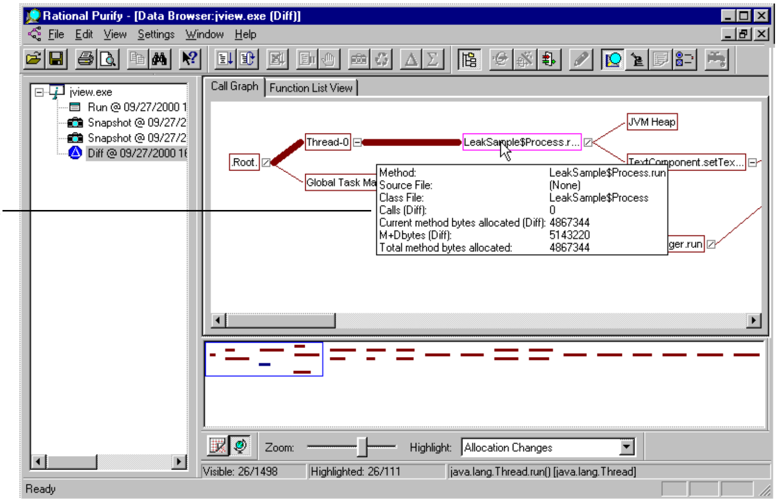
Visible: 26/1498 Highlighted: 26/111 java.lang.Thread.run() [java.lang.Thread]

The thickest lines indicate the paths where the most memory is allocated

The call graph overview helps you orient yourself within the call graph

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

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



Method: LeakSample$Process.run
Source File: (None)
Class File: LeakSample$Process
Calls (Diff): 0
Current method bytes allocated (Diff): 4867344
M+Dbytes (Diff): 5143220
Total method bytes allocated: 4867344

Visible: 26/1498
Highlighted: 26/111
java.lang.Thread.run() [java.lang.Thread]

Memory usage data is available directly from the call graph

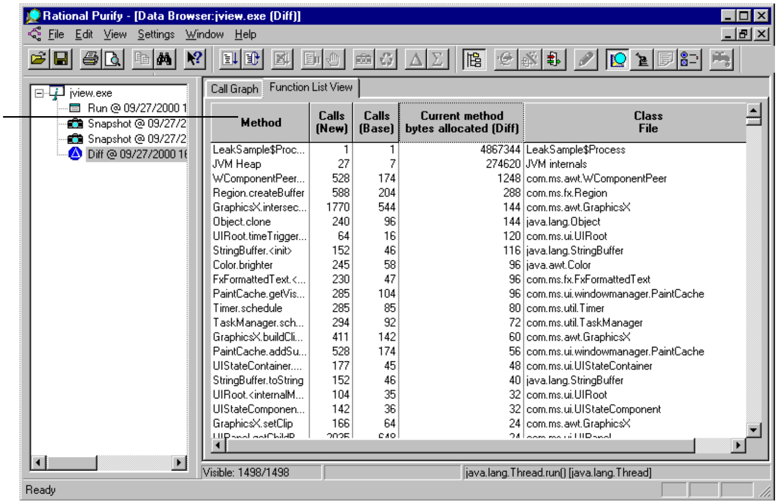
This allows you to zero in on the method that is consuming memory, as well as its descendants.

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 (Java), call graph (Java),* and *source code* 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 File
LeakSample$Proc...	1	1	4867344	LeakSample$Process
JVM Heap	27	7	274620	JVM internals
WComponentPeer...	528	174	1248	com.ms.awt.WComponentPeer
Region.createBuffer	588	204	288	com.ms.fx.Region
GraphicsX.intersec...	1770	544	144	com.ms.awt.GraphicsX
Object.clone	240	96	144	java.lang.Object
UIRoot.timeTrigger...	64	16	120	com.ms.ui.UIRoot
StringBuffer.<init>	152	46	116	java.lang.StringBuffer
Color.brighter	245	58	96	java.awt.Color
FxFormattedText.<...	230	47	96	com.ms.fx.FxFormattedText
PaintCache.getVis...	285	104	96	com.ms.ui.windowmanager.PaintCache
Timer.schedule	285	85	80	com.ms.util.Timer
TaskManager.sch...	294	92	72	com.ms.util.TaskManager
GraphicsX.buildCli...	411	142	60	com.ms.awt.GraphicsX
PaintCache.addSu...	528	174	56	com.ms.ui.windowmanager.PaintCache
UIStateContainer....	177	45	48	com.ms.ui.UIStateContainer
StringBuffer.toString	152	46	40	java.lang.StringBuffer
UIRoot.<internalM...	104	35	32	com.ms.ui.UIRoot
UIStateComponen...	142	36	32	com.ms.ui.UIStateComponent
GraphicsX.setClip	166	64	24	com.ms.awt.GraphicsX

Visible: 1498/1498

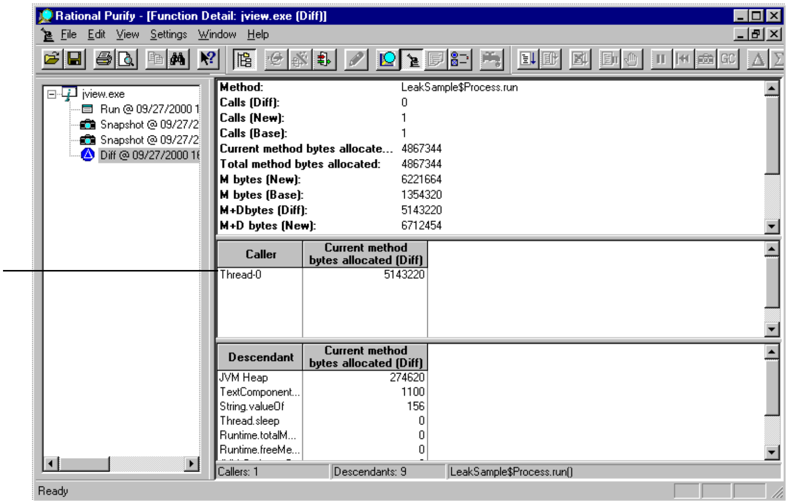
java.lang.Thread.run() [java.lang.Thread]

Ready

More information? Look up *function list view (Java)* 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.



Rational Purify - [Function Detail: jview.exe (Diff)]
File Edit View Settings Window Help
jview.exe
Run @ 09/27/2000 1
Snapshot @ 09/27/2
Snapshot @ 09/27/2
Diff @ 09/27/2000 1
Ready

Method:	LeakSample$Process.run
Calls (Diff):	0
Calls (New):	1
Calls (Base):	1
Current method bytes allocate...	4867344
Total method bytes allocated:	4867344
M bytes (New):	6221664
M bytes (Base):	1354320
M+Dbytes (Diff):	5143220
M+D bytes (New):	6712454

Caller	Current method bytes allocated (Diff)
Thread-0	5143220

Descendant	Current method bytes allocated (Diff)
JVM Heap	274620
TextComponent...	1100
String.valueOf	156
Thread.sleep	0
Runtime.totalM...	0
Runtime.freeMe...	0

Callers: 1 Descendants: 9 LeakSample$Process.run()

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

More information? Look up *function detail (Java)* 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.

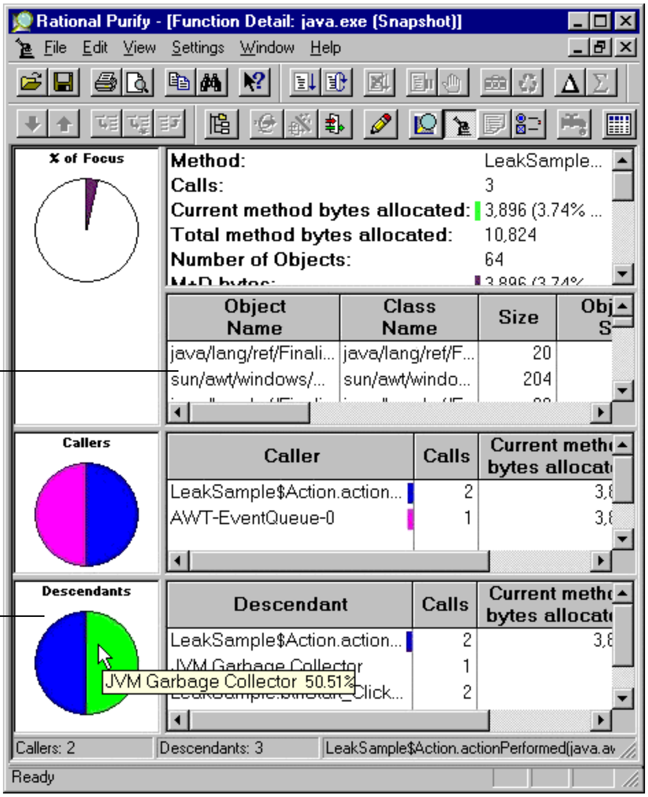
Looking for unneeded objects

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 Δ, 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.


Rational Purify - [Function Detail: java.exe (Snapshot)]
File Edit View Settings Window Help
% of Focus
Callers
Descendants
JVM Garbage Collector 50.51%

Method: LeakSample...
Calls: 3
Current method bytes allocated: 3,896 (3.74% ...
Total method bytes allocated: 10,824
Number of Objects: 64
M+D bytes: 3,896 (3.74%

Object Name	Class Name	Size	Obj S
java/lang/ref/Finali...	java/lang/ref/F...	20	
sun/awt/windows/...	sun/awt/windo...	204	

Caller	Calls	Current meth bytes allocat
LeakSample$Action.action...	2	3,8
AWT-EventQueue-0	1	3,8

Descendant	Calls	Current meth bytes allocat
LeakSample$Action.action...	2	3,8
JVM Garbage Collector	1	
LeakSample.btnStart_Click...	2	

Callers: 2 | Descendants: 3 | LeakSample$Action.actionPerformed(java.av

Ready

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

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

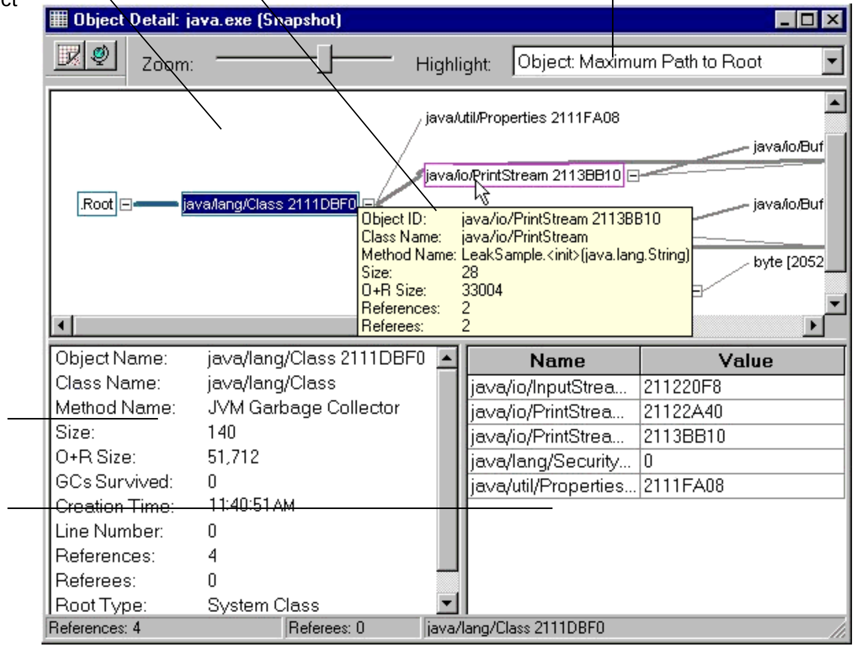
Examining object details

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



Object Detail: java.exe (Snapshot)
Zoom:
Highlight:
Root
java/lang/Class 2111DBF0
java/util/Properties 2111FA08
java/io/PrintStream 2113BB10
java/io/Buf
java/io/Buf
byte [2052

Highlight: Object: Maximum Path to Root

Object ID: java/io/PrintStream 2113BB10
Class Name: java/io/PrintStream
Method Name: LeakSample.<init>(java.lang.String)
Size: 28
O+R Size: 33004
References: 2
Referees: 2

Object Name: java/lang/Class 2111DBF0
Class Name: java/lang/Class
Method Name: JVM Garbage Collector
Size: 140
O+R Size: 51,712
GCs Survived: 0
Creation Time: 11:40:51 AM
Line Number: 0
References: 4
Referees: 0
Root Type: System Class

Name	Value
java/io/InputStrea...	211220F8
java/io/PrintStrea...	21122A40
java/io/PrintStrea...	2113BB10
java/lang/Security...	0
java/util/Properties...	2111FA08

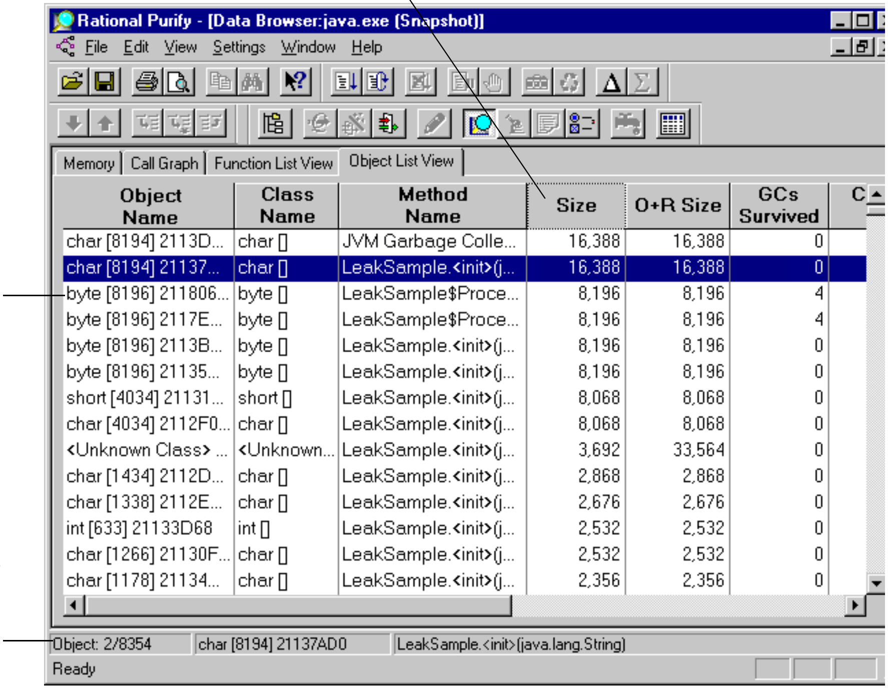
References: 4 Referees: 0 java/lang/Class 2111DBF0

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

The “object dump” shows the contents (data, references to other objects) of the object currently selected in the graph

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.


Rational Purify - [Data Browser:java.exe (Snapshot)]
File Edit View Settings Window Help
Memory
Call Graph
Function List View
Object List View

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

Object Name	Class Name	Method Name	Size	O+R Size	GCs Survived	C
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 (Java)*, *object detail (Java)* and *object list view (Java)* 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.

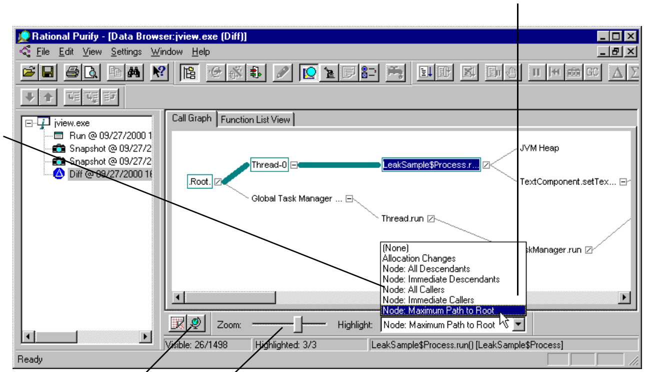
More information? Look up *saving data (Java)* in the Purify online Help index.

Advanced features for Java users

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



Highlight: Node: Maximum Path to Root

Visible: 26/1498 | Highlighted: 3/3 | LeakSample$Process.run() [LeakSample$Process]

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

26 of the 1498 functions in the current dataset are displayed in the call graph

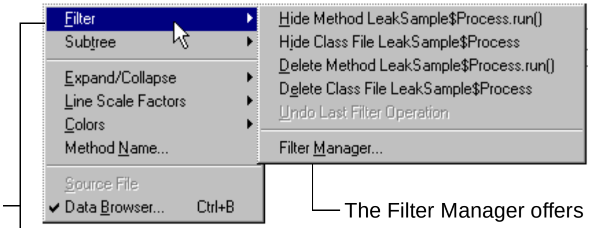
All 3 of the 3 functions on the maximum path to .Root are displayed in the call graph

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

Focusing your data

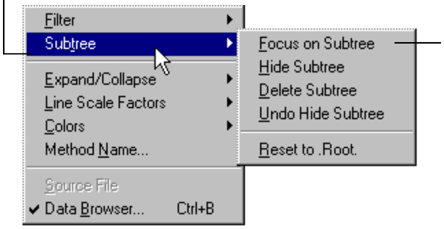
Use Purify's *filter* commands to remove a selected method, or all methods in a class file, from the set of data that Purify has collected. Alternatively, use *subtree* commands to focus on or remove a specific method and all its descendants from the dataset. Right-click a method in the call graph, function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

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


Filter
Subtree
Expand/Collapse
Line Scale Factors
Colors
Method Name...
Source File
✓ Data Browser... Ctrl+B
Hide Method LeakSample$Process.run()
Hide Class File LeakSample$Process
Delete Method LeakSample$Process.run()
Delete Class File LeakSample$Process
Undo Last Filter Operation
Filter Manager...

The Filter Manager offers additional filtering options

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

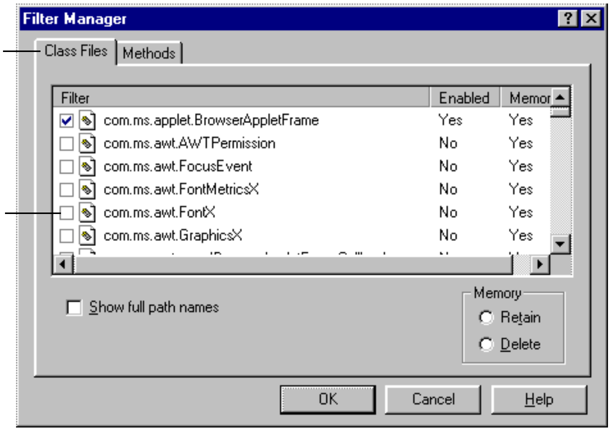


Select **Focus on Subtree** to delete all methods except those in the subtree

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

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

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



	Filter	Enabled	Memor
☑	com.ms.applet.BrowserAppletFrame	Yes	Yes
□	com.ms.awt.AWTPermission	No	Yes
□	com.ms.awt.FocusEvent	No	Yes
□	com.ms.awt.FontMetricsX	No	Yes
□	com.ms.awt.FontX	No	Yes
□	com.ms.awt.GraphicsX	No	Yes

□ Show full path names

Memory
○ Retain
○ Delete

You can filter data based on class file or on method

Click to enable or disable filters

More information? Look up *filtering data (Java)* and *subtrees (Java)* 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.

Index