IBM Rational PurifyPlus for Linux and UNIX
IBM Rational PurifyPlus for AIX
IBM Rational Purify for Linux and UNIX

Getting Started

VERSION: 7.0.1

MATERIAL ID: GI11-9427-00

Contents

Preface

What's in this guide?

This guide is designed to help you get up and running quickly with the components of IBM® Rational® PurifyPlus™. It includes information about:

- Using Purify to automatically pinpoint bugs and memory leaks everywhere in your C and C++ application code.
- Using PureCoverage to prevent untested C, C++, and Java applications from reaching end users.
- Using Quantify to improve the performance of your C and C++ applications by finding and eliminating bottlenecks.

Note: PurifyPlus for C/C++ applications is supported on commonly used UNIX platforms; see the release notes for specific information. For Java, you can use PureCoverage with applications running on the Solaris SPARC 32-bit Java virtual machine.

PurifyPlus—the essential tool for delivering reliable, high-performance applications—inserts monitoring instructions into the program's object code (C/C++) or byte code (PureCoverage for Java). This enables you to check your entire program, including third-party code and shared libraries, even when you don't have the source code.

Starting to use PurifyPlus is as easy as adding one of the component names (`purify`, `purecov`, or `quantify`) to the front of your link command line. For example, for a C program:

```
% purify cc -g hello_world.c
```

Audience

Read this guide for an introduction to the use of Purify, PureCoverage, or Quantify.

Other resources

- A complete online help system is available for each application. Select **Help > Help topics** in the user interface.

  For help with a window, select **Help > On window.** For help with a specific menu item or control button in a window, select **Help > On context**, then click the menu item or control button.

  Note: You can also view the help systems independently of the user interface if a web browser is on your PATH. Use the following commands:

  - `purify -onlinehelp`
  - `purecov -onlinehelp`
  - `quantify -onlinehelp`

- Rational developerWorks provides guidance and information that can help you implement and deepen your knowledge of Rational tools and best practices. It includes access to white papers, artifacts, source code, discussion forums, training, and documentation

  To access Rational developerWorks, go to www.ibm.com/developerWorks/rational/.

- For information about IBM Rational Software products, go to www.ibm.com/software/rational.

Contacting IBM Software Support

If you have questions about installing, using, or maintaining this product, contact IBM Software Support as follows:

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

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

For information regarding electronic problem submission and tracking, visit www.ibm.com/software/esr/

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

- Your name, company name, ICN number (IBM Customer Number), telephone number, and e-mail address
- Operating system, version number, and any service packs or patches you have applied
- Compiler version number
- Product name and release number
- Type of bug (examples: installation, build-time warning, build-time crash, run-time warning, run-time crash, unexpected results, user-interface problem)
- Instructions for reproducing the problem
- Workarounds used
- Your PMR number (if you are following up on a previously reported problem)

Using Purify

1

Purify: What it does

Purify® is the most comprehensive dynamic analysis tool available for automatically finding software bugs. It checks all the code in your program, including any application, system, and third-party libraries. Purify works with complex software applications, including multi-threaded and multi-process applications.

Purify checks every memory access operation, pinpointing *where* errors occur and providing detailed diagnostic information to help you analyze *why* the errors occur. Among the many errors that Purify helps you locate and understand are:

- Reading or writing beyond the bounds of an array
- Using uninitialized memory
- Reading or writing freed memory
- Reading or writing beyond the stack pointer
- Reading or writing through null pointers
- Leaking memory and file descriptors

With Purify, you can develop clean code from the start, rather than spending valuable time debugging problem code later.

This chapter introduces the basic concepts involved in using Purify. For complete information, see the Purify online help system.

Finding errors in Hello World

This chapter shows you how to use Purify to find memory errors in an example Hello World program. If you run the example yourself, you should expect minor platform-related differences in program output from what is shown here.

Before you begin:

1 Create a new working directory. Go to the new directory and copy all files that begin with `hello` from the `<purifyhome>/example` directory. For example:

```
% mkdir /usr/home/chris/pwork
% cd /usr/home/chris/pwork
% cp <purifyhome>/example/hello* .
```

2 Examine the code in `hello_world.c`. The version of `hello_world.c` provided with Purify is slightly different from the traditional version.

```
/*
 * (C) Copyright IBM Corporation. 1992, 2009. All Rights Reserved.
  ...
 * This is a test program used in Purifying Hello World
 */

#include <stdio.h>
#include <malloc.h>

static char *helloWorld = "Hello, World";

main()
{
   char *mystr = malloc(strlen(helloWorld));

   strncpy(mystr, helloWorld, 12);
   printf("%s\n", mystr);
}
```

At first glance there are no obvious errors, yet the program actually contains a memory access error and leaked memory that Purify will help you to identify.

Instrumenting a program

1 Compile and link the Hello World program, then run the program to verify that it produces the expected output:

```
% cc -g hello_world.c
% a.out
```

output ——— `Hello, World`

2 Instrument the program by adding `purify` to the front of the compile/link command line. To get the maximum amount of detail in Purify messages, use the `-g` option:

```
% purify cc -g hello_world.c
```

Compiling and linking in separate stages

If you compile and link your program in separate stages, specify `purify` only on the link line. For example:

On the compile line, use:

```
% cc -c -g hello_world.c
```

On the link line, use:

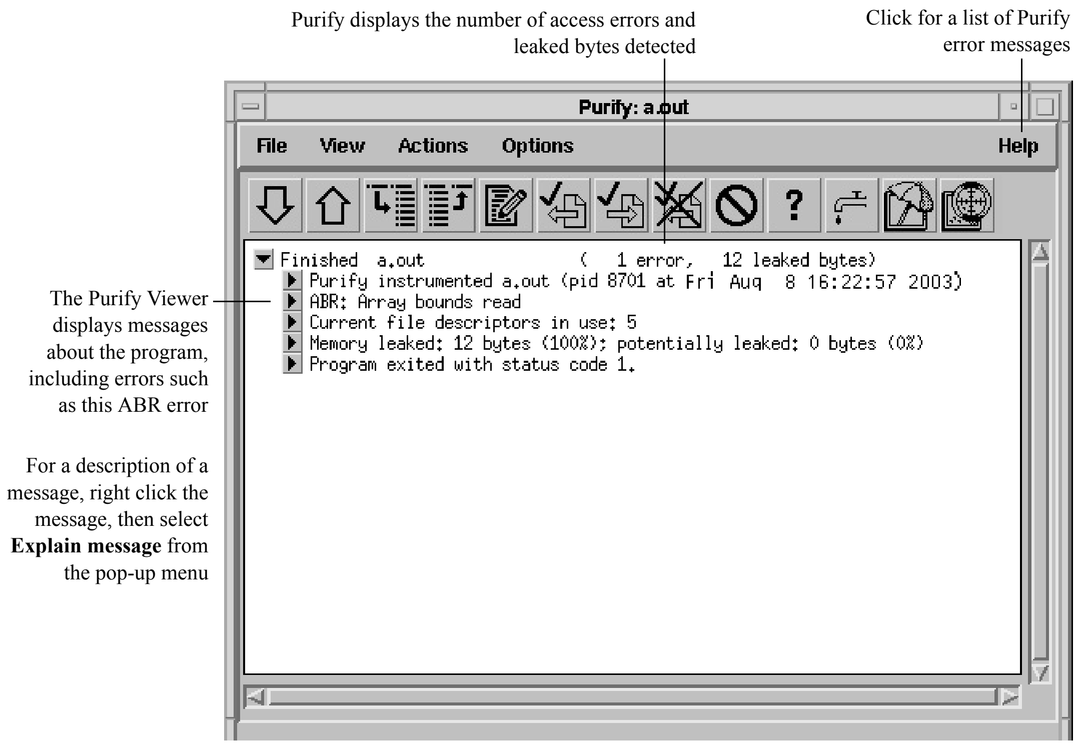
```
% purify cc -g hello_world.o
```

Running the instrumented program

Run the instrumented Hello World program:

```
% a.out
```

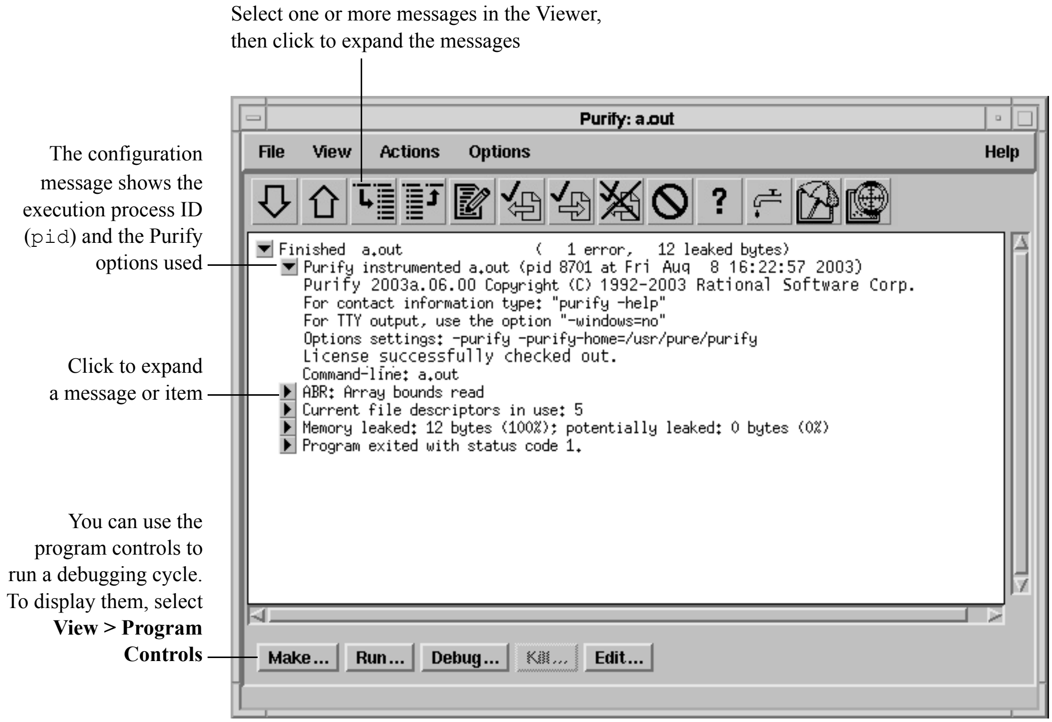
This prints "Hello, World" in the current window and displays the Purify Viewer.



Notice that the instrumented Hello World program starts, runs, and exits normally. Purify does not stop the program when it finds an error.

Seeing all your errors at a glance

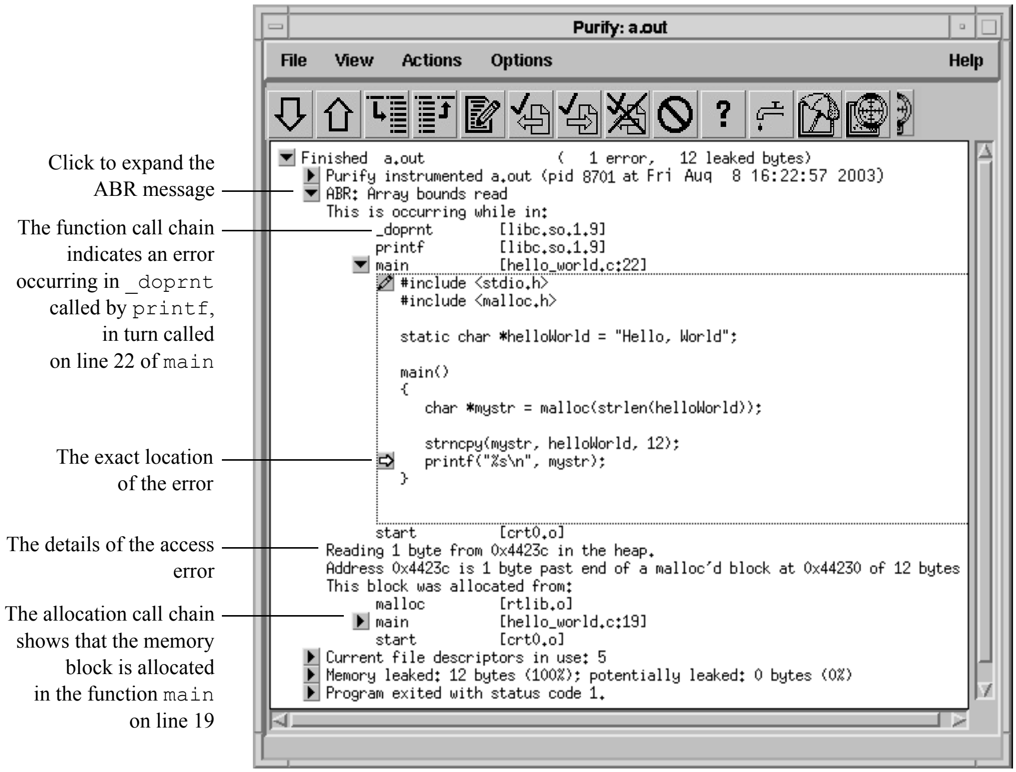
The Purify Viewer displays the results of the run of the instrumented Hello World program. You can expand each message to see additional details.



Note: The Viewer displays messages for a single executable only. It is specific to the name of the executable, the directory containing the executable, and the user ID.

Finding and correcting errors

Purify reports an array bounds read (ABR) memory access error in the Hello World program. You can expand the ABR message to see the exact location of the error.



Note: To make debugging easier, Purify reports line numbers, source filenames, and local variable names whenever possible if you use the `-g` compiler option when you build the program. If you do not use the `-g` option, Purify reports only function names and object filenames.

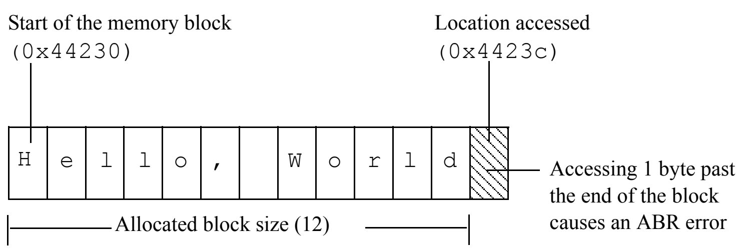
Understanding the cause of the error

To understand the cause of the ABR error, look at the code in `hello_world.c` again.

```
.
.
.
15  static char *helloWorld = "Hello, World";
16
17  main()
18  {
19     char *mystr = malloc(strlen(helloWorld));
20
21     strncpy(mystr, helloWorld, 12);
22     printf("%s\n", mystr);
23  }
```

Purify reports that the ABR error occurs here (line 22)

On line 22, the program requests `printf` to display `mystr`, which is initialized by `strncpy` on line 21 for the 12 characters in “Hello, World.” However, `_doprnt` is accessing one byte more than it should. It is looking for a `NULL` byte to terminate the string. The extra byte for the string’s `NULL` terminating character has *not* been allocated and initialized.



For more information, see *How Purify finds memory-access errors* on page 32.

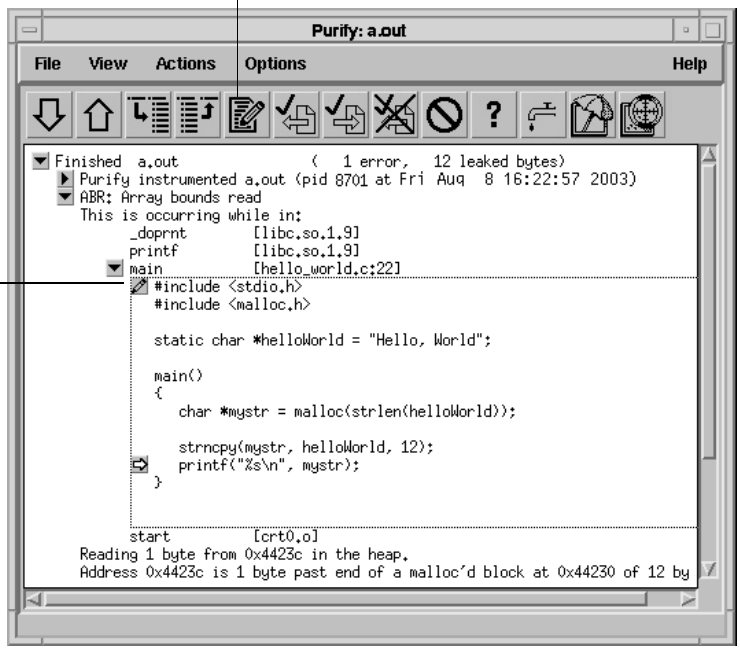
Correcting the ABR error

To correct this ABR error:

1 Click the Edit tool  to open an editor.

Click to edit the source code

Or click here to edit the source code



Note: By default, Purify displays seven lines of the source code file in the Viewer. You can change the number of lines of source code displayed by setting an X resource.

2 Change lines 19 and 21 as follows:

```
19  char *mystr = malloc(strlen(helloWorld)+1);
20
21  strncpy(mystr, helloWorld, 13);
```

Finding leaked memory

When a program exits, Purify searches for memory leaks and reports all memory blocks that were allocated but for which no pointers exist.

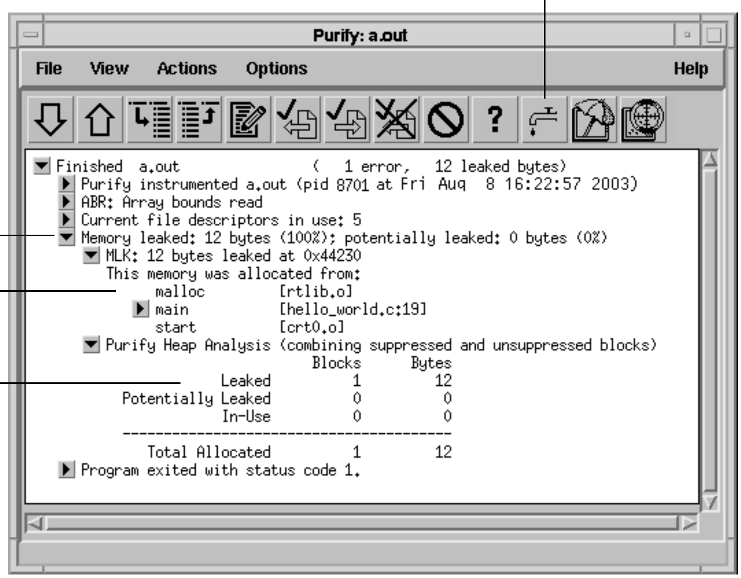
Note: When you run longer-running instrumented programs, you can click the New Leaks tool to generate a new leaks summary while the program is running.

1 Expand the memory-leaked summary for Hello World.

The memory-leaked summary shows the number of leaked bytes as a percentage of the total heap size. If there is more than one memory leak, Purify sorts them by the number of leaked bytes, displaying the largest leaks first.

2 Expand the MLK message.

When you run your programs, click the New Leaks tool to generate a new leaks summary while the program is running



The memory-leaked summary reports 12 bytes of leaked memory

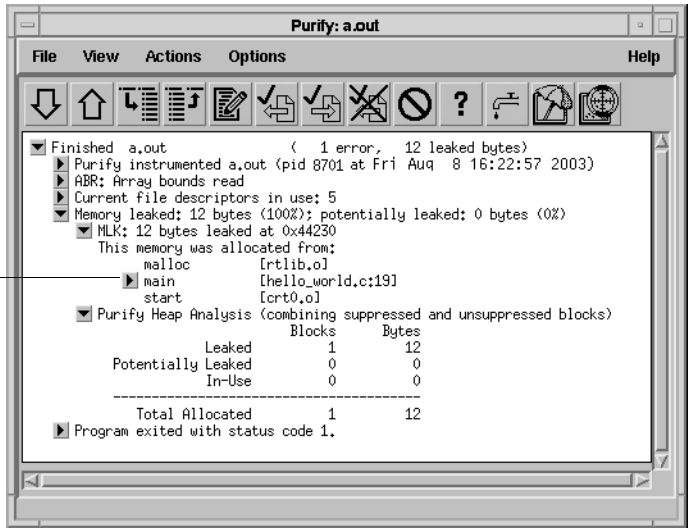
The call chain shows how the leaked memory was allocated

Memory analysis by category

Correcting the MLK error

It is not immediately obvious why this memory was leaked. If you look closer, however, you can see that this program does not have an `exit` statement at the end. Because of this omission, the `main` function returns rather than calls `exit`, thereby making `mystr`— the only reference to the allocated memory—go out of scope.

Line 19 of `hello_world.c` in `main` allocates 12 bytes of leaked memory. The start of this memory block is `0x44230`, the same block with the array bounds read error in `_doprnt`



If `main` called `exit` at the end, `mystr` would remain in scope at program termination, retaining a valid pointer to the start of the allocated memory block. Purify would then have reported it as memory in use rather than memory leaked. Alternatively, `main` could `free` `mystr` before returning, deallocating the memory so it is no longer in use or leaked.

To correct this MLK error:

1 Click the Edit tool  to open an editor.

2 Add a call to `exit(0)` at the end of the Hello World program.

Looking at the heap analysis

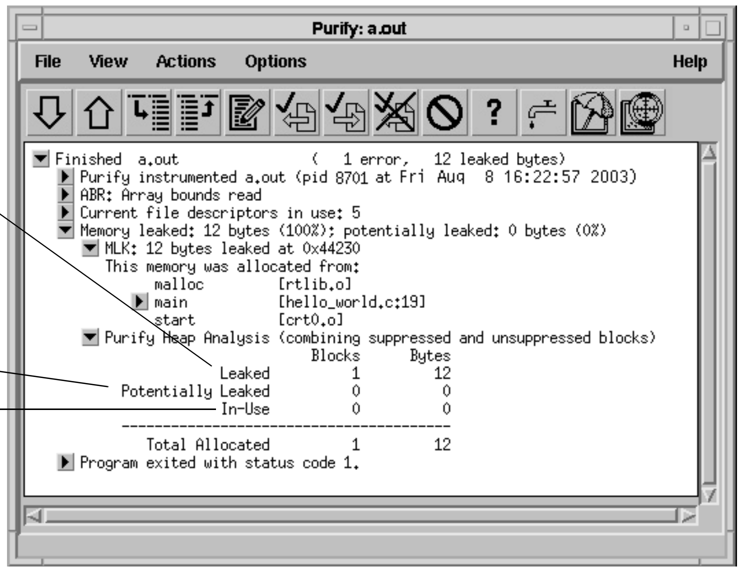
Purify distinguishes between three memory states, reporting both the number of blocks in each state and the sum of their sizes:

- Leaked memory
- Potentially leaked memory
- Memory in use

A true memory leak (MLK) is memory to which your program has no pointer

A potential memory leak (PLK) is memory that does not have a pointer to its beginning, but does have one to its interior

Memory in use (MIU) is memory to which your program has pointers (these are not leaks)



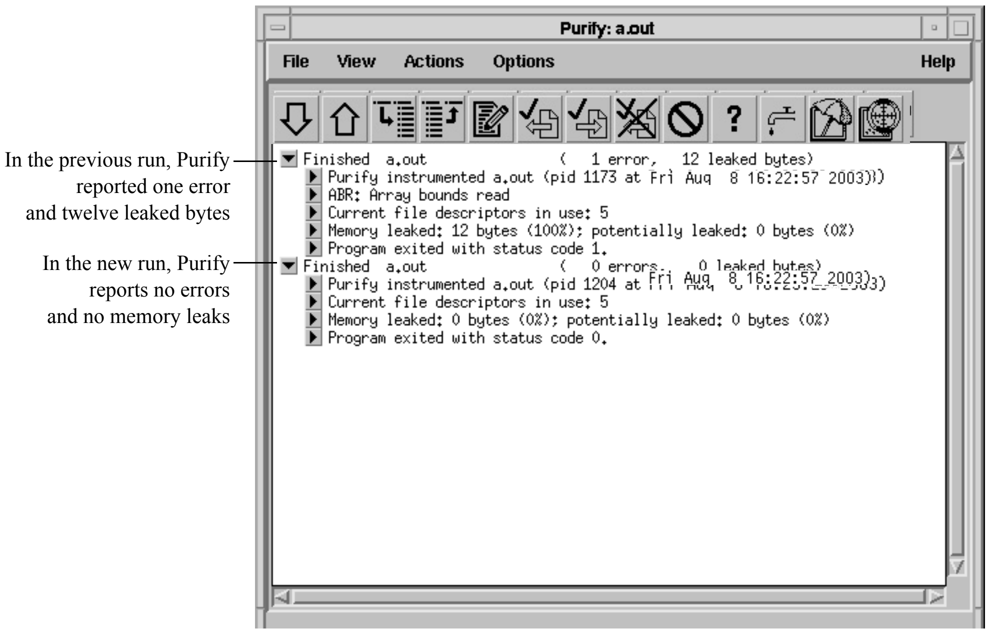
The exit status message provides information about:

- *Basic memory usage* containing statistics not easily available from a single shell command. It includes program code and data size, as well as maximum heap and stack memory usage in bytes.
- *Shared-library memory usage* indicating which libraries were dynamically linked and their sizes.

Comparing program runs

To verify that you have corrected the ABR and MLK errors, recompile the program with `purify`, and run it again.

Purify displays the results of the new run in the same Viewer as the previous run so it's easy to compare them. In this simple Hello World program, you can quickly see that the new run no longer contains the ABR and MLK errors.



Congratulations! You have successfully Purify'd the Hello World program.

Suppressing Purify messages

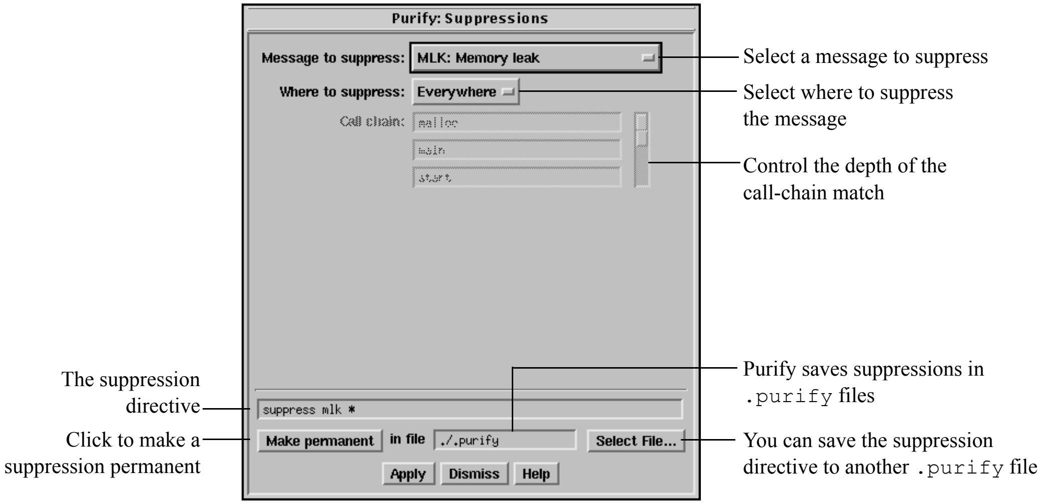
A large program can generate hundreds of error messages. To quickly focus on the most critical ones, you can suppress the less critical messages based on their type and source. For example, you might want to hide all informational messages, or hide all messages that originate in a specific file.

You can suppress messages in the Viewer either during or after a run of your program. To suppress a message in the Viewer:

1 Select the message you want to suppress.

2 Select **Options > Suppressions**.

Purify displays the Suppressions dialog, containing information about the selected message.



You can also specify suppressions directly in a `.purify` file. Suppressions created in the Viewer take precedence over suppressions in `.purify` files; however, they apply only to the current Purify session. Unless you click **Make permanent**, they do not remain when you restart the Viewer.

Saving Purify output to a view file

A view file is a binary representation of all messages generated in a Purify run that you can browse with the Viewer or use to generate reports independent of a Purify run. You can save a run to a view file to compare the results of one run with the results of subsequent runs, or to share the file with other developers.

Saving a run to a view file from the Viewer

To save a program run to a view file from the Viewer:

1. Wait until the program finishes running, then click the run to select it.
2. Select **File > Save As**.
3. Type a filename, using the `.pv` extension to identify the run as a Purify view file.

Opening a view file

To open a view file from the Viewer:

1 Select **File > Open**.

2 Select the view file you want to open.

Purify displays the run from the view file in the Viewer. You can work with the run just as you would if you had run the program from the Viewer.

You can also use the `-view` option to open a view file. For example:

```
% purify -view <filename>.pv
```

This opens the `<filename>.pv` view file in a new Viewer.

Using your debugger with Purify

You can run an instrumented program directly under your debugger so that when Purify finds an error, you can investigate it immediately.

Alternatively, you can enable Purify's just-in-time (JIT) debugging feature to have Purify start your debugger *only* when it encounters an error—and you can specify which types of errors trigger the debugger. JIT debugging is useful for errors that appear only once in a while. When you enable JIT debugging, Purify suspends execution of your program just before the error occurs, making it easier to analyze the error.

Using Purify with PureCoverage

Purify is designed to work closely with PureCoverage, the component of PurifyPlus used for code coverage analysis. PureCoverage identifies the parts of your program that have not yet been tested so you can tell whether you're exercising your program sufficiently for Purify to find all the memory errors in your code.

To use Purify with PureCoverage, add both product names to the front of your link line. For example:

```
% purify <purifyoptions> purecov <purecovoptions> \
   cc -g hello_world.c -o hello_world
```

To start PureCoverage from the Purify Viewer, click the PureCoverage icon  in the toolbar.

For more information, see *Purify: What it does* on page 13.

Purify API functions

You can call Purify's API functions from your source code or from your debugger to gain more control over Purify's error checking. By calling these functions from your debugger, you get additional control without modifying your source code. You can use Purify's API functions to check memory state and to search for memory and file-descriptor leaks.

For example, by default Purify reports memory leaks only when you exit your program. However, if you call the API function `purify_new_leaks` at key points throughout your program, Purify reports the memory leaks that have occurred since the last time the function was called. This periodic checking enables you to locate and track memory leaks more effectively.

To use Purify API functions, include `<purifyhome>/purify.h` in your code and link with `<purifyhome>/purify_stubs.a`.

Commonly used API functions	Description
`int purify_describe (char *addr)`	Prints specific details about memory
`int purify_is_running (void)`	Returns `"TRUE"` if the program is instrumented
`int purify_new_inuse (void)`	Prints a message on all memory newly in use
`int purify_new_leaks (void)`	Prints a message on all new leaks
`int purify_new_fds_inuse (void)`	Lists the new open file descriptors
`int purify_printf (char *format, ...)`	Prints formatted text to the Viewer or log-file
`int purify_watch (char *addr)`	Watches for memory `write`, `malloc`, `free`
`int purify_watch_n (char *addr, int size, char *type)`	Watches memory: `type` = `"r"`, `"w"`, `"rw"`
`int purify_watch_info (void)`	Lists active watchpoints
`int purify_watch_remove (int watchno)`	Removes a specified watchpoint
`int purify_what_colors (char *addr, int size)`	Prints the color coding of memory

Build-time options

Specify build-time options on the link line when you instrument a program with Purify. For example:

```
% purify -cache-dir=$HOME/cache -always-use-cache-dir cc ...
```

Commonly used build-time options	Default
`-always-use-cache-dir` Forces all instrumented object files to be written to the global cache directory	`no`
`-cache-dir` Specifies the global directory where Purify caches instrumented object files	`<purifyhome>/cache`
`-ignore-runtime-environment` Prevents the runtime Purify environment from overriding the option values used in building the program	`no`
`-linker` Sets the alternative linker to build the executables instead of the system default	system-dependent
`-print-home-dir` Prints the name of the directory where Purify is installed, then exits	

Conversion characters for filenames

Use these conversion characters when specifying filenames for options such as `-log-file` and `-view-file`.

Character	Converts to
`%V`	Full pathname of program with "`/`" replaced by "`_`"
`%v`	Program name
`%p`	Process id (pid)
qualified filenames (`./%v.pv`)	Absolute or relative to current working directory
unqualified filenames (no '/')	Directory containing the program

Runtime options

Specify runtime options on the link line or by using the `PURIFYOPTIONS` environment variable. For example:

```
% setenv PURIFYOPTIONS "-log-file=mylog.%v.%p ‘printenv PURIFYOPTIONS‘"
```

Commonly used runtime options	Default
`-auto-mount-prefix` Removes the prefix used by file system auto-mounters	`/tmp_mnt`
`-chain-length` Sets the maximum number of stack frames to print in a report	`6`
`-fds-in-use-at-exit` Specifies that the file descriptor in use message be displayed at program exit	`yes`
`-follow-child-processes` Controls whether Purify monitors child processes in an instrumented program	`no`
-jit-debug Enables just-in-time debugging	none
`-leaks-at-exit` Reports all leaked memory at program exit	`yes`
`-log-file` [a] Writes Purify output to a log file instead of the *Viewer* window	`stderr`
`-messages` Controls display of repeated messages: `"first"`, `"all"`, or in a `"batch"` at program exit	`first`
-program-name Specifies the full pathname of the instrumented program if `argv[0]` contains an undesirable or incorrect value	`argv[0]`
`-show-directory` Shows the directory path for each file in the call chain, if the information is available	`no`
`-show-pc` Shows the full pc value in each frame of the call chain	`no`
`-show-pc-offset` Appends a pc-offset to each function name in the call chain	`no`

Commonly used runtime options	Default
`-view-file` [1] Saves Purify output to a view file (`.pv`) instead of the *Viewer*.	none
-user-path Specifies a list of directories in which to search for programs and source code	none
`-windows` Redirects Purify output to `stderr` instead of the *Viewer* if `-windows=no`	none

a. *Can use the conversion characters listed on page 28*

Purify messages

Purify reports the following messages. For detailed, platform-specific information, see the Purify online help system.

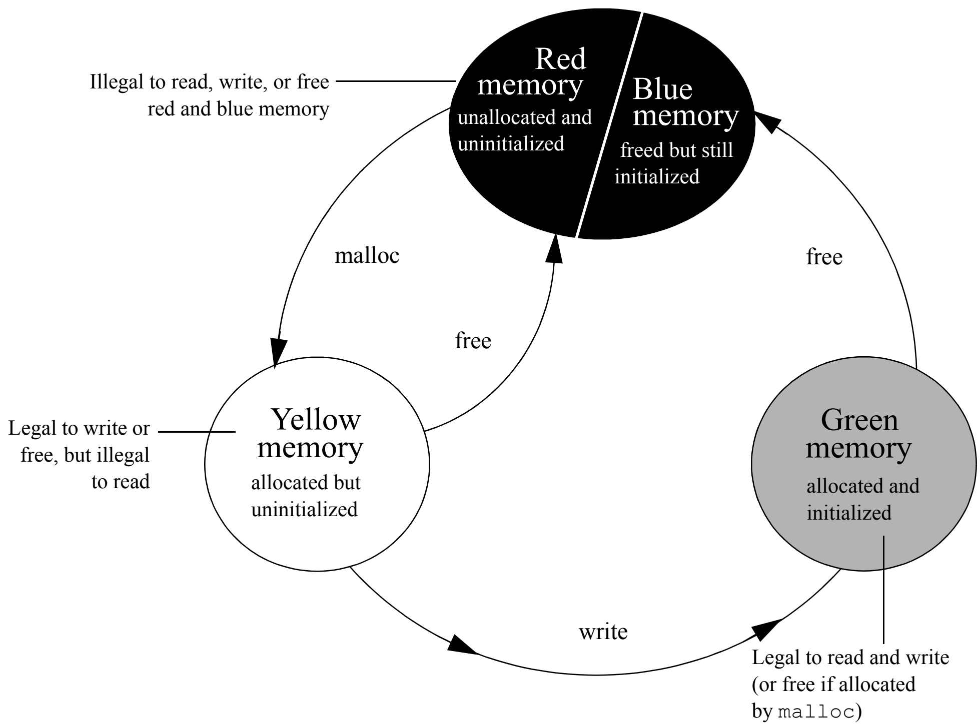
Message	Description	Severity*	Message	Description	Severity*
ABR	Array Bounds Read	W	**NPR**	Null Pointer Read	F
ABW	Array Bounds Write	C	**NPW**	Null Pointer Write	F
BRK	Misuse of Brk or Sbrk	C	**PAR**	Bad Parameter	W
BSR	Beyond Stack Read	W	**PLK**	Potential Leak	W
BSW	Beyond Stack Write	W	**PMR**	Partial UMR	W
COR	Core Dump Imminent	F	**SBR**	Stack Array Bounds Read	W
FIU	File Descriptors In Use	I	**SBW**	Stack Array Bounds Write	C
FMM	Freeing Mismatched Memory	C	**SIG**	Signal	I
FMR	Free Memory Read	W	**SOF**	Stack Overflow	W
FMW	Free Memory Write	C	**UMC**	Uninitialized Memory Copy	W
FNH	Freeing Non Heap Memory	C	**UMR**	Uninitialized Memory Read	W
FUM	Freeing Unallocated Memory	C	**WPF**	Watchpoint Free	I
IPR	Invalid Pointer Read	F	**WPM**	Watchpoint Malloc	I
IPW	Invalid Pointer Write	F	**WPN**	Watchpoint Entry	I
MAF	Malloc Failure	I	**WPR**	Watchpoint Read	I
MIU	Memory In-Use	I	**WPW**	Watchpoint Write	I
MLK	Memory Leak	W	**WPX**	Watchpoint Exit	I
MRE	Malloc Reentrancy Error	C	**ZPR**	Zero Page Read	F
MSE	Memory Segment Error	W	**ZPW**	Zero Page Write	F

** Message severity: F=Fatal, C=Corrupting, W=Warning, I=Informational*

How Purify finds memory-access errors

Purify monitors every memory operation in your program, determining whether it is legal. It keeps track of memory that is not allocated to your program, memory that is allocated but uninitialized, memory that is both allocated and initialized, and memory that has been freed after use but is still initialized.

Purify maintains a table to track the status of each byte of memory used by your program. The table contains two bits that represent each byte of memory. The first bit records whether the corresponding byte has been allocated. The second bit records whether the memory has been initialized. Purify uses these two bits to describe four states of memory: red, yellow, green, and blue.



Purify checks each memory operation against the color state of the memory block to determine whether the operation is valid. If the program accesses memory illegally, Purify reports an error.

- *Red:* Purify labels heap memory and stack memory red initially. This memory is unallocated and uninitialized. Either it has never been allocated, or it has been allocated and subsequently freed.

  In addition, Purify inserts guard zones around each allocated block and each statically allocated data item, in order to detect array bounds errors. Purify colors these guard zones red and refers to them as *red zones*. It is illegal to read, write, or free red memory because it is not owned by the program.

- *Yellow:* Memory returned by `malloc` or `new` is yellow. This memory has been allocated, so the program owns it, but it is uninitialized. You can write yellow memory, or free it if it is allocated by `malloc`, but it is illegal to read it because it is uninitialized. Purify sets stack frames to yellow on function entry.

- *Green:* When you write to yellow memory, Purify labels it green. This means that the memory is allocated and initialized. It is legal to read or write green memory, or free it if it was allocated by `malloc` or `new`. Purify initializes the *data* and `bss` sections of memory to green.

- *Blue*: When you free memory after it is initialized and used, Purify labels it blue. This means that the memory is initialized, but is no longer valid for access. It is illegal to read, write, or free blue memory.

Since Purify keeps track of memory at the byte level, it catches all memory-access errors. For example, it reports an uninitialized memory read (UMR) if an `int` or `long` (4 bytes) is read from a location previously initialized by storing a `short` (2 bytes).

How Purify checks statically allocated memory

In addition to detecting access errors in dynamic memory, Purify detects references beyond the boundaries of data in global variables and static variables; that is, data allocated statically at link time as opposed to dynamically at run time.

Here is an example of data that is handled by the static checking feature:

```
int array[10];
main() {
   array[11] = 1;
}
```

In this example, Purify reports an array bounds write (ABW) error at the assignment to `array[11]` because it is 4 bytes beyond the end of the array.

Purify inserts red zones around each variable in your program's static-data area. If the program attempts to read from or write to one of these red zones, Purify reports an array bounds error (ABR or ABW).

Purify inserts red zones into the data section *only* if all data references are to known data variables. If Purify finds a data reference that is relative to the start of the data section as opposed to a known data variable, Purify is unable to determine which variable the reference involves. In this case, Purify inserts red zones at the beginning and end of the data section only, not between data variables.

Purify provides several command-line options and directives to aid in maximizing the benefits of static checking.

Using PureCoverage 2

PureCoverage: What it does

During the development process, software changes daily, sometimes hourly. Unfortunately, test suites do not always keep pace. PureCoverage® is a simple, easily deployed tool that identifies the lines and functions in your code that have not been exercised by testing.

PureCoverage supports C and C++ applications, as well as Java applications running on a Solaris SPARC 32-bit Java virtual machine (JVM).

Using PureCoverage, you can:

- Pinpoint untested areas of your code
- Accumulate coverage data over multiple runs and multiple builds
- Merge data from different programs sharing common source code
- Work closely with Purify to make sure that Purify finds errors throughout your *entire* application
- Automatically generate a wide variety of useful reports
- Access the coverage data so you can write your own reports
- Collect coverage data on UNIX for viewing on a Windows system

PureCoverage provides the information you need to identify gaps in testing quickly, saving time and effort.

This chapter introduces the basic concepts involved in using PureCoverage. For complete information, see the PureCoverage online help, including the *Java Supplement* for PureCoverage.

Finding untested Java code

PureCoverage provides accurate coverage information that identifies all the gaps in your testing of Java code.

Before you run your Java application under PureCoverage, note that the default setting for Java, unlike C and C++, is to collect data at the method level. Method-level data allows you to identify which methods are the least tested.

Unless you already know which classes you want to focus on, collect method-level data the first time you run your program. Then, when you know the classes you want to investigate in detail, collect line-level data for them. To collect line-level data for specific classes:

1 Specify the PureCoverage option `-purecov-granularity=line`. Note that debug data must be available for PureCoverage to collect data at this level.

2 Define directives in the `<purecovhome>\.purecov.java` file to limit profiling to the classes that you want to analyze. The file provides information about the directive syntax. You can find the `<purecovhome>` directory with the following command:

```
% purecov -printhomedir
```

To collect method-level code coverage data for Java code, run PureCoverage with the `-java` option, as follows:

- For an applet:

```
% purecov [<PureCoverage options>] -java \
<applet viewer> [<applet viewer options>] <html file>
```

- For a class file:

```
% purecov [<PureCoverage options>] -java \
<Java executable> <Java options>] <class>
```

- For a JAR file:

```
% purecov [<PureCoverage options>] -java \
<Java executable> [<Java options>] <JAR switch> \
<JAR file>.jar
```

- For a container program:

```
% purecov [<PureCoverage options>] -java <exename> \
[<arguments to exename>]
```

To display the coverage data for the program, use a command such as the following:

```
% purecov -view java.234.0.pcv
```

where 234 is the process id and 0 is a sequence number assigned when the data file is saved.

For an example showing how to use PureCoverage to monitor Java code, and for information about ways to control code monitoring, see the *Java Code Coverage Supplement* for PureCoverage, which is included with the PureCoverage online help.

Finding untested C/C++ code

This chapter shows you how to use PureCoverage to find the untested parts of the `hello_world.c` program.

Before you begin:

1 Create a new working directory. Go to the new directory, and copy the files that begin with `hello` from the `<purecovhome>/example` directory:

```
% mkdir /usr/home/pat/example
% cd /usr/home/pat/example
% cp <purecovhome>/example/hello* .
```

2 Examine the code in `hello_world.c`.

The version of `hello_world.c` provided with PureCoverage is slightly more complicated than the usual textbook version.

```
#include <stdio.h>
void display_hello_world();
void display_message();

main(argc, argv)
int argc;
char** argv;
{
if (argc == 1)
display_hello_world();
else
display_message(argv[1]);
exit(0);
}

void
display_hello_world()
{
   printf("Hello, World\n");
}

void
display_message(s)
   char *s;
{
   printf("%s, World\n", s);
}
```

Instrumenting a C/C++ program

1 Compile and link the Hello World program, then run the program to verify that it produces the expected output:

```
% cc -g hello_world.c
% a.out
Hello, World
```

output

2 Instrument the program by adding `purecov` to the front of the compile/link command line. To have PureCoverage report the maximum amount of detail, use the `-g` option:

```
% purecov cc -g hello_world.c
```

Note: If you compile your code *without* the `-g` option, PureCoverage provides only function-level data. It does not show line-level data.

A message appears, indicating the version of PureCoverage that is instrumenting the program:

```
PureCoverage 7.0.1 Solaris 2 (32-bit), (C) Copyright IBM
Corporation. 1994, 2009. All Rights Reserved.
Instrumenting: hello_world.o Linking
```

Note: When you compile and link in separate stages, add `purecov` only to the link line.

Running the instrumented C/C++ program

Run the instrumented Hello World program:

```
% a.out
```

PureCoverage displays the following:

Name of the instrumented executable

You can use this command to display technical support contact information

Start-up banner

```
**** PureCoverage instrumented a.out (pid 3466 at Wed May 10 10:32:40 2009)
* PureCoverage 7.0.1 (32-bit), (C) Copyright IBM Corporation. 1994, 2009.
  * All Rights Reserved.
  * For contact information type: "purecov -help"
  * Options settings: -purecov \
    -purecov-home=/usr/rational/releases/purecov.sol.7.0.1
  * License successfully checked out
  * Command-line: a.out
```

Normal program output

```
Hello, World
```

PureCoverage saves coverage data to a .pcv file

```
****  PureCoverage instrumented a.out (pid 3466)  ****
  * Saving coverage data to /usr/home/pat/example/a.out.pcv.
  * To view results type: purecov -view /usr/home/pat/example/a.out.pcv
```

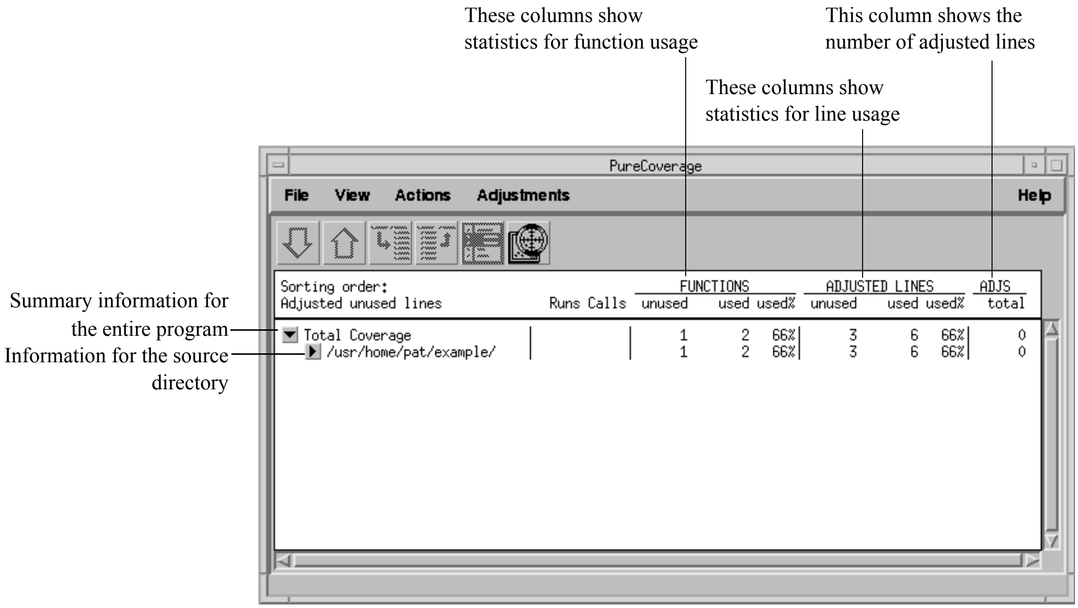
The `a.out` program produces its normal output, just as if it were not instrumented. When the program completes execution, PureCoverage writes coverage information for the session to the file `a.out.pcv`. Each time the program runs, PureCoverage updates this file with additional coverage data.

Displaying C/C++ coverage data

To display the coverage data for the program, use the command:

```
% purecov -view a.out.pcv
```

This displays the PureCoverage Viewer.



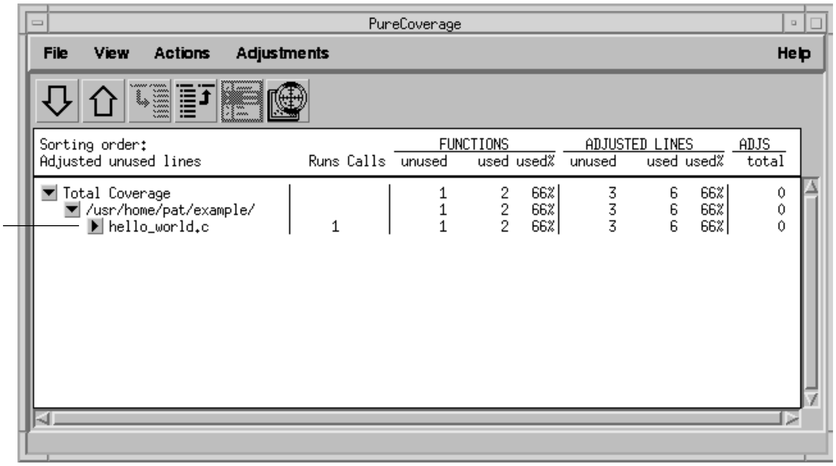
In this example, there is only one source directory, so the information displayed for the directory is identical to the `Total Coverage` information.

Note: The default header for line statistics is `ADJUSTED LINES`, not just `LINES`. This is because PureCoverage has an adjustment feature that lets you adjust coverage statistics by excluding specific lines. Under certain circumstances, the adjusted statistics give you a more practical reflection of coverage status than the actual coverage statistics. The `ADJS` column in this example contains zeroes, indicating that it does not include adjustments.

Expanding the file-level detail

Click ↵ next to `.../example/` to expand the file-level information for the directory.

File-level information includes the number of runs for which PureCoverage collected data



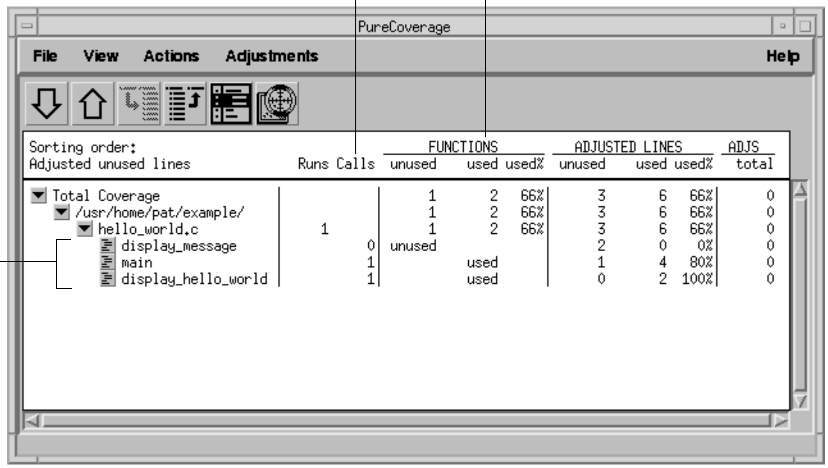
You used only one file in the `example` directory to build `a.out`. Therefore the `FUNCTIONS` and `ADJUSTED LINES` information for the file is the same as for the directory. The number 1 in the Runs column indicates that you ran the instrumented `a.out` only once.

Note: When you are examining data collected for multiple executables, or for executables that have been rebuilt with some changed files, the number of runs can be different for each file.

Examining function-level detail

Expand the `hello_world.c` line to show function-level information.

The Viewer shows coverage information for the functions `display_message`, `main`, and `display_hello_world`.

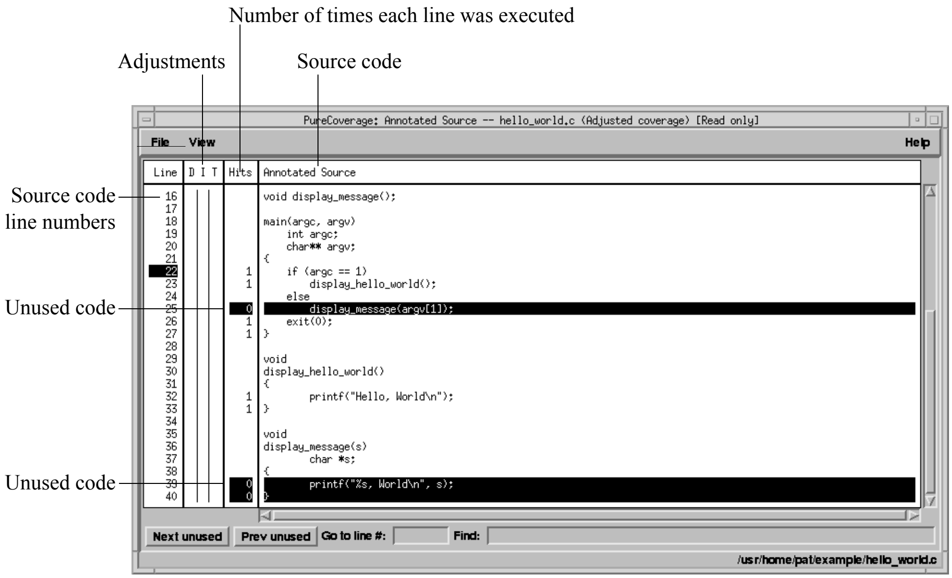


PureCoverage does not list the `printf` function or any functions that it calls. The `printf` function is a part of the system library, `libc`. By default, PureCoverage excludes collection of data from system libraries.

Examining the annotated source

To see the source code for `main` annotated with coverage information, click the Annotated Source tool next to `main` in the Viewer. PureCoverage displays the Annotated Source window.

Note: The Annotated Source window is available only for files that you compile using the `-g` debugging option. If you are working with Java code, you must, in addition, specify the option `-purecov-granularity=line` when you run the program.



PureCoverage highlights code that was not used when you ran the program. In this file only two pieces of code were not used:

- The `display_message(argv[1]);` statement in `main`
- The entire `display_message` function

A quick analysis of the code reveals the reason: the program was invoked without arguments.

Improving Hello World's test coverage

To improve the test coverage for Hello World:

1 Without exiting PureCoverage, run the program again, this time with an argument. For example:

```
% a.out Goodbye
```

PureCoverage displays the following:

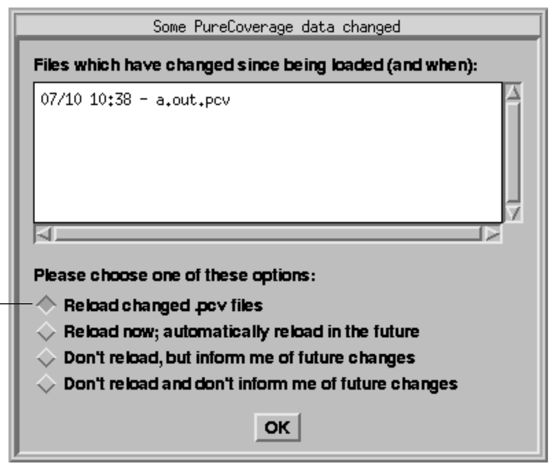
```
**** PureCoverage instrumented a.out (pid 3466 at Wed May 10
10:32:40 2009)
* PureCoverage 7.0.1 (32-bit), (C) Copyright IBM
Corporation. 1994, 2009.
```

```
  * All Rights Reserved.
* For contact information type: "purecov -help"
* Options settings: -purecov \
-purecov-home=/usr/rational/releases/purecov.sol.7.0.1
* License successfully checked out
* Command-line: a.out Goodbye
Goodbye, World

****  PureCoverage instrumented a.out (pid 17331)  ****
* Saving coverage data to
/usr/home/pat/example/a.out.pcv.
* To view results type: purecov -view
/usr/home/pat/example/a.out.pcv
```

2 PureCoverage displays a dialog confirming that coverage data has changed for this run. Select **Reload changed .pcv files** and click **OK**.

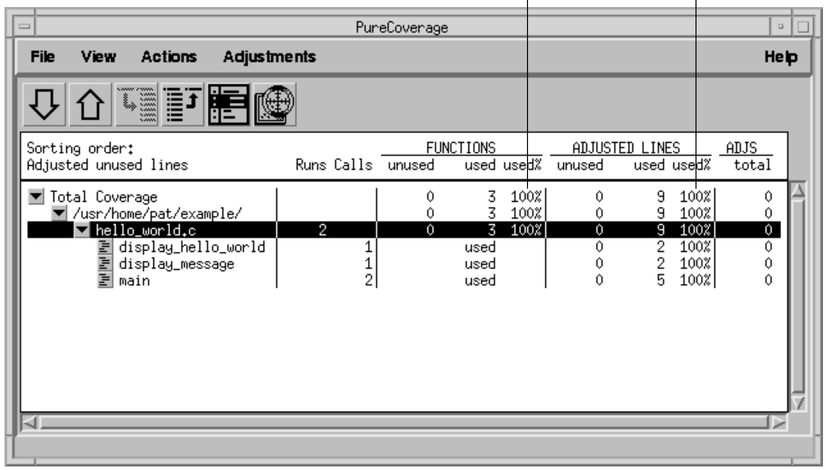
Reload the changed `a.out.pcv` file



Note: This dialog appears only if the PureCoverage Viewer is open when you run the program.

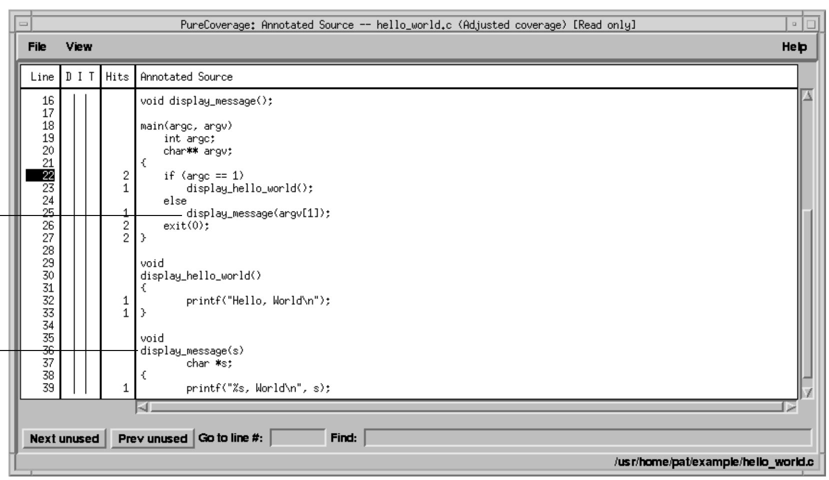
PureCoverage updates the coverage information in the Viewer and the Annotated Source window.

Function and line coverage is now 100%



The statement `display_message (argv[1]);...`

and the function `display_message` are now shown as used



Note: If you still have untested lines, it is possible that your compiler is generating unreachable code.

3 Select **File > Exit**.

Viewing UNIX coverage data on Windows

You can collect coverage data on your UNIX system and view it on Windows using Rational PureCoverage for Windows.

To collect coverage data for viewing on Windows, assign the value windows or both to the -view-file-format option. You can specify the option in the environment variable PURECOVOPTIONS or on the command line.

With the option set to windows, PureCoverage saves coverage data to a .cfy file, which you can analyze using PureCoverage for Windows. With the option set to both, PureCoverage saves data to a .pcv file as well.

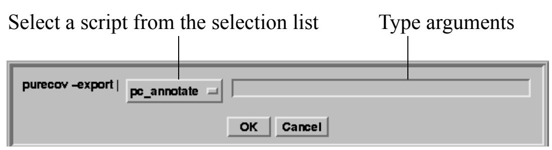
PureCoverage for UNIX does not merge .cfy files, unlike .pcv files. You can merge .cfy files when you view them on Windows.

For more information, see the online help for PureCoverage on both UNIX and Windows.

Using report scripts

You can use PureCoverage report scripts to format and process PureCoverage data. The report scripts are located in the <purecovhome>/scripts directory.

Select **File > Run script** to open the script dialog.



You can also run report scripts from the command line.

Report scripts

pc_annotate   Produces an annotated source text file

```
% pc_annotate [-force-merge][-apply-adjustments=no]\
[-file=<basename>...][-type=<type>][<prog>.pcv...]
```

pc_below   Reports low coverage

```
% pc_below [-force-merge][-apply-adjustments=no][-percent=<pct>]\
[<prog>.pcv...]
```

Report scripts

`pc_build_diff` Compares PureCoverage data from two builds of an application

```
% pc_build_diff [-apply-adjustments=no][-prefix=XXXX....] old.pcv \
new.pcv
```

`pc_covdiff` Annotates the output of diff for modified source code

Note: Cannot run from Viewer

```
% yourdiff <name> | pc_covdiff [-context=<lines>] \
[-format={diff|side-by-side|new-only}][-lines=<boolean>] \
[-tabs=<stops>][-width=<width>][-force-merge][-apply-adjustments=no] \
-file=<name> <prog>.pcv...
```

`pc_diff` Lists files for which coverage has changed

```
% pc_diff [-apply-adjustments=no] old.pcv new.pcv
```

`pc_email` Mails a report to the last person who modified insufficiently covered files

```
% pc_email [-force-merge][-apply-adjustments=no][-percent=<pct>] \
[<prog>.pcv...]
```

`pc_select` Identifies the subset of tests required to exercise modified source code

```
% <list of changed files> | pc_select \
[-diff=<rules>][-canonicalize=<rule>]test1.pcv test2.pcv...
```

`pc_ssheet` Produces a summary in spreadsheet format

```
% pc_ssheet [-force-merge][-apply-adjustments=no][<prog>.pcv...]
```

`pc_summary` Produces an overall summary in table format

```
% pc_summary [-file=<name>...] [-force-merge] [-apply-adjustments=no]
[<prog>.pcv...]
```

PureCoverage options

PureCoverage provides command-line options for controlling operations and handling coverage data both for C/C++ and Java code.

Build-time options

For a C or C++ application, specify build-time options on the link line when you instrument with PureCoverage. For example:

```
% purecov -always-use-cache-dir cc ...
```

For a Java application, specify these options (which for Java are not actually build-time options) on the command line when you run the application with PureCoverage.

For C, C++, and Java applications, you can also set these options using the `PURECOVOPTIONS` environment variable. For example:

```
% setenv PURECOVOPTIONS "-always-use-cache-dir"
```

Commonly used build-time options	Default
`-always-use-cache-dir` Forces all PureCoverage instrumented object files to be written to the global cache directory. Does not apply to Java.	`no`
`-auto-mount-prefix` Removes the prefix used by file system auto-mounters.	`/tmp_mnt`
`-cache-dir` Specifies the global directory for caching instrumented object files. Does not apply to Java.	`<purecovhome>/cache`
`-ignore-runtime-environment` Prevents the runtime PureCoverage environment from overriding the option values used in building the program. Does not apply to Java.	`no`
`-linker` Specifies a linker other than the system default for building executables. Does not apply to Java.	system-dependent

Runtime options

For a C or C++ application, specify runtime options on the link line when you instrument with PureCoverage. For a Java application, specify these options on the command line when you run the application with PureCoverage.

For C, C++, and Java applications, you can also set these options using the `PURECOVOPTIONS` environment variable. For example:

```
% setenv PURECOVOPTIONS \
"-counts-file=./test1.pcv `printenv PURECOVOPTIONS`"
```

Commonly used runtime options	Default
`-counts-file` Specifies an alternate file for writing coverage count data in binary format. Can use the conversion characters listed on page 28.	`%v.pcv` for C/C++; `%v%p%n.pcv` for Java, where `%n` is a sequence number.
`-follow-child-processes` Controls whether PureCoverage is enabled in forked child processes	`no`
`-log-file` Specifies a log file for PureCoverage runtime messages. Can use the conversion characters listed on page 28.	`stderr`
`-program-name` Specifies the full pathname of the PureCoverage instrumented program. Does not apply to Java.	`argv[0]`
`-user-path` Specifies a list of directories to search for source code. Can use the conversion characters listed on page 28	none

Analysis-time options

Use analysis-time options with analysis-time mode options. For example:

```
% purecov -merge=result.pcv -force-merge filea.pcv fileb.pcv
```

Commonly used analysis-time options	Default
`-apply-adjustments` Applies all adjustments in the $HOME/.purecov.adjust file to exported coverage data	`yes`
`-force-merge` Forces the merging of coverage data files (.pcv) obtained from different versions of the same object file	`no`

Analysis-time mode options

Command-line syntax:

```
% purecov -<mode option> [analysis-time options] \
<file1.pcv file2.pcv ...>
```

Analysis-time mode options	Compatible options
`-export` Merges and writes coverage counts from multiple coverage data files (.pcv) in export format to a specified file (-export=<filename>) or to stdout	`-apply-adjustments`
`-extract` Extracts adjustment data from source code files and writes it to $HOME/.purecov.adjust	none
`-merge=<filename.pcv>` Merges and writes coverage counts from multiple coverage data files (.pcv) in binary format	`-force-merge`
`-view` Opens the PureCoverage Viewer for analysis of one or more coverage data files (.pcv)	`-force-merge,` `-user-path`

Using Quantify

3

Quantify: What it does

Your application's runtime performance—its speed—is one of its most visible and critical characteristics. Developing high-performance software that meets the expectations of customers is not an easy task. Complex interactions between your code, third-party libraries, the operating system, hardware, networks, and other processes make it difficult to identify the causes of performance degradation. Use Quantify® to improve your code's runtime performance.

Profiling runtime performance

Quantify is a powerful analytic tool that identifies the portions of you application that dominate its execution time. It supports C and C++ applications. Quantify gives you the insight to eliminate performance problems so that your software runs faster. With Quantify, you can:

- Get accurate and reliable performance data
- Control how data is collected, collecting data for a small portion of your application's execution or the entire run
- Compare *before* and *after* runs to see the impact of your changes on performance
- Easily locate and fix only the problems with the highest potential for improving performance

This chapter introduces the basic concepts involved in using Quantify to collect and analyze runtime performance data. For complete information, see the Quantify online help system.

How Quantify profiles application performance

Unlike sampling-based profilers, Quantify reports performance data for your program without any profiler overhead. The numbers you see represent the time your program would take without Quantify. Quantify instruments and reports performance data for *all* the code in your program, including system and third-party libraries, shared libraries, and statically linked modules.

Quantify counts machine cycles: Quantify uses Object Code Insertion (OCI) technology to count the instructions your program executes and to compute how many cycles they require to execute. Counting cycles means that the time Quantify records in your code is independent of accidental local conditions and, assuming that the input does not change, identical from run to run. The fact that performance data is **repeatable** enables you to see precisely the effects of algorithm and data-structure changes.

Since Quantify counts cycles, it gives you accurate data at any scale. You do *not* need to create long runs or make numerous short runs to get meaningful data as you must with sampling-based profilers—one short run and you have the data. As soon as you can run a test program, you can collect meaningful performance data and establish a baseline for future comparison.

Quantify times system calls: Quantify measures the elapsed (wall clock) time of each system call made by your program and reports how long your program waited for those calls to complete. You can immediately see the effects of improved file access or reduced network delay on your program. You can optionally choose to measure system calls by the amount of time the kernel records for the process, which is the same as the time the UNIX `/bin/time` utility records.

Quantify distributes time accurately: Quantify distributes each function's time to its callers so you can tell at a glance which function calls were responsible for the majority of your program's time. Unlike `gprof`, Quantify does not make assumptions about the average cost per function. Quantify measures it directly.

Collecting performance data

To collect performance data for a C/C++ program:

1 Add `quantify` to the front of the *link* command line. For example:

```
% quantify cc -g hello_world.c -o hello_world
```

2 Run the instrumented program as you usually do:

```
% hello_world
```

When the program starts, Quantify prints license and support information, followed by the expected output from your program.

```
**** Quantify instrumented hello_world (pid 20352 at Sat 5
Jul 08:41:27 2009)
Quantify 7.0.1 Solaris 2, (C) Copyright IBM Corporation.
1993, 2009. All Rights Reserved.
   * For contact information type: "quantify -help"
   * Quantify licensed to Quantify Evaluation User
   * Quantify instruction counting enabled.
Hello, World.

Quantify: Sending data for 37 of 1324 functions
 from hello_world (pid 20352).........done.
```

Program output — Hello, World.

Data transmission — Quantify: Sending data for 37 of 1324 functions from hello_world (pid 20352).........done.

When the program finishes execution, Quantify transmits the performance data it collected to qv, Quantify's data-analysis program.

Interpreting the program summary

After each dataset is transmitted, Quantify prints a program summary showing at a glance how the original, non-instrumented, program is expected to perform.

Time Quantify expects the original program to take

Time spent executing program functions (compute-bound)

Time spent waiting for system calls to complete

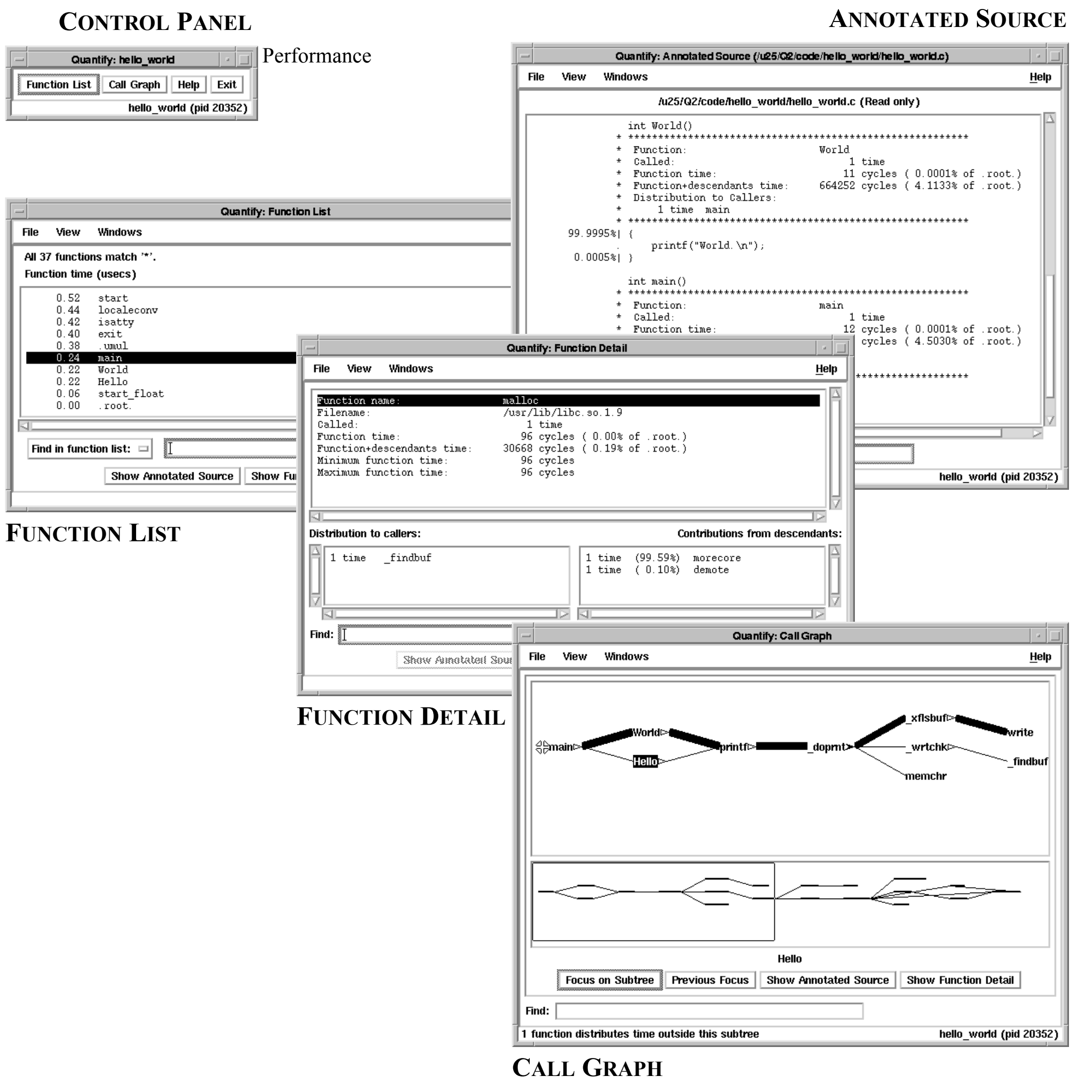
Time spent loading dynamic libraries

Time taken to collect data includes Quantify's counting overhead and any memory effects

```
Quantify: Resource Statistics for hello_world (pid 20352)
 *                                       cycles       secs
 * Total counted time:                  16148821     0.323 (100.0%)
 *     Time in your code:                   2721     0.000 (  0.0%)
 *     Time in system calls:              843950     0.017 (  5.2%)
 *     Dynamic library loading:         15302150     0.306 ( 94.8%)
 *
 *
 * Note: Data collected assuming a sparcstation_lx with clock rate of 750 MHz.
 * Note: These times exclude Quantify overhead and possible memory effects.
 *
 * Elapsed data collection time:       0.336 secs
 *
 * Note: This measurement includes Quantify overhead.
```

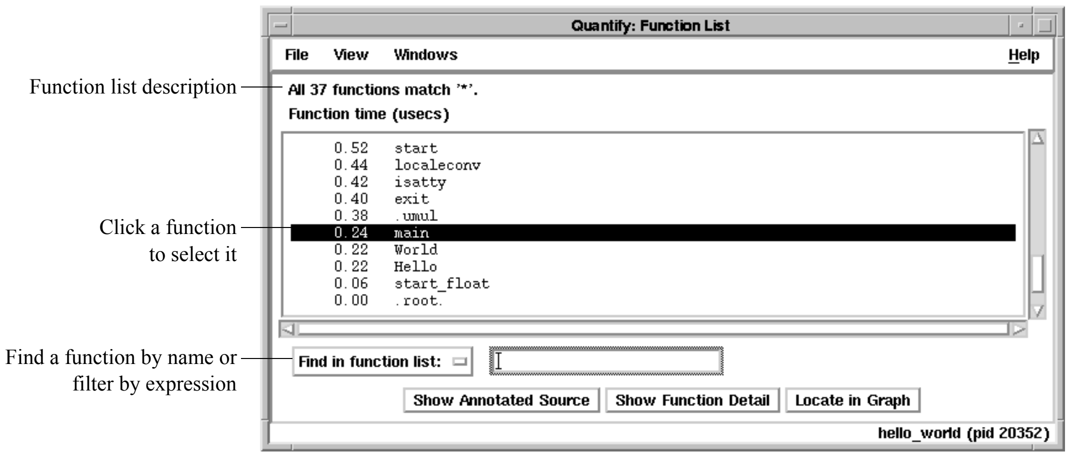
Using Quantify's data analysis windows

After transmitting the last dataset, Quantify displays the Control Panel. From here, you can display Quantify's data analysis windows and begin analyzing your program's performance.



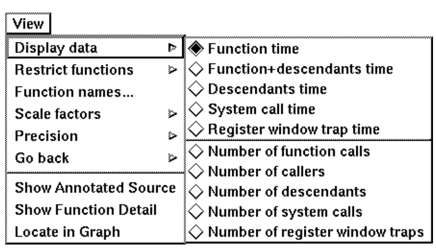
The Function List window

The Function List window, for performance profiling runs, shows the functions that your program executed. By default, it displays all the functions in your program, sorted by their *function time*. This is the amount of time a function spent performing computations (compute-bound) or waiting for system calls to complete.



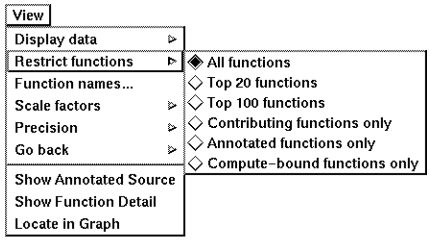
Sorting the function list

For performance data, you can sort the function list based on the various types of data Quantify collects. To do this, select **View > Display data**.



Restricting functions

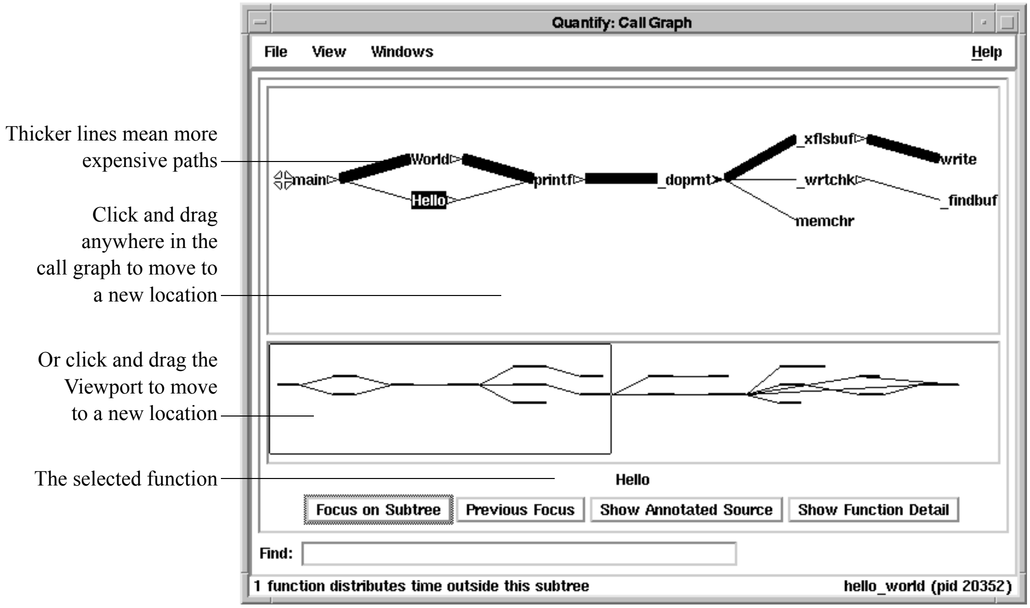
To focus attention on specific types of functions, or to speed up the preparation of the function list report in large programs, you can restrict the functions shown in the report. Select **View > Restrict functions**.



The Call Graph window

For performance profiling runs, the Call Graph window presents a graph of the functions called during the run. It uses lines of varying thickness to graphically depict where your program spends its time. Thicker lines correspond directly to larger amounts of time spent along a path.

The call graph helps you understand the calling structure of your program and the major call paths that contributed to the total time of the run. Using the call graph, you can quickly discover the sources of bottlenecks.



By default, Quantify expands the call paths to the top 20 functions contributing to the overall time of the program.

Using the pop-up menu

To display the pop-up menu, right-click any function in the call graph.

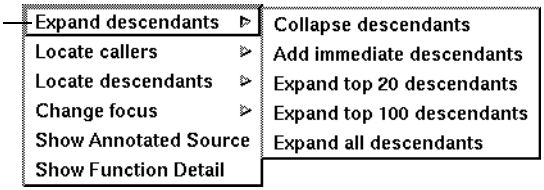
You can use the pop-up menu to:

- Expand and collapse the function’s subtree
- Locate individual caller and descendant functions
- Change the focus of the call graph to the selected function
- Display the annotated source code or the function detail for the selected function

Expanding and collapsing descendants

Use the pop-up menu to expand or collapse the subtrees of descendants for individual functions.

Select to expand or collapse descendant subtrees



After expanding or collapsing subtrees, you can select **View > Redo layout** to remove any gaps that your changes create in the call graph.

The Function Detail window

The Function Detail window, for performance profiling runs, presents detailed performance data for a single function, showing its contribution to the overall execution of the program.

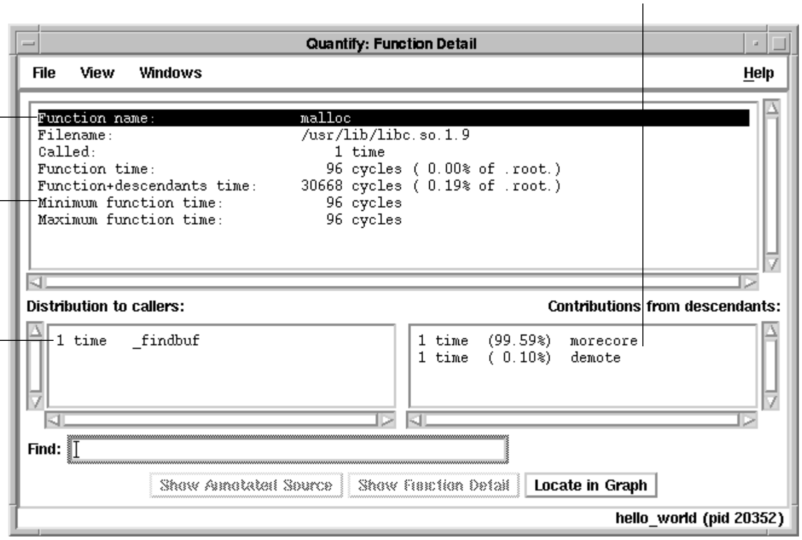
For each function, Quantify reports both the time spent in the function's own code (its *function* time) and the time spent in all the functions that it called (its *descendants* time). Quantify distributes this accumulated *function+descendants* time to the function's immediate caller.

The immediate descendants of `malloc`, and how they contributed to `malloc`'s function+descendants time

All the data collected for `malloc`

The minimum and maximum time spent in `malloc` on any one call

The functions that called `malloc`

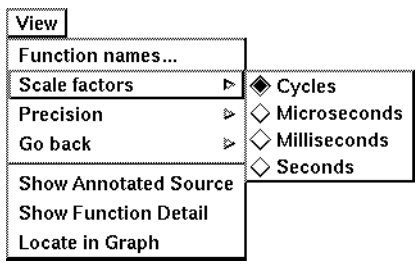


Double-click a caller or descendant function to display the detail for that function.

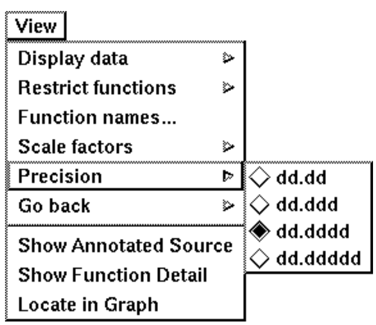
The function time and the function+descendants time are shown as a percentage of the total accumulated time for the entire run. These percentages help you understand how this function's computation contributed to the overall time of the run. These times correspond to the thickness of the lines in the call graph.

Changing the scale and precision of performance data

Quantify can display recorded performance data in cycles (the number of machine cycles) and in microseconds, milliseconds, or seconds.
To change the scale of data, select **View > Scale factors**.



To change the precision of data, select **View > Precision**.



Saving function detail data

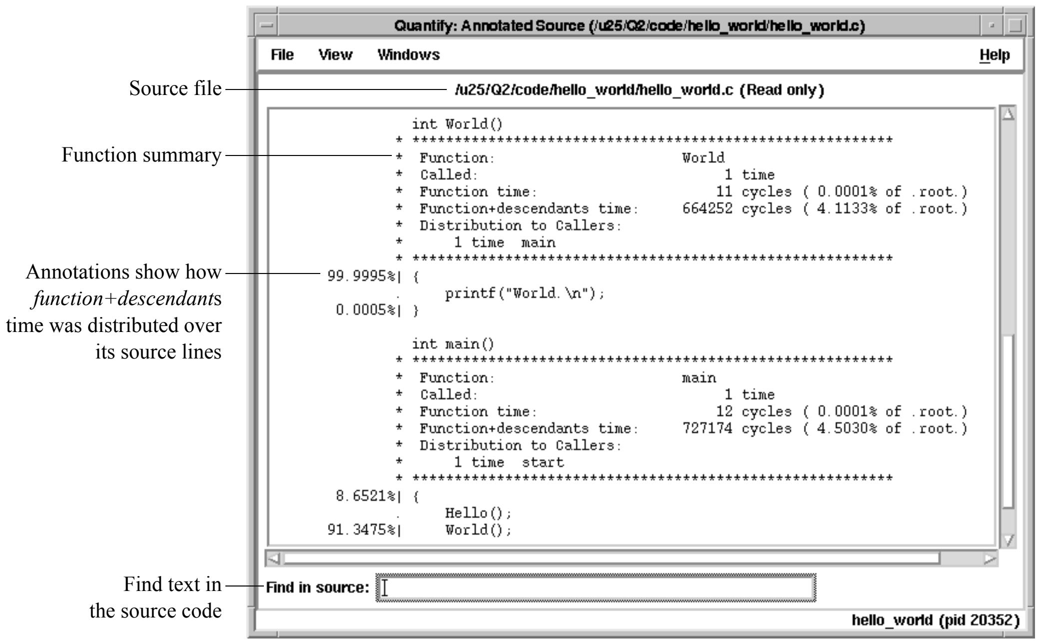
To save the current function detail display to a file, select **File > Save current function detail as**.

To append additional function detail displays to the same file, select **File > Append to current detail file**.

The Annotated Source window

Quantify's Annotated Source window presents line-by-line performance data using the function's source code.

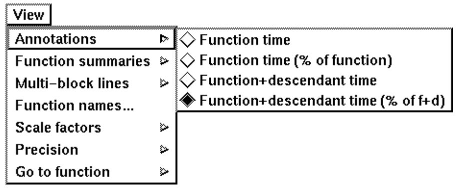
Note: The Annotated Source window is available only for files that you compile using the `-g` debugging option.



The numeric annotations in the margin reflect the time recorded for that line or basic block over all calls to the function. By default, Quantify shows the function time for each line, scaled as a percentage of the total function time accumulated by the function.

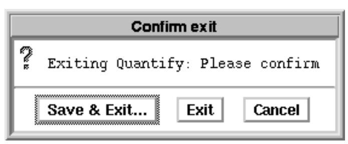
Changing annotations for performance data

To change annotations for performance data, use the **View** menu. You can select both *function* and *function+descendants* data, either in cycles or seconds and as a percentage of the *function+descendants* time.



Saving data on exit

To exit Quantify, select **File > Exit Quantify**. If you analyze a dataset interactively, Quantify does not automatically save the last dataset it receives. When you exit, you can save the dataset for future analysis.



By default, Quantify saves data to binary .qv files, and assigns file names that reflect the program name and its runtime process identifier. You can analyze a saved dataset at a later time by running `qv`, Quantify's data analysis program.

You can also save Quantify data in export format. This is a clear-text version of the data suitable for processing by scripts.

Comparing program runs with qxdiff

The `qxdiff` script compares two export data files and reports any changes in performance or memory usage.

To use the `qxdiff` script:

1 Save baseline performance or memory data to an export file. Select **File > Export Data As** in any data analysis window.

2 Change the program and run Quantify on it again.

3 Select **File > Export Data As** to export the data for the new run.

4 Use the `qxdiff` script to compare the two export data files. For example:

```
% qxdiff -i testHash.pure.20790.0.qx
improved_testHash.pure.20854.0.qx
```

You can use the `-i` option to ignore functions that make calls to system calls.

Below is the output from this example:

`qxdiff` lists the functions that have changed . . .

and summarizes the differences for the entire run

```
Differences between:
program testHash.pure (pid 20790) and
program improved_testHash.pure (pid 20854)
                     Function name          Calls      Cycles    % change
!                           strcmp         -40822    -1198640   93.77% faster
!                          putHash              0      -32912    6.61% faster
!                          getHash              0      -28376    7.86% faster
!                          remHash              0       -7856    5.91% faster
!                        hashIndex              0       10000    1.49% slower
5 differences; -1257784 cycles (-0.025 secs at 50 MHz)
25.01% faster overall (ignoring system calls).
```

Quantify options

Quantify provides command-line options for controlling operations and handling data.

Build-time options

Specify build-time options on the link line when you instrument with Quantify. For example:

```
% quantify -cache-dir=$HOME/cache -always-use-cache-dir \
cc ...
```

You can also set these options by using the `QUANTIFYOPTIONS` environment variable. For example:

```
% setenv QUANTIFYOPTIONS "-always-use-cache-dir"
```

Commonly used build-time options	Default
`-always-use-cache-dir` Specifies whether instrumented files are written to the global cache directory.	`no`
`-cache-dir` Specifies the global cache directory.	`<quantifyhome>/cache`
`-collection-granularity` Specifies the level of collection granularity.	`line`
`-ignore-runtime-environment` Prevents the runtime Quantify environment from overriding option values used in building the program.	`no`
`-linker` Specifies an alternative linker to use instead of the system linker.	system-dependent
`-use-machine` Specifies the build-time analysis of instruction times according to a particular machine.	system-dependent

qv runtime options

To run `qv`, specify the option and the saved `.qv` file. For example:

```
% qv -write-summary-file a.out.23.qv
```

qv options	Default
`-add-annotation` Specifies a string to add to the binary file.	none
`-print-annotations` Writes the annotations to `stdout`.	`no`
`-windows` Controls whether Quantify runs with the graphical interface.	`yes`
`-write-export-file` Writes the recorded data in the dataset to a file in export format.	none
`-write-summary-file` Writes the program summary for the dataset to a file.	none

Runtime options

Specify runtime options on the link line when you instrument with Quantify.

You can also set these options using the `QUANTIFYOPTIONS` environment variable. For example:

```
% setenv QUANTIFYOPTIONS "-windows=no"; a.out
```

Commonly used runtime options	Default
`-avoid-recording-system-calls` Avoids recording specified system calls.	system-dependent
`-measure-timed-calls` Specifies measurement for timing system calls.	`elapsed-time`
`-record-child-process-data` Records data for child processes created by `fork` and `vfork`.	`no`
`-record-system-calls` Records system calls.	`yes`
`-report-excluded-time` Reports time that was excluded from the dataset.	`0.5`
`-run-at-exit` Specifies a shell script to run when the program exits.	none
`-run-at-save` Specifies a shell script to run each time the program saves counts.	none
`-save-data-on-signals` Saves data on fatal signals.	`yes`
`-save-thread-data` Saves composite or per-stack thread data.	`composite`
`-write-export-file` Writes the dataset to an export file as ASCII text.	none
`-write-summary-file` Writes the program summary for the dataset to a file.	`/dev/tty`
`-windows` Specifies whether Quantify runs with the graphical interface.	`yes`

API functions

To use Quantify API functions, include *<quantifyhome>*/quantify.h in your code and link with *<quantifyhome>*/quantify_stubs.a

Commonly used C/C++ functions	Description
quantify_help (void)	Prints description of Quantify API functions
quantify_is_running (void)	Returns true if the executable is instrumented
quantify_print_recording_state (void)	Prints the recording state of the process
quantify_save_data (void)	Saves data from the start of the program or since last call to quantify_clear_data
quantify_save_data_to_file (char * filename)	Saves data to a file you specify
quantify_add_annotation (char * annotation)	Adds the specified string to the next saved dataset
quantify_clear_data (void)	Clears the performance data recorded to this point
quantify_<action>_recording_data (void)[a]	Starts and stops recording of all data
quantify_<action>_recording_dynamic_library_data (void)[a]	Starts and stops recording dynamic library data
quantify_<action>_recording_register_window_traps (void)[a]	Starts and stops recording register-window-trap data
quantify_<action>_recording_system_call (char *system_call_string)[a]	Starts and stops recording specific system-call data
quantify_<action>_recording_system_calls (void)[a]	Starts and stops recording of all system-call data

a. *<action>* is one of: start, stop, is. For example: quantify_stop_recording_system_call

Notices

4

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

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

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

Trademarks

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

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

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

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

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

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

Index