PureCoverage User's Guide

Version 4.1

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

IMPORTANT NOTICE

DISCLAIMER OF WARRANTY

Rational Software Corporation makes no representations or warranties, either express or implied, by or with respect to anything in this guide, and shall not be liable for any implied warranties of merchantability or fitness for a particular purpose or for any indirect, special or consequential damages.

COPYRIGHT NOTICE

PureCoverage, copyright © 1992-1997 Rational Software Corporation. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of Rational Software Corporation. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, Rational Software Corporation assumes no responsibility for errors or omissions. This publication and features described herein are subject to change without notice.

The program and information contained herein are licensed only pursuant to a license agreement that contains use, reverse engineering, disclosure and other restrictions; accordingly, it is "Unpublished — rights reserved under the copyright laws of the United States" for purposes of the FARs.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct 1988), FAR 12.212(a) 1995, FAR 52.227-19, or FAR 52.227-14, as applicable.

TRADEMARKS

Rational, Purify, PureCoverage, Quantify, PureLink, ClearDDTS, and ClearCase are U. S. registered trademarks of Rational Software Corporation.

All other products or services mentioned in this guide are covered by the trademarks, service marks, or product names as designated by the companies who market those products.

PATENTS

Purify, PureCoverage, and Quantify are covered by one or more of U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329. Purify is licensed under Sun Microsystems Inc.'s U.S. Pat. No. 5,404,499. Other U.S. and foreign patents pending.

Printed in the U.S.A.

Contents

Welcome to PureCoverage

Using this guide vii
PureCoverage features vii
Getting started vii
Taking advantage of special features viii
Common questions and reference material viii
Conventions used in this guide viii
Using online Help ix
Displaying the release notes ix
Installing PureCoverage ix
Contacting technical support x

1 Introducing PureCoverage

PureCoverage: simple & effective 1-2
Key PureCoverage features 1-3

2 Finding Untested Areas of Hello World

Instrumenting a program 2-1
Running an instrumented program 2-4
Program output 2-4
Coverage data 2-5
Displaying coverage data 2-5
Expanding the detail level 2-6
Examining function level detail 2-7
Examining the annotated source 2-8
Improving Hello World's test coverage 2-9
Modifying makefiles for PureCoverage 2-11

Beyond Hello World: how PureCoverage works 2-12
Files created by PureCoverage 2-12
Compiling with the debugging option -g 2-13
How PureCoverage finds source files 2-14
Covering multiple processes 2-15
Signal handling 2-17
Covering multi-threaded applications 2-17
Saturating counters 2-18

3 When to Use PureCoverage

Using PureCoverage in nightly builds 3-1
Using PureCoverage with test harnesses 3-2
Separating data for individual test runs 3-3
Combining data from multiple program runs 3-4
Discarding data from failed tests 3-6
Exporting data 3-8
Running report scripts 3-9
Using PureCoverage with other Rational Software products . . . 3-11
Using PureCoverage with Purify 3-11
Using PureCoverage and Purify with PureLink 3-12
Using PureCoverage with ClearDDTS 3-12

4 Customizing Coverage

Excluding libraries, directories, and files from coverage 4-1
Coverage for libraries 4-1
Customizing data collection 4-2
Adjusting coverage on a line-by-line basis 4-4
Adjustments 4-4
Types of adjustments 4-5
Marking adjustments manually 4-5
Marking adjustments interactively 4-10
Unnecessary adjustments 4-12
Saving files with adjustments 4-12

Removing adjustments in the Annotated Source window 4-13
Adjustment file format 4-14
Strategies for using PureCoverage adjustments 4-15
Adjustment usage considerations 4-15
Models for using PureCoverage adjustments 4-17

5 Using the PureCoverage Graphical Display
PureCoverage Viewer 5-1
Selecting items 5-1
Using the toolbar 5-2
Selecting Viewer columns 5-3
Using the Viewer File menu 5-6
Using the Viewer View menu 5-7
Using the Viewer Adjustments menu 5-8
Using the Viewer Help menu 5-8
Annotated Source window 5-9
Using the Annotated Source File menu 5-10
Using the Annotated Source View menu 5-10
Using the Annotated Source Help menu 5-11
Navigating in the Annotated Source window 5-11

6 Report Scripts
PureCoverage report scripts 6-1
Coverage summary report 6-2
Low coverage report 6-3
Low coverage mail report 6-3
Spreadsheet report 6-4
Differences report 6-5
Build differences summary report 6-6
Annotated source report 6-8
Annotated differences report 6-9
Selected tests report 6-11

Custom reports 6-15
A sample custom report script 6-15

7 PureCoverage Options

Option tables 7-1
Using PureCoverage options 7-2
PureCoverage option syntax 7-2
Using conversion characters in filenames 7-3
PureCoverage option types 7-3
PureCoverage option processing 7-4
Specifying options in environment variables 7-4
Using the PUREOPTIONS environment variable 7-4
Setting options for Purify and PureCoverage 7-5
Specifying options on the link line 7-5
Using the ignore-run-time-environment option 7-6
Using analysis-time mode options 7-7
Build-time options 7-7
Options for caching 7-7
Options for linker and collector 7-9
Run-time options 7-10
Options for file identification 7-10
An option for saving data 7-12
An option for data collection 7-12
Options for signal handling 7-13
An option for exit processing 7-14
Analysis-time options 7-15
An option for handling adjustments 7-15
An option for merging 7-15
Analysis-time mode options 7-16
Informational options 7-19

8 PureCoverage API

Calling PureCoverage API functions from your program 8-1
Calling PureCoverage API functions from a debugger 8-2
Data collection API functions 8-3

Appendixes

A Common Questions

Customizing coverage A-1
General questions A-2
Performance issues A-4

B Export Format

Export format description B-1
The effect of coverage adjustments on export B-9

3 Annotation Variations

Complex source lines C-1
Multi-line statements C-2
Function entry points C-3
Local variable declarations C-5
Switch statements C-5
exit() statements C-7
C++ inline functions C-7

PureCoverage Quick Reference

Index

Welcome to PureCoverage

PureCoverage is a test-coverage monitoring program that is both effective and easy to use. Once PureCoverage is installed, you can immediately start using it on your applications by adding the word `purecov` to your link line. For example:

```
% purecov cc -g <myprog>.o
```

or, if you compile and link at the same time:

```
% purecov cc -g <myprog>.c
```

This guide is intended to assist you as you are learning to use PureCoverage, and to serve as a reference after you have mastered the program.

Using this guide

This guide can help you get the most out of PureCoverage at all levels of use.

PureCoverage features

For an overview of what PureCoverage can do, read Chapter 1, "Introducing PureCoverage."

Getting started

To start using PureCoverage, read Chapter 2, "Finding Untested Areas of Hello World." This chapter also gives you a look into how PureCoverage works.

To learn how to use PureCoverage with your test harness or nightly builds, and how to take advantage of PureCoverage when

you are using other Rational Software products, read Chapter 3, “When to Use PureCoverage.”

To control coverage collection and adjust coverage statistics, read Chapter 4, “Customizing Coverage.”

To make full use of the features that the PureCoverage graphical user interface provides, read Chapter 5, “Using the PureCoverage Graphical Display.”

Taking advantage of special features

Chapters 6 through 8 contain information about important PureCoverage features: report scripts, options, and the application programming interface (API).

Common questions and reference material

A great investment of your time is to read through Appendix A, “Common Questions.” This appendix contains answers to the most frequently asked questions about PureCoverage.

Appendix B provides information about export format for coverage data. Appendix C shows some of the variations in coverage display that result from using different compilers.

Conventions used in this guide

- `<purecovhome>` refers to the directory where PureCoverage is installed. Whenever you see `<purecovhome>`, use the PureCoverage directory name instead. To find the PureCoverage directory on your system, use the command

```
% purecov -printhomedir
```

- `Courier` font indicates source code, program names or output, file names, and commands that you enter.
- Angle brackets `< >` indicate variables.
- *Italics* introduce new terms and show emphasis.

- This icon appears next to instructions for the Sun SPARC SunOS 4.1 operating system.

- This icon appears next to instructions for the Sun SPARC Solaris 2 operating system, also referred to as SunOS 5.

- This icon appears next to instructions for the HP-UX operating system.
- Unless otherwise noted, commands are shown using `csh(1)` syntax.
- Unless otherwise noted, debugging examples are shown using the debugger `dbx`.

Using online Help

PureCoverage provides online Help through the Help menu in each PureCoverage window.

To get online Help, click any item in the Help menu. If you click **On Context** in the Help menu, the cursor becomes a question mark (?). Click the cursor on any component of the window for specific information about that component.

Displaying the release notes

The PureCoverage `README` file is located in the `<purecovhome>` directory. You can display it by using UNIX commands, or by selecting **Release notes** from the Help menu in any PureCoverage window. The `README` file contains the latest information about this release of PureCoverage, including new features and supported hardware and software.

Installing PureCoverage

For information about licensing and installing PureCoverage, refer to the *Installation & Licensing Guide*, part number 800-009921-000.

Contacting technical support

If you have a technical problem and you can't find the solution in this guide, contact Rational Software Technical Support. See the back cover of this guide for the addresses and telephone numbers of technical support centers.

Note the sequence of events that led to the problem and any program messages you see. If possible, have the product running on your computer when you call.

For technical information about PureCoverage, answers to common questions, and information about other Rational Software products, visit the Rational Software World Wide Web site at `http://www.rational.com`. To contact technical support directly, use `http://www.rational.com/support`.

1 Introducing PureCoverage

Test coverage data is a great help in developing high-quality software, but most developers find coverage tools and the data they produce too complex to use effectively. PureCoverage is the first test coverage product to produce highly useful information *easily*.

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 portions of your code that have not been exercised by testing.

PureCoverage lets you:

- Identify the portions of your application that your tests have not exercised
- Accumulate coverage data over multiple runs and multiple builds
- Merge data from different programs sharing common source code
- Work closely with Purify, Rational's run-time error detection program, 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

One of the keys to high quality software is comprehensive testing and identification of problem areas throughout the development process. PureCoverage provides the information you need to identify gaps in testing quickly, saving precious time and effort.

If you use PureCoverage consistently during development, you will be able to ship more reliable software while meeting your aggressive schedules.

PureCoverage: simple & effective

PureCoverage is easy to use, and its output is easy to understand and analyze. PureCoverage pinpoints critical holes in your test suites; scales well to large, long-running applications; and works seamlessly with Purify. You no longer need to labor over tedious and error-prone compilation processes or hard-to-understand output.

You can install and master PureCoverage in one hour. Incorporate PureCoverage into your development process simply by adding one word, `purecov`, to your makefile and re-linking. You do not need to recompile or link special libraries. You can use PureCoverage with existing makefiles and debugging tools. Seamless integration with your development environment ensures that developers receive the most complete information quickly and painlessly.

PureCoverage provides an optional API that allows you to control data collection during the application run. The API consists of functions for configuring the data collection mechanism.

PureCoverage accumulates data over multiple runs and keeps track of the data for you. You no longer have to determine whether a changed file invalidates previous data. PureCoverage does it automatically.

PureCoverage's graphical and textual output lets you determine quickly and effectively where your tests need to be improved in order to exercise your code.

Key PureCoverage features

PureCoverage's unique capabilities include:

- **Faster analysis.** PureCoverage helps reduce time spent determining problem areas of your code. PureCoverage's outline view provides detailed, yet manageable, coverage information at the executable, library, file, function, block and line levels. A point-and-click interface provides immediate access to increasing levels of detail and lets you view annotated source with line-by-line statistics.
- **Comprehensiveness.** PureCoverage monitors coverage in *all* of the code in an application including third-party libraries, even in multi-threaded applications. It identifies critical gaps in application testing suites, providing accumulated statistics over multiple runs and multiple executables.
- **Customized output.** PureCoverage allows you to tailor coverage output to your specific needs using *report scripts*. For example, you can use the scripts to generate reports that:
  - Inform you when coverage for a file dips below a threshold
  - Show differences between successive runs of an executable, making improvements immediately obvious
  - Identify the appropriate subset of test cases that must be run to exercise the changed portions of your program

  You can use the scripts as-is, modify them, or follow them as models for your own scripts.
- **Support for your tools and practices.** PureCoverage works *behind the scenes* with standard tools and compilers. PureCoverage supports shared libraries and all standard industry compilers.

2 Finding Untested Areas of Hello World

This chapter describes how to use PureCoverage to determine which parts of your program are untested. It steps you through an analysis of a sample `hello_world.c` program, telling you how to:

- Compile and link your program under PureCoverage to *instrument* the program with coverage monitoring instructions
- Run the program to collect coverage data
- Display and analyze the coverage data to determine which parts of your program were not tested
- Improve coverage for the program
- Modify your makefiles to use PureCoverage throughout your development cycle

The chapter concludes with a behind-the-scenes look at how PureCoverage works, and a discussion of how PureCoverage handles more complex programming situations.

Instrumenting a program

Begin your analysis of the sample `hello_world.c` program by copying the program file into your working directory. Then instrument the program with PureCoverage and run it.

When you work through this example on your own system, remember that there is no single standard compiler. Different compilers produce slightly different output. Do not be concerned about minor differences between what you see on your monitor and what you see in this guide.

1 Create a new working directory. Go to the new directory, and copy the `hello_world.c` program and related files 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);
}
```

3 Compile, using the `-g` debugging option, and link the program. Then run the program:

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

Verify that it produces the expected output:

```
Hello, World
```

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

4 Now add `purecov` at the beginning of the compile-and-link line:

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

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

```
PureCoverage 4.1 Solaris 2, Copyright 1994-1997 Rational Software Corp.
All rights reserved.
Instrumenting: hello_world.o Linking
```

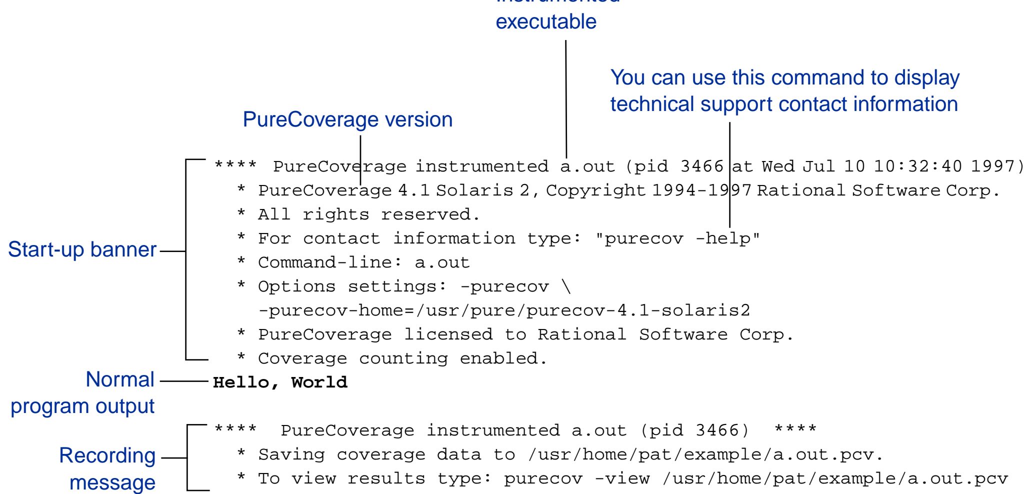
Note: When you compile and link in separate stages, add `purecov` only to the link line.

Running an instrumented program

1 You now have a PureCoverage-instrumented executable. Run it:

```
% a.out
```

Typical output is:



Program output

In addition to the PureCoverage start-up banner and recording message, the program produces its normal output, just as if it were not instrumented. For programs that do not ordinarily produce console output, PureCoverage displays only the start-up banner and the recording message.

You can redirect all PureCoverage output to a file by using the `-log-file` option. For more information about this and other PureCoverage options, see Chapter 7.

Coverage data

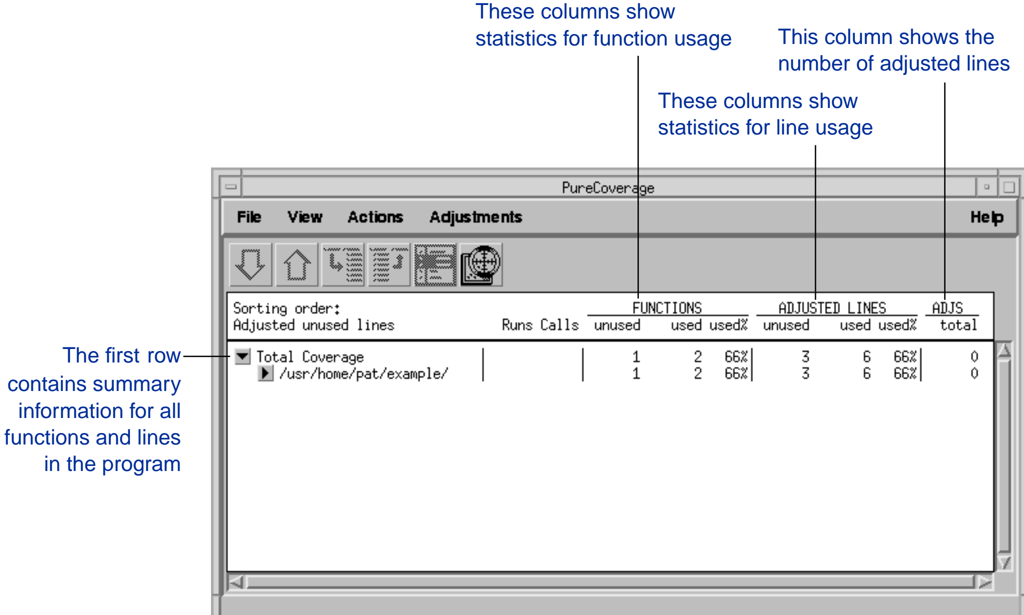
When the program `a.out` completes execution, PureCoverage writes coverage information for the session to the file `a.out.pcv`. Each time the program runs, PureCoverage updates this file with additional coverage data.

Displaying coverage data

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

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

The PureCoverage Viewer appears:



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. See “Adjusting coverage on a line-by-line basis” on page 4-4.

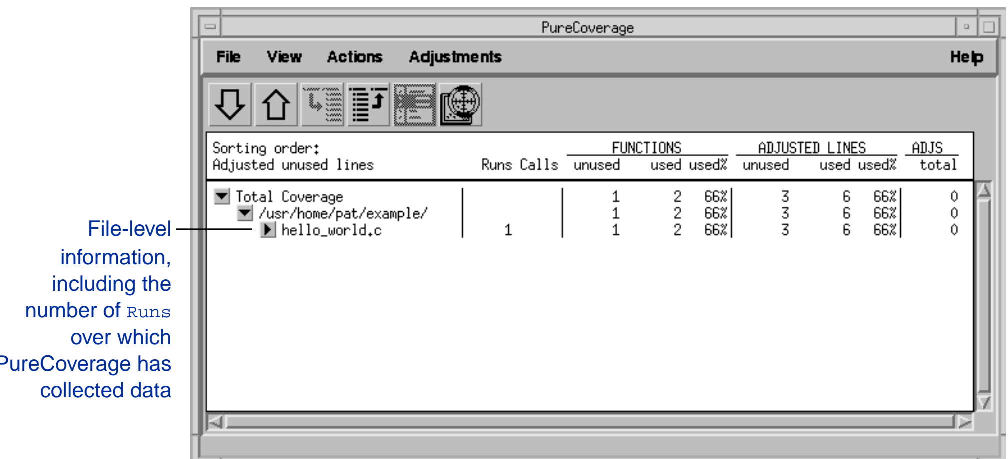
The ADJS column, to the right of the ADJUSTED LINES column, contains zeroes, which indicates that this sample does not actually include adjustments.

The row below Total Coverage displays function and line information for each source directory. In this example, there is only one source directory, so the information displayed for the directory is identical to the information in the Total Coverage row.

For more information about displaying coverage data in the Viewer, see page 5-3.

Expanding the detail level

2 Click the ▶ button to the left of the example row. The display expands to show *file-level* information for the directory, and the button changes to ▼:



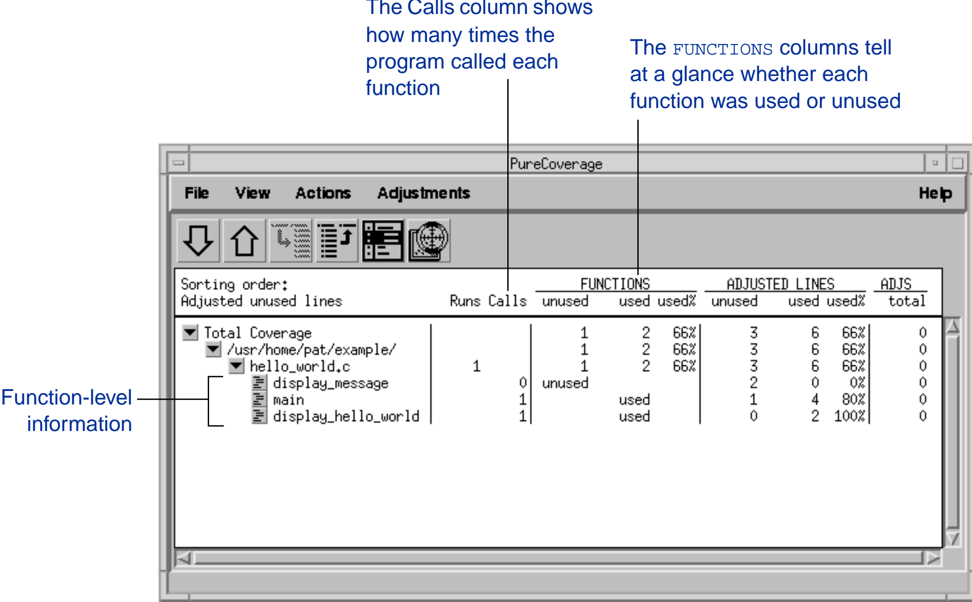
You used only one file in the example directory to build a.out. Therefore the FUNCTIONS and ADJUSTED LINES information for the

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 which have been rebuilt with some changed files, the number of runs can be different for each file.

Examining function level detail

3 Expand the `hello_world.c` row to show *function-level* information.



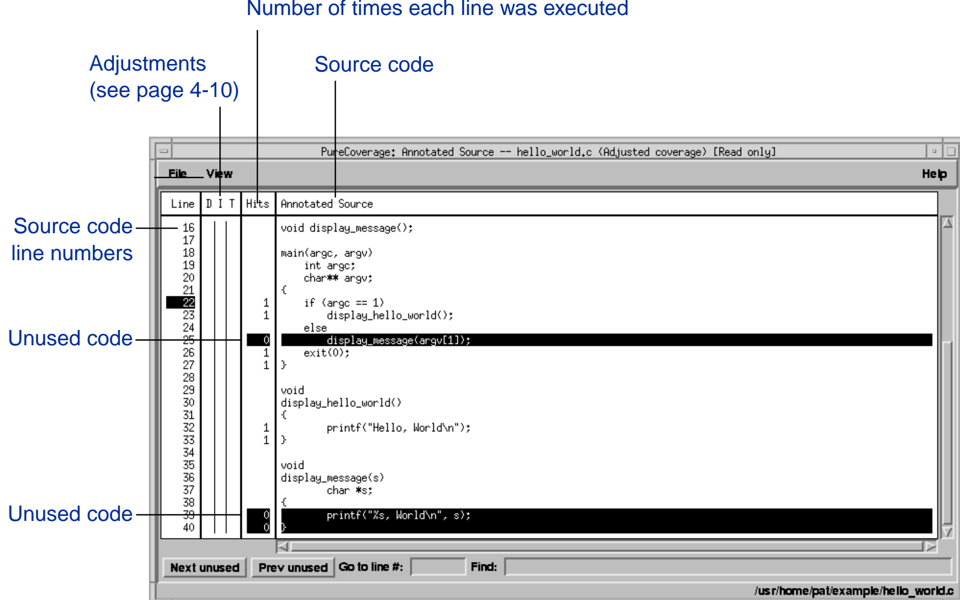
The Viewer shows coverage information for the functions `display_message`, `main`, and `display_hello_world`.

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. For details, see “Coverage for system libraries” on page 4-2.

Examining the annotated source

4 To see the source code for `main` annotated with coverage information, click the button to the left of `main` in the Viewer:

The Annotated Source window appears.



PureCoverage highlights code that was not used when the program ran. In this file only a few 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:

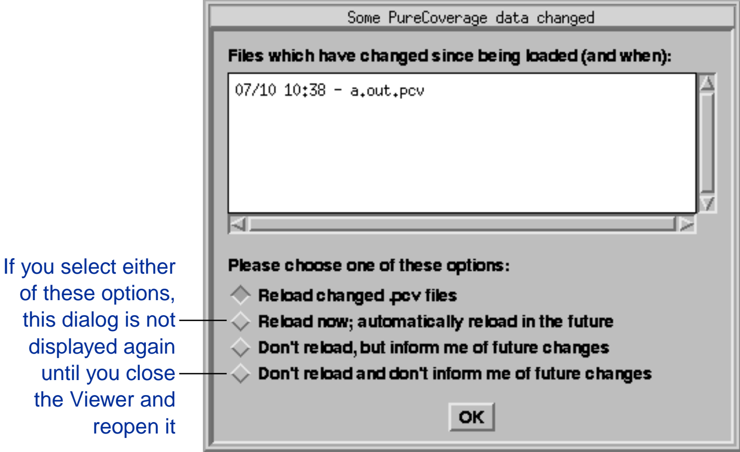
1 Without exiting PureCoverage, run the program again, this time with an argument:

```
% a.out Goodbye
```

Here is the output:

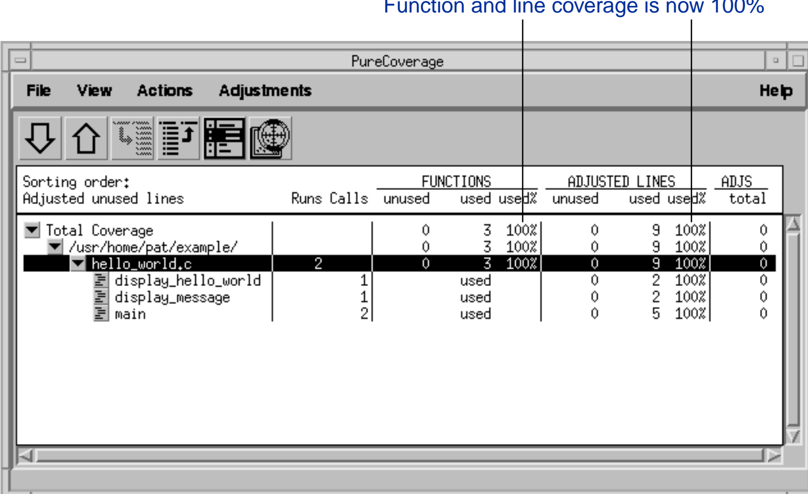
```
****  PureCoverage instrumented a.out (pid 17331 at Wed Jul 10 10:38:07 1997)
 * PureCoverage 4.1 Solaris 2, Copyright 1994-1997 Rational Software Corp.
 * All rights reserved.
 * For contact information type: "purecov -help"
 * Command-line: a.out Goodbye
 * Options settings: -purecov \
   -purecov-home=/usr/pure/purecov-4.1-solaris2
 * PureCoverage licensed to Rational Software Corp.
 * Coverage counting enabled.
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
```

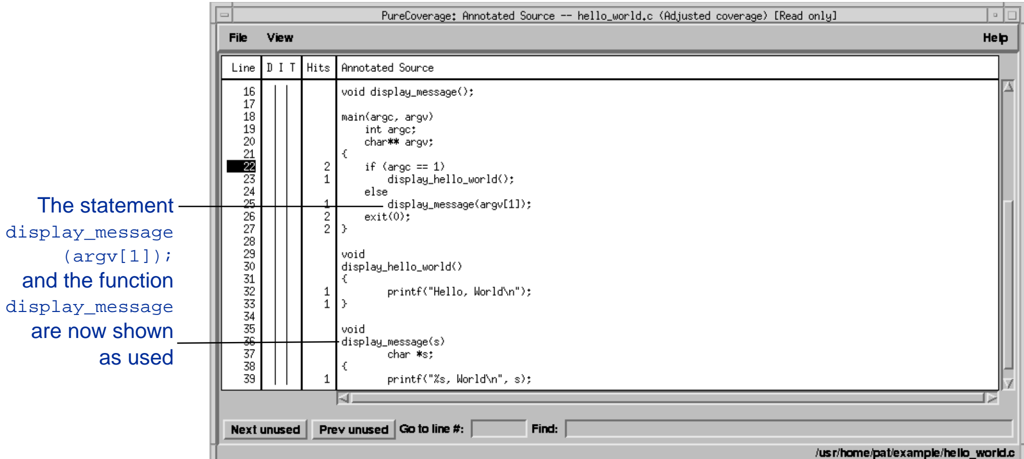
2 A dialog appears with options for handling coverage data from this and future runs. Select **Reload changed .pcv files** and click **OK**.



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:

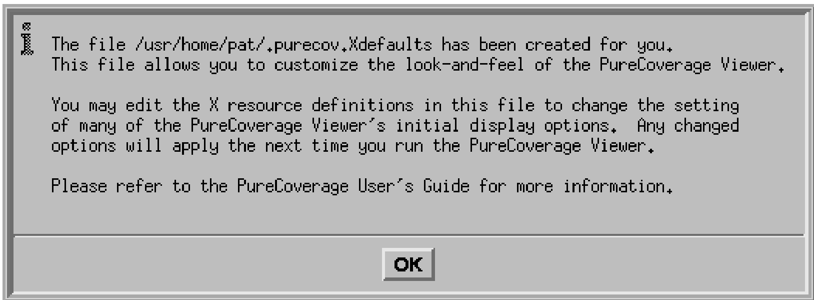




Note: If you still have untested lines, it is possible that your compiler is generating unreachable code. See Chapter 4, “Customizing Coverage,” for ways to handle this.

3 To exit PureCoverage, select **Exit** from the Viewer's File menu.

Note: The first time you exit, PureCoverage displays information about your default settings. Click the **OK** button to continue.



For information about the `.purecov.Xdefaults` file, see page 5-8, or read the instructions in the `.purecov.Xdefaults` file.

Modifying makefiles for PureCoverage

You can start building your applications with PureCoverage by adding just one word, `purecov`, in front of the link line in your makefile.

```
a.out: hello_world.c
        purecov cc -g hello_world.c
```

You can also create separate targets for instrumented and non-instrumented executables by adding a few lines to the makefile. The sample makefile `hello_world.Makefile.simple`, located in the `<purecovhome>/example` directory, illustrates how to do this:

```
# Sample makefile template
# May also be used in instrumenting Hello World with PureCoverage
# Use make -f hello_world.Makefile.simple a.out.pure

a.out: hello_world.c
        cc -g -o a.out hello_world.c

a.out.pure: hello_world.c
        purecov cc -g -o a.out.pure hello_world.c
```

Beyond Hello World: how PureCoverage works

You have now seen PureCoverage in action. This section discusses how PureCoverage works so that you can use the program more effectively.

Files created by PureCoverage

PureCoverage inserts usage-tracking instructions into the object code of your application. After the compiler creates the object files for your application, PureCoverage instruments the object files, using Object Code Insertion (OCI) to add the monitoring instructions. The instrumented object files, or *cache files*, are given new names so that your original object files are not modified.

PureCoverage passes the cache files, complete with the instrumented versions of any libraries required for the application, to the linker, in place of the original object files.

The cache file names always include `pure` and an encoded PureCoverage version number. The names can also include information about the size of the original file, or the name and number of the operating system.

PureCoverage stores the cache files in the same directories as their original counterparts, unless those directories do not have write permission. In this case, PureCoverage creates shadow directories in the central cache, `<purecovhome>/cache`, where it stores the cache files.

Note: To force PureCoverage to store *all* cache files in the central cache, use the `-always-use-cache-dir` option. For more about caching, see “Options for caching” on page 7-7.

You can clean up and remove old cached files with the script `<purecovhome>/pure_remove_old_files`. It is safe to remove these files because PureCoverage rebuilds them as needed. By removing them, you gain disk space, but lose some time during linking.

To remove all cache files that have been in the system 14 days or longer, use:

```
% pure_remove_old_files / 14
```

To remove all of the cache files in the current directory and subdirectories, useful in `clean` targets of makefiles, use:

```
% pure_remove_old_files . 0
```

To remove the cache files periodically, add a cron job. For example, to remove files that have not been accessed in two weeks, add an entry to your crontab file:

```
15 2 * * *  <purecovhome>/pure_remove_old_files / 14
```

This runs `pure_remove_old_files` every day at 2:15 a.m. and, starting at the root directory, removes all cached files that have not been read in the last 14 days.

Compiling with the debugging option -g

PureCoverage adds an instruction sequence for each function to record the number of times the function is entered.[1] PureCoverage always provides the same summary information for the functions in your program, as well as for the files and directories, regardless of which compilation options you use.

When you compile code using the `-g` debugging option, the compiler includes debugging information in the resulting object file, relating source line numbers to the program instructions. In this case, PureCoverage adds a count sequence at the beginning of each *basic block* of code.

A basic block is an indivisible sequence of instructions always executed together in succession. PureCoverage uses this

---

1. The exact location of the instruction sequence, at or near the entry point of the function, depends on your compiler.

information to generate an annotated source listing that shows the lines that are tested and untested within a function.

If you compile code *without* the `-g` option, PureCoverage does not provide coverage data below the function level. It cannot annotate source code with line coverage data because there is no way to tell which program instructions relate to which lines. Although you can display summary information for this code in the Viewer, you cannot open the Annotated Source window for it.

Note: Do not use PureCoverage to instrument files that were compiled with both `-g` and `-O` at the same time. The `-O` option changes instruction ordering, so that data for line counts can be misleading.

How PureCoverage finds source files

PureCoverage uses full pathnames to identify source files. This allows it to distinguish between like-named files in different directories.

PureCoverage ordinarily determines the location of source files when you instrument them. If you move source files to a different directory, however, PureCoverage cannot automatically identify them. In addition, some compilers do not record enough information in the object files for PureCoverage to determine where the source files are located.

For this reason, PureCoverage is designed so that it can also determine the location of a source file when you view coverage data for the file.

At instrumentation time

If the compiler encodes a full pathname into the object file, and if that pathname is valid, PureCoverage attributes counts to the specified file.

If the compiler provides only a file basename or if the full pathname is invalid, and if you specify the option `-user-path`,

PureCoverage looks for the file in each of the directories specified in `-user-path`.

PureCoverage automatically appends both the current working directory and the directory where the object file resides to the end of the `-user-path` directory list. This helps PureCoverage find source files with missing or incorrect directory names.

For details on the option `-user-path`, see “Options for file identification” on page 7-10.

At view time

If the filename determined at instrumentation time still exists, PureCoverage uses it to generate annotated source listings. If the filename no longer exists:

- PureCoverage attempts to find the file’s basename in each directory specified with the option `-user-path`. It automatically appends the current working directory to the `-user-path` directory list.
- If PureCoverage still cannot find the file, it displays a dialog asking you for the location of the source file. If PureCoverage can open the file in the directory you specify, it appends the directory to the list in `-user-path` and uses it for subsequent searches.

Note: PureCoverage outputs the filename determined during instrumentation in exported counts data. For details on exporting data, see “Exporting data” on page 3-8 and Appendix B, “Export Format.”

Covering multiple processes

Using fork

By default, PureCoverage does not accumulate coverage data for child processes, since a common reason for forking is to execute a new process right away.

You can specify the `-follow-child-processes` option if you want PureCoverage to accumulate coverage data for the child process. When this option is set and the child process exits or executes another process, PureCoverage writes the accumulated counts in the child to the `.pcv` file. Likewise, when the parent process exits, PureCoverage writes counts accumulated for the parent to the `.pcv` file.

By default, when this option is set, PureCoverage combines coverage data for both the parent and child in the same `.pcv` file.

To separate the counts for parent and child, you can use the option `-counts-file=%v.%p.pcv` to specify a counts file including the process `pid` in the filename, or call the API function `purecov_set_filename()` in either or both the parent or child process to control where the data is written.

Using exec

When an instrumented program calls `exec` or related functions with a valid executable filename specified in the path argument, PureCoverage accumulates the coverage data in the `.pcv` file for the calling process before the `exec` is attempted. The counts are then set to zero. If the `exec` fails, and the calling process continues to run, the counts are correctly updated when the process finally exits or does a successful `exec`.

Note: If the `exec` is done from a child process and you have not set the `-follow-child-processes` option, no coverage data is accumulated, and no data is written at the time of the `exec`.

The process invoked by `exec` is a new process, and does not share coverage information or status with the calling process. If the invoked process is itself a PureCoverage instrumented program, it starts accumulating coverage data regardless of whether the program calling `exec` was instrumented, and whether you set the option `-follow-child-processes`.

Using vfork

A child process started with `vfork` shares a common address and accumulated coverage data with its parent. PureCoverage counting is *not* disabled in the child process of a `vfork`, even if you set the option `-follow-child-processes=no`.

Signal handling

PureCoverage installs a signal handler for many of the possible software signals that can be delivered to an instrumented process. The signal handler prints an informative message and saves coverage data to the `.pcv` file in case the process crashes.

The `-handle-signals` and `-ignore-signals` options specify which signals are handled. For details about these options, see "Options for signal handling" on page 7-13.

The signal handler installed by PureCoverage outputs a signal message to `stderr`, or to the Purify Viewer if the process is Purify'd. If the signal is a fatal signal such as a `segv`, and the program has not installed a user signal handler to catch such a condition, PureCoverage writes the coverage data to the `.pcv` file before normal signal termination. If the instrumented program has installed a signal handler, the PureCoverage handler passes control to that handler instead.

If you do not want to save data for a process that crashes, you can specify the signal name as the value of the `-ignore-signals` option. Or you can have your program install a handler of its own that invokes the API function `purecov_disable_save()` or `purecov_set_filename()` to specify an alternate destination for the data from the failed application.

Covering multi-threaded applications

PureCoverage supports multi-threaded applications.

PureCoverage writes coverage data whenever any of the lightweight process threads invokes `exit` or `exec`. Those system

calls terminate all the threads at the same time, so the total coverage data is recorded for the combination of threads.

PureCoverage options and API functions do not control separate recording of coverage data by an individual thread. This is because all the threads share the coverage counters.

Saturating counters

PureCoverage uses saturating counters to record the number of times a function or line has been executed. These counters *saturate* when their value reaches 9999. All counts of 10,000 and more are represented as `10K+` in the PureCoverage Viewer and as `10000` everywhere else.

3 When to Use PureCoverage

The PureCoverage Viewer and Annotated Source window, introduced in Chapter 2, are useful for interactive analysis of the coverage data. However, for many applications, other approaches to accessing and handling coverage data are more practical.

The chapter includes:

- Examples of how to use PureCoverage with your nightly builds and test suites
- Basic information about exporting coverage data and using PureCoverage report scripts
- Information about using PureCoverage with other Rational Software products

Using PureCoverage in nightly builds

You can use PureCoverage to collect coverage data and distribute reports automatically as part of your nightly build procedure.

Take as an example two hypothetical programs, `prog1` and `prog2`, which share some code segments. The programs are recompiled from scratch each night and then a test suite is run to exercise each program multiple times. Results from the tests are collected and distributed automatically.

The section of the makefile responsible for recompiling and linking your application, before you start using PureCoverage, looks like this:

```
all: prog1 prog2

prog1: prog1.o shared.a
      cc -o prog1 prog1.o shared.a

prog2: prog2.o shared.a
      cc -o prog2 prog2.o shared.a
```

This section is used both for automated nightly builds and for routine daytime builds. You can change it so that the nightly builds run in PureCoverage mode while the daytime builds remain unchanged. For example:

```
all: prog1 prog2

prog1: prog1.o shared.a
        $(PURECOV) cc -o prog1 prog1.o shared.a

prog2: prog2.o shared.a
        $(PURECOV) cc -o prog2 prog2.o shared.a
```

Change the *nightly* compile command line from:

```
% make all
```

to:

```
% make PURECOV=purecov all
```

Do not change the command line for the daytime builds.

PureCoverage now collects coverage data for each test as the applications run each night.

Using PureCoverage with test harnesses

When the programs are instrumented with PureCoverage, the coverage data is automatically accumulated for each execution of each program. PureCoverage generates the files `prog1.pcv` and

prog2.pcv. You do not need to change the test harness to incorporate the basic data collection from PureCoverage.

However, there are several reasons to consider changing the harness:

- To control whether the coverage data is collected for each test separately or for the entire test suite
- To discard coverage data from failed tests
- To send coverage data reports automatically when the test suite is finished

Separating data for individual test runs

By default, PureCoverage accumulates coverage data for the various executions of each program in the files prog1.pcv and prog2.pcv. These files reside in the same directory as the programs prog1 and prog2. As each session completes, PureCoverage adds coverage data for that session to the application's default .pcv file. At the end of the test harness, the files contain the *cumulative* coverage data for the entire test suite.

Note: PureCoverage .pcv files are often in existence at the beginning of a test suite, as for example when they are left over from the previous night's run. If this is the case, PureCoverage combines the existing data and the new data in the .pcv file.

To collect separate data for each test rather than accumulating the data over multiple program runs, specify the option -counts-file=<filename> in the environment variable PURECOVOPTIONS when running the application.

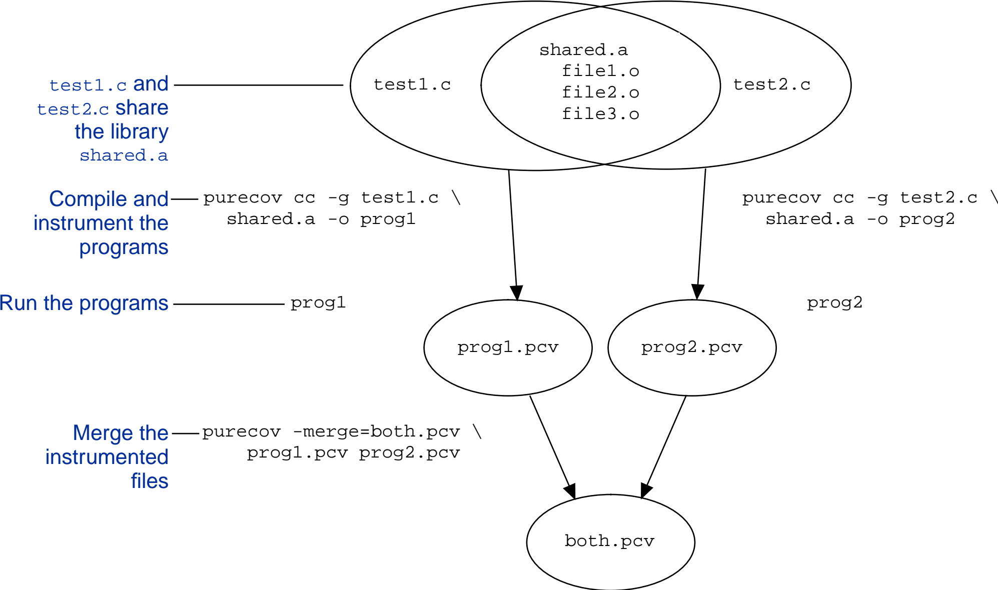
```
csh % setenv PURECOVOPTIONS -counts-file=prog1.test1.pcv; \
         prog1 < test1.input

sh $ PURECOVOPTIONS=-counts-file=prog1.test1.pcv \
         prog1 < test1.input
```

You can also use the PureCoverage API to specify the name of the output file. This is particularly useful when you want to get

information about particular sections of a program. See Chapter 8, “PureCoverage API,” for details.

Combining data from multiple program runs

Since coverage data is collected on a program-by-program basis, the coverage data for the library `shared.a` is split between the two programs. To combine the coverage data for the programs, you can use the `-merge` option. For example:



You can use the Viewer to examine `both.pcv`. It shows the combined data for the shared portion of the code. To open the Viewer, use the command:

```
% purecov -view both.pcv
```

Note: You do not have to use the `-merge` option if you plan to examine the data in the Viewer. You get the same results with the command `purecov -view prog1.pcv prog2.pcv`. If, however, you

have a large number of files and plan to run multiple Viewer sessions, using the `-merge` option is more efficient.

You can use the `-merge` option to combine coverage data for separate test cases that were stored separately. For example:

```
% purecov -merge=tests.pcv test1.pcv test2.pcv test3.pcv
```

How PureCoverage discards data

Suppose that the compilation fails and that your test suite runs tonight's version of `prog1` but last night's version of `prog2`, so that some of the shared code is from tonight and some is from last night. PureCoverage tracks the version of each program internally, using an object-code checksum scheme to detect whether the code has been changed.

Before combining sets of data for an object file, PureCoverage looks at the checksums of each version of the object file that is associated with a set of data. If the checksums match, PureCoverage treats the sets of data as generated from the same version and combines them. Otherwise, it uses the newer data and discards the older data. This way, when you examine the results for shared code, you can be sure you are seeing only valid data.

PureCoverage merges or discards data on an object-file basis. If you change one part of an application, and then recompile and make additional runs, PureCoverage discards data only for the files containing changed code and continues to accumulate data for the rest of the application.

If PureCoverage discards data when you merge `.pcv` files, it prints a warning:

```
PureCoverage 4.1 Solaris 2, Copyright 1994-1997 Rational Software Corp.
PureCoverage licensed to Rational Software Corp.
Warning: some data ignored because they were from older versions.
```

Check the `Runs` column in the Viewer for each file to note the number of runs for which the data has been collected. If the `Runs` value differs from file to file, either data for multiple applications sharing some code has been merged together, or data has been

discarded for earlier runs of a changed file included in the application.

You can display columns in the Viewer that show you the dates of the oldest and newest data for each directory and file; see “Optional columns” on page 5-4. This is often helpful when the `Runs` figures differ from file to file.

How to force PureCoverage to merge data

You can use the `-force-merge` option to force PureCoverage to accumulate data even when it appears to be from different versions of the program. This is useful if you build into your program timestamps or other data that varies from one compilation to the next. In these cases, you know that the instructions and line-numbers are unchanged between compilations, and that you can safely merge the results. For details, see “An option for merging” on page 7-15.

Note: When PureCoverage computes checksums, it ignores the timestamps that the compiler automatically inserts into object files.

Unless you are absolutely certain that the two versions were both built from precisely the same source, you should avoid using the `-force-merge` option. Otherwise, you can end up with meaningless data.

Discarding data from failed tests

PureCoverage cannot tell if a run of your program succeeded or failed. By default, PureCoverage accumulates the data for each run.

To discard data from failed test runs, you can:

- Discard data using the test harness
- Discard data from within the test program

Discarding data using the test harness

When a given test finishes and your test harness determines that the test has failed, the test harness can delete the appropriate `.pcv` file. This prevents PureCoverage from accumulating data from the failed tests. This is useful if you are keeping separate data for each test case.

To discard data using the test harness, you can change your test script from this:

```
#!/bin/sh
if test_program
then
      echo "test passed!"
else
      echo "test failed!"
fi
```

to this:

```
#!/bin/sh
if test_program
then
      echo "test $name passed!"
      mv test_program.pcv $name.pcv
else
      echo "test $name failed!"
      echo "cleaning up coverage data...."
      rm test_program.pcv
fi
```

At the end of the test suite, you can view the coverage data for a single test, `test1`:

```
% purecov -view test1.pcv
```

or you can see the combined results of the passing tests:

```
% purecov -view *.pcv
```

You can use the `-run-at-exit` option to provide an alternate method of preventing the accumulation of data from failed tests. This method has several advantages: It requires no change in the

test harness, and the exit script runs only with the version of the test program that has been instrumented by PureCoverage. See page 7-14 for details about this option.

Discarding data from within the test program

If you do not collect separate data for each test, you can modify the programs to use the PureCoverage API so that data for the failed tests is discarded.

For example, you can use `purecov_disable_save()` to disable saving data when a test program detects a failure condition. Or you can call `purecov_set_filename("pass.pcv")` right before a successful exit to arrange that data for passing tests ends up in `pass.pcv` while all other data ends up in the default file.

You can even choose to discard coverage data for only the failed portion of your program, and to retain the data for the rest of the run. See Chapter 8, "PureCoverage API."

Exporting data

Once PureCoverage has collected the data, you can convert it to PureCoverage export format. To export data to a file, use the syntax:

```
% purecov -export=<filename> prog1.pcv prog2.pcv prog3.pcv...
```

To supply export data to a script for processing, use the syntax:

```
% purecov -export prog.pcv | <report_script>
```

See also "`-export`" on page 7-17 and Appendix B, "Export Format."

Running report scripts

Report scripts let you process PureCoverage data for viewing in a variety of plain-text formats.

PureCoverage comes with a set of ready-to-use report scripts. You can use them just as they are to generate reports, modify them, or use them as models for your own scripts.

PureCoverage's ready-to-use report scripts provide the following reports:

- An overall summary report similar to the information displayed in the Viewer: `pc_summary`
- A report about files that have coverage below a specified minimum percentage: `pc_below`
- A low-coverage report sent by e-mail to the person who last modified the insufficiently tested files: `pc_email`
- A conversion of the coverage information into a form suitable for import by various spreadsheet programs: `pc_ssheet`
- A report listing files for which coverage has changed: `pc_diff`
- A comparative summary of PureCoverage data from two nightly builds: `pc_build_diff`
- An annotated source text file: `pc_annotate`
- A report showing the output of `diff`, with PureCoverage annotations, for modified source code: `pc_covdiff`
- A report identifying the subset of tests required to exercise modified source code: `pc_select`

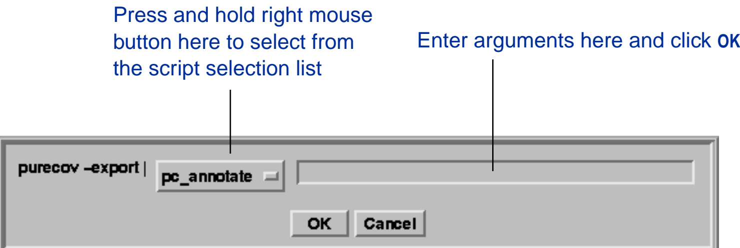
For a detailed description of these reports and their syntax, see Chapter 6.

To write your own scripts, see "Custom reports" on page 6-15.

The PureCoverage report scripts are located in the directory `<purecovhome>/scripts`. To run scripts from a shell or makefile, you can include this directory on your `$PATH`, make symbolic links

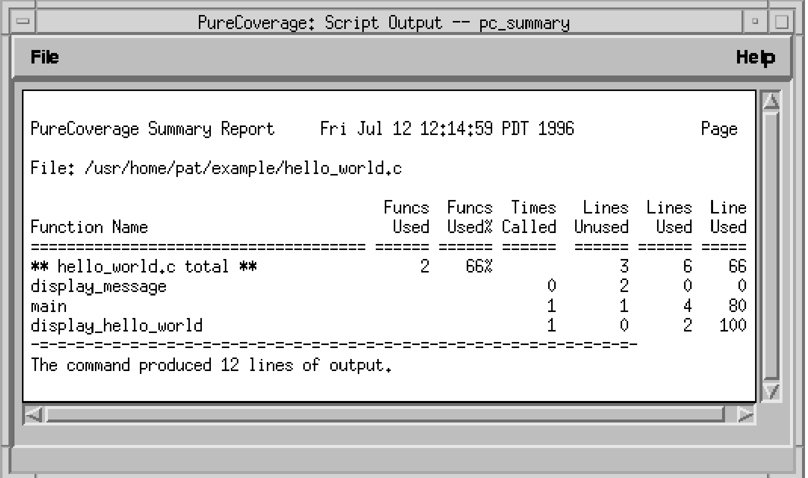
from `/usr/local/bin` to the script, or simply invoke the scripts by absolute pathname.

You can also run the report scripts from within the Viewer. Select **Run Script** from the File menu. This opens the Script dialog:



Select the script name from the script selection list. Enter any arguments, being sure to escape any character that has a special meaning to the shell. The script works on the data files currently being viewed, and shows the adjustments that are currently displayed.

The Script Output window displays the results:



You can also run *custom* scripts in the Viewer, provided that they use `stdin` as the source of their export data. Install your custom script in the directory `<purecovhome>/scripts`. The next time you start the Viewer, the custom script name appears in the Script dialog, where you can select it.

To run the `pc_diff` or `pc_build_diff` script from the Viewer, use "`-`" (for `stdin`) as one of the arguments, and the name of the file to be compared as the other. For details, see "Differences report" on page 6-5 and "Build differences summary report" on page 6-6. You cannot run the `pc_covdiff` script from the Viewer.

A report script that you run from the dialog produces output in the Script Output window that is similar to the output when you invoke scripts using the command:

```
% purecov -export prog.pcv | <report_script>
```

To save a report, select **Save as . . .** from the File menu in the Script Output window.

Using PureCoverage with other Rational Software products

You can easily use PureCoverage with other Rational Software products such as Purify, PureLink, and ClearDDTS.

Note: The instruction sequences that PureCoverage inserts during instrumentation are incompatible with the instruction sequences inserted by Quantify, Rational Software's performance analysis product. You cannot use PureCoverage and Quantify at the same time.

Using PureCoverage with Purify

PureCoverage is designed to work closely with Purify, Rational Software's run-time error detection application. Use PureCoverage with Purify to improve coverage for your test cases while verifying that the tests do not have memory access errors or memory leaks. PureCoverage identifies the parts of your program that have not yet been tested and Purify'd.

To use PureCoverage with Purify, add both product names to the beginning of your link line. Include all options with the programs to which they refer. For example:

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

For information about the order in which the Purify and PureCoverage options are applied, see "PureCoverage option processing" on page 7-4.

When you run your program, you see the Purify banner and the PureCoverage banner. Purify reports memory access errors and memory leaks as the program runs. You can examine test coverage data after the program terminates. You can also open the PureCoverage Viewer from the Purify Viewer by clicking the PureCoverage icon in the toolbar.

Using PureCoverage and Purify with PureLink

Both PureCoverage and Purify are designed to work with PureLink, Rational Software's incremental linking tool. Always enter `purelink` as the first word on the link line.

To use PureCoverage with PureLink, type:

```
% purelink <purelinkoptions> \
      purecov <purecovoptions> cc -g \
      hello_world.c -o hello_world
```

To use PureCoverage with Purelink and Purify, type:

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

Using PureCoverage with ClearDDTS

If ClearDDTS, Rational Software's defect-tracking tool, is installed at your site and in your path, you can start it directly from the PureCoverage Viewer. Click the ClearDDTS icon in the toolbar or select **Start ClearDDTS** from the File menu.

Customizing Coverage

This chapter discusses two ways of customizing coverage:

- Excluding libraries, directories, and files from coverage
- Adjusting coverage on a line-by-line basis

Excluding libraries, directories, and files from coverage

This section tells you how PureCoverage collects coverage data for different types of libraries. It also explains how you can customize coverage by excluding libraries, directories, and individual files.

Coverage for libraries

PureCoverage provides different coverage information for third-party, shared, and system libraries.

Coverage for third-party libraries

The data that PureCoverage collects includes coverage data for third-party libraries. The degree of detail available depends on how the library code is compiled. Typically, debugging data is not available for third party libraries. Therefore, PureCoverage accumulates only function-level summary data. Even when the code is compiled with debugging data, source code is not usually available, and so the Annotated Source window cannot display it.

Coverage for shared libraries

PureCoverage collects data for shared libraries in the same way as for other code. It records coverage at the function level for code without debugging data. It also collects coverage data at the line level when debugging data is available.

Coverage for system libraries

The coverage data for system libraries such as `libc` and `libm` is usually low and often meaningless. By default, PureCoverage does not collect coverage data for system libraries in standard directory locations.[1] PureCoverage accomplishes this with `exclude` directives in the default `.purecov` and `.purecov.<platform>` files.

Customizing data collection

You can use the `exclude` directive to disable coverage insertion for files and directories. This is useful for avoiding instrumentation of files for which you do not need coverage data, such as system libraries. By excluding such files, you also improve run-time performance.

Excluding coverage by filename or directory

PureCoverage reads the `exclude` directives from the `.purecov` and `.purecov.<platform>` files at build-time. PureCoverage looks for these files in:

- The current directory
- Your home directory
- The `<purecovhome>` directory

These files contain directives such as:

```
exclude /usr/lib
```

This directive disables coverage instruction for files whose pathname begins `/usr/lib`, such as `/usr/lib/libc.a`.

To see which files are excluded from instrumentation by default, look at the `.purecov` and `<purecov.platform>` files. You can

---

1. PureCoverage instruments libraries even when they are excluded, but the instrumentation is only a *partial* instrumentation, required to maintain the necessary registers. The overhead for partial instrumentation is minimal.

change these files and re-link your application to include coverage data for any system libraries of your choice.

Note: Since `exclude` directives are read at build time, you need to re-link your program for new directives to take effect.

PureCoverage creates canonical versions of all pathnames by expanding all symbolic links, so that the same file does not appear under multiple names in the coverage data. The `exclude` directives must match the canonical names; `exclude` directives that refer to the symbolic link names do not work.

This is especially an issue when your program obtains site-wide libraries from a remote file system via a generic pathname that is a symbolic link to a specific installation directory. The best way to handle this problem is to use wildcards (`*` or `?`) in the `exclude` directive for the root part of the pathname.

For example, you can find the real name of the library you know as `/usr/site/lib/libdrivers.a` by examining previously collected coverage data where the library was not excluded, and use that full name:

```
exclude /nfs/u46/site/releases/3.0/lib/libdrivers.a
```

or use wildcards:

```
exclude */lib/libdrivers*
```

You can also omit the entire leading pathname and exclude object files and libraries by unqualified names (containing no "`/`"), such as:

```
exclude libdrivers*
```

Excluding a file or library in your program does *not* remove previously collected data from any `.pcv` files. No *new* data will be collected for the excluded files. To eliminate the old data, remove the `.pcv` files.

Excluding coverage by source filename

You can exclude files by source name, but this is not recommended, because there is no way for PureCoverage to know which object files to update when source name `exclude` directives are changed.

If you change source name `exclude` directives, you must *manually* remove the instrumented versions of all object files that may contain contributions from those source files. For example a C++ header file can easily generate code every time it is included, affecting code in hundreds of object files. Or you can rebuild your program once with the option `-force-rebuild` to force all files and libraries to be re-instrumented.

Adjusting coverage on a line-by-line basis

This section explains:

- What adjustments are
- What types of adjustments can be made
- How to apply adjustments to code manually and interactively
- How to save files with adjustments
- How to remove adjustments interactively
- Adjustment file format
- Strategies for using adjustments

Adjustments

Actual coverage data is the basis for PureCoverage's statistical and annotated source code displays. The displays, by default, reflect data for all code that PureCoverage reviews.

At times, however, the displayed information is more useful if you adjust the scope of coverage to exclude specific lines. For example, if your program contains code that is logically unreachable, or extremely difficult to reach, PureCoverage highlights the

unreached code as untested and includes it in the statistics as unused code. For practical purposes, this can be misleading.

PureCoverage lets you mark source code that is difficult or impossible to reach. This keeps it from being displayed as untested and counted as unused code.

Types of adjustments

You can mark lines with three types of adjustments:

- *Deadcode* adjustments indicate that the lines are impossible to reach, even though the compiler has generated code for them.
- *Tested* adjustments indicate that someone has been able to reach the lines, and that they worked correctly.
- *Inspected* adjustments indicate that someone has examined the lines of code and concluded that they are correct.

All three types of adjustments apply to *lines*, so you must compile with the `-g` debugging option if you want to use the adjustment feature.

You can mark adjustments in two ways:

- Manually, using your text editor.
- Interactively, using the Annotated Source window. (See page 4-10.)

Marking adjustments manually

To mark adjustments manually, embed adjustment directives *as comments* in your source code as you edit it. You can subsequently tell PureCoverage to *extract* the directives and *apply* them in calculating coverage data.

Two styles of comments are available: *single-line comments*, each of which affects one line of code, and *block comments*, which indicate the beginning and end of an adjusted block of lines. Each

comment consists of the keyword purecov followed by a colon (:), and then the type of adjustment. The complete set of comments is:

Comment	Meaning
/* purecov: deadcode */ /* purecov: begin deadcode */ /* purecov: end */	The indicated lines are not reachable, even in theory.
/* purecov: tested */ /* purecov: begin tested */ /* purecov: end */	The indicated lines have been tested by someone, at some point, and should be considered as working.
/* purecov: inspected */ /* purecov: begin inspected */ /* purecov: end */	The indicated lines have been through some kind of code inspection or review, and are believed to be correct.

For example, the following line is marked as deadcode:

```
printf("bar!\n"); /* purecov: deadcode */
```

If you use begin and end pairs, the adjustment applies to the lines containing the comment, as well as all lines in between.

The begin and end pairs *do not nest*, regardless of whether they are the same type of adjustment. It is not necessarily an error for a begin to exist without a matching end: Everything in the file after the begin is adjusted.[2] However, a warning that there is no corresponding end appears on the output.

Directives are recognized wherever they occur, even if they are not in comments.

2. When there is no explicit end, PureCoverage uses the line of the EOF as the missing end line. Note that if the file grows, this adjustment still extends only to the old EOF unless you extract the adjustments from the file again.

If there is no code on an adjusted line, the adjustment has no effect on the coverage data, so there is no danger in something like this:

```
/* $Log: myfile.c,v $
 * Revision 2.54 1995/05/16 12:32:02 frodo
 * added "purecov: deadcode" directives
 */
```

Be aware, however, that PureCoverage also suppresses a line like this:

```
printf("/* purecov: deadcode */\n");
```

Extracting adjustments from source files

PureCoverage supports the `-extract` option for use on source files:

```
purecov -extract <sourcefile> [<sourcefile....>]
```

This option extracts all adjustments from the source files and stores them in `~/.purecov.adjust`.

Note: PureCoverage extracts adjustments from an individual file automatically when you open the Annotated Source window for that file. To extract adjustments from all files referenced in the PureCoverage Viewer, you can use the Viewer's Adjustments menu. Or use the command `purecov -extract *.c` before opening the Viewer.

You can use the `-extract` option with two or more files, for example:

```
purecov -extract foo.c bar.c
```

The adjustment data for both files is stored in your `~/.purecov.adjust` file. If you run the extract option on the same file twice, the newer version of the `~/.purecov.adjust` file replaces the older version.

The adjustments should be extracted each time the source file is modified. You can do this automatically by adding the `-extract` option in the makefile as part of the `cc -c` step.

Whenever PureCoverage extracts adjustments, it outputs a short summary of the extraction process. For example:

```
% purecov -extract bim.c
Updating adjustments for /canonical/path/to/bim.c:
      23 lines of dead code
      42 manually tested lines
```

Viewing adjusted displays

When you open the PureCoverage Viewer, PureCoverage applies the adjustments in `~/.purecov.adjust` to the display. To make sure the adjustments are current, select **Extract adjustments from all source files** from the Viewer's Adjustments menu.

Note: The adjustments are always current if you include the `-extract` option in your makefile.

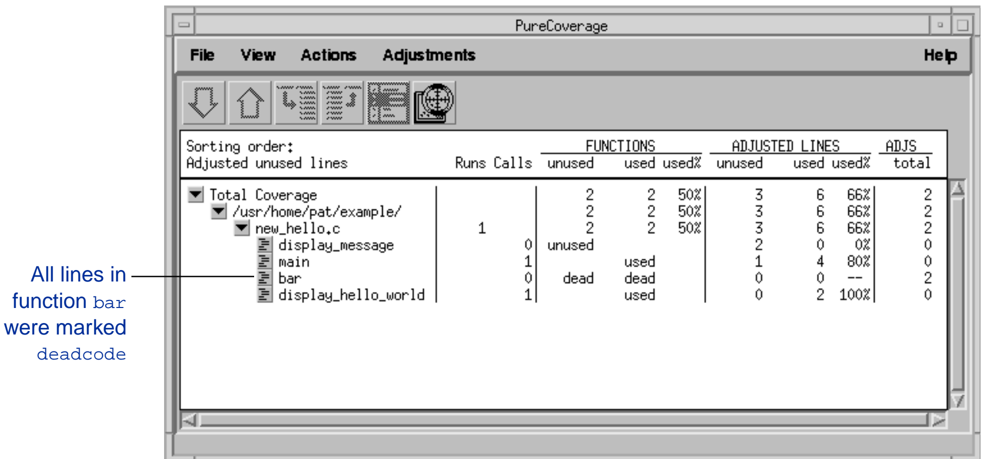
The adjusted Viewer display

The three adjustment types have different effects on the `ADJUSTED LINES` information in the Viewer display:

- Marking a line `tested` moves it from the `unused` column to the `used` column, with a corresponding change in the `used%` statistics.
- Marking a line `inspected` has the same effect on the Viewer display as marking it `tested`.
- Marking a line `deadcode` removes it entirely from the `ADJUSTED LINES` statistics, as if it did not exist. The number in the `unused` column is decremented by 1, with a corresponding change in the `used%` column but *no change* in the `used` column.

You can display additional columns in the Viewer to see more information about adjustments. Use the **Select columns** item from the View menu. See “Using the Viewer View menu” on page 5-7.

If you mark *all lines in a function* `deadcode`, PureCoverage still counts the function as an unused function, but treats it as if it had no lines in it. The `unused` and `used` columns for `FUNCTIONS` list it as `dead`.



If you mark *all lines in a file* `deadcode`, the file still appears in the Viewer statistics, but PureCoverage treats the file as if it contained only functions with no lines in them.

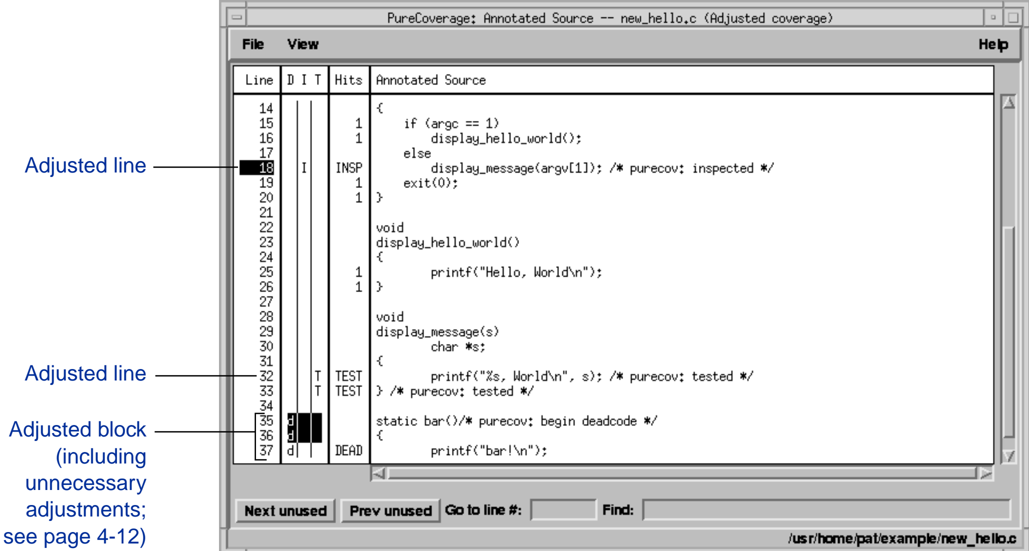
The adjusted Annotated Source window display

The different types of adjustments are marked `DEAD`, `TEST`, or `INSP` in the `Hits` column of the Annotated Source window.

Likewise, each adjusted line is marked `D`, `I`, or `T` in the adjustments columns:

- `D` for `deadcode`
- `I` for `inspected`
- `T` for `tested`

A lowercase `d`, `i`, or `t` indicates an adjustment to a *block* of lines.

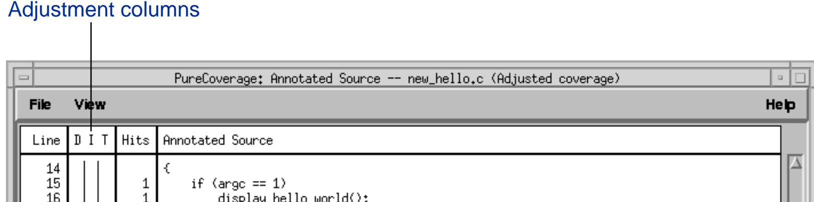


You can turn on highlighting for any type of adjustment through choices in the Annotated Source window View menu. In addition, you can change the highlight colors for the different types by customizing your `~/.purecov.Xdefaults` file.

Marking adjustments interactively

In addition to making adjustments manually in your source code, you can make adjustments interactively in the Annotated Source window.

To adjust code in the Annotated Source window, click one of the three adjustment columns (`D`, `I`, or `T`) next to the line you want to adjust. Note that the cursor changes to a mouse icon when you move it over the adjustment columns.



PureCoverage appends adjustment directives as comments to the right of code lines, for example: `/* purecov: inspected */`. An adjustment directive automatically replaces any previous adjustment to the line.

Each adjustment is effective *immediately*. PureCoverage updates both the Annotated Source window and the statistics in the Viewer. It does not re-sort the Viewer display, however, but only changes the `Sorting order` to `Unsorted`:

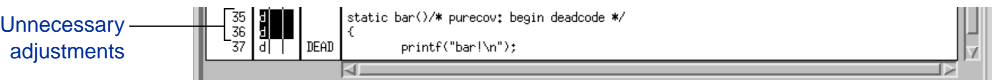
To re-sort the display, select **Select sorting order** from the View menu in the PureCoverage Viewer.

Note: You cannot use this interactive method to mark FORTRAN dialects that require all comments to exist on a separate line. You can, however, change the comment *style* (to C++ `//`-style comments, for example) by editing your `~/.purecov/.Xdefaults` file.

Unnecessary adjustments

Adjustments are appropriate only for lines that meet two conditions: They must contain code, and they must be unused. If you mark an adjustment for a line that does not meet these conditions, PureCoverage *highlights the adjustment columns* to indicate that the adjustment is unnecessary. Other than the highlight, marking these lines has no effect.

Using `begin` and `end` comment pairs to mark multi-line statements as blocks (see page 4-5) frequently results in unnecessary adjustments. Marking multi-line statements interactively, through the Annotated Source window, allows you to avoid this.



To mark multi-line statements manually with individual comments, mark the line that your compiler would annotate if the line were used. Different compilers annotate multi-line statements differently. See “Multi-line statements” on page C-2 for examples.

Saving files with adjustments

To rewrite the current source file, including all adjustments you added interactively through the Annotated Source window, and the `~/.purecov.adjust` adjustments file, select **Save** from the Annotated Source window File menu.

Likewise, when you close the Annotated Source window, PureCoverage asks you whether you want to save the modifications you made. If you select **Save and close window**, PureCoverage saves both the modified source file and the modified `~/.purecov.adjust` file.

Note: The items **Save source and annotations as . . .** and **Snapshot** in the Annotated Source window File menu save a record of the

window *display*. They do not affect either the source file or the `~/.purecov.adjust` file.

Removing adjustments in the Annotated Source window

Remove an adjustment by clicking the `D`, `I`, or `T` to the left of the line.

You can also remove an adjustment by selecting a line and pressing the space bar. To select a line, click on the number; use the arrow keys or the j and k keys; or enter a line number in the **Go to line #** field at the lower edge of the window. The selection point shows which line is currently selected.

Selection point

```
15          1      if (argc == 1)
16          1          display_hello_world();
17                 else
18    I  INSP          display_message(argv[1]); /* purecov: inspected */
19          1      exit(0);
20          1  }
```

PureCoverage deletes the entire comment, but only if the comment's format is *exactly identical* to the format that the Annotated Source window uses for interactive adjustments; see page 4-6. Otherwise, PureCoverage inserts the word `NOT`.

For example, PureCoverage cannot delete the following manual adjustment because it does not include a space between the colon and the word `deadcode`:

```
/* purecov:deadcode */
```

Instead, PureCoverage modifies it as follows:

```
/* purecov:NOT deadcode */
```

Note: For information on how export mode handles adjustments, refer to "The effect of coverage adjustments on export" on page B-9.

Adjustment file format

Refer to this description of the adjustment file format when you are writing scripts that process adjustments.

Most users process adjustments with `purecov -extract`, or through the Viewer and Annotated Source window. Direct editing of PureCoverage adjustments, which are in ASCII format, is also possible. For example, you can use a script to apply adjustments to export data produced by an old version of PureCoverage.

If you have already annotated your code with markers that another tool understands, you can process your code into a format that PureCoverage can read without editing all your source files. For example, you can convert `lint`-style `NOTREACHED` comments into PureCoverage deadcode adjustments.

Format description

All adjustment files must start with this line, exactly as shown:

```
PureCoverage Adjustments File 1.0
```

The line must not contain any extra spaces or tabs. It must be the first line in the file, and cannot have a comment at the end.

Each additional line in the file can be arbitrarily wide. Lines beginning with a `#` in the leftmost column are comments. The format of each non-comment line is:

```
<name> <tab> <time> <tab> <type> <tab> <style> <tab> <list of adjustments>
```

where:

- `<name>` is the canonical fullpath to the file containing adjustments. This is exactly the form of the name that appears in the Viewer.
- `<time>` is the update time of the adjustments for that file as an `int` (UNIX `time_t`).
- `<type>` is `deadcode`, `tested`, or `inspected`.

- `<style>` is either an uppercase `L`, indicating that the adjustment came from a single-line adjustment, or an uppercase `B`, indicating that the adjustment came from a block-style adjustment.
- `<list of adjustments>` is a list of single line numbers or ranges of lines, each separated by a space; for example, `4 10 15-23`.

If there are multiple entries for a single file, PureCoverage uses all entries with times matching the most recent time given for that file; it discards the others. It also discards incorrectly formatted lines (or sometimes only portions of them). Depending on how you are reading the file, PureCoverage displays errors either in a dialog or on `stderr`.

Strategies for using PureCoverage adjustments

The PureCoverage adjustment feature requires you to adopt an appropriate strategy for adapting it to your own situation. This section raises considerations that can influence your use of the feature, and suggests two possible usage models.

Adjustment usage considerations

To use the PureCoverage adjustments feature in the most efficient and productive way, keep the following considerations in mind as you develop your strategy.

Can you use adjusted coverage?

Does your development and testing methodology permit the use of adjusted coverage?

If not, there is no requirement to use the adjustment feature. PureCoverage still provides you with the actual coverage data.

Note: The `Adjusted Lines` column header appears in the Viewer by default, even if you do not use the adjustment feature. To change the header, select the **Select columns . . .** item in the Viewer's View menu. Then select columns from the **Actual Line Coverage**

category instead of the **Adjusted Line Coverage** category. Also, use **Select sorting order** in the View menu to reflect actual coverage.

Who can make adjustments?

Who is permitted to insert adjustments into the code? What rules must they follow?

Many answers are possible. Here is one:

Allow any developer to insert any type of adjustment into the code. For `tested` and `deadcode` adjustments, the developer must also add a comment nearby, indicating who made the adjustment, when the adjustment was applied, and why the code is dead or how it was tested. The exact placement of the comments depends on your compiler. For example:

```
void ErrExit(char * message)
{
    fflush(stdout);
    fprintf(stderr, "\nError: %s\n", message);
    fflush(stderr);
    exit(1);    /* pat 12/11/95: exit is a non-returning */
                /* function, so this 'return' line is   */
}               /* unreachable.  purecov: deadcode */

void *SafeMalloc(int num_bytes)
{
    int real_num_bytes = num_bytes < 1? 1 : num_bytes;
    char *ptr          = (char *)malloc(real_num_bytes);
    if (ptr == NULL) {              /* pat 12/11/95: I used */
                                    /* dbx to force ptr */
                                    /* null; this worked. */
        ErrExit("out of memory");   /* purecov: tested   */
    } else {
        memset(ptr, 0, real_num_bytes);
    }
    return(ptr);
}
```

`Inspected` adjustments also require a comment nearby including the developer's name, the date, and the name of another person who has also examined the code and agrees with the placement of the `inspected` adjustment.

However, many groups that follow informal development methodologies find this approach too strict. The main point is that everyone in the group must mean the same thing when they mark a line of code with an adjustment.

When are adjustments applied?

When are adjustments applied to the data, and how is the database of adjustments maintained?

This question has many practical day-to-day implications. Its answer depends on the model your development group follows, as discussed in the next section.

Models for using PureCoverage adjustments

This section describes two models for using the PureCoverage adjustments feature.

Full-usage model

The model that makes the most thorough utilization of the adjustment features is one in which adjustments are always applied by all the developers. This implies complete integration of adjustments into the daily build system.

Under this model, you can modify the `cc` rule in your makefiles for compiling source files to `.o` files. For example, instead of:

```
.c.o:
    $(COMPILE.c) -o $% $<
```

you can use:

```
.c.o:
    $(COMPILE.c) -o $% $<
    purecov -extract $<
```

The additional step says that whenever a C file has been changed, not only must the file be recompiled, but the updated coverage adjustments must also be extracted from it.

Any time the PureCoverage Viewer is started, it loads the adjustments from ~/.purecov.adjust, so coverage for the program is always adjusted.

The advantages of this method are that adjustments are always up-to-date, and that no effort other than the initial makefile setup is required.

The disadvantages are that the makefiles must be modified globally. Everyone has to agree to use this approach, which is not possible if some groups are not using PureCoverage, or if some developers do not have a license. It also requires a small amount of extra time for extracting adjustments during every compilation.

Single-user model

This is a good way to operate if an individual uses PureCoverage with adjustments in a group that, for any reason, does not want to include PureCoverage in its makefiles.

Under this model, when you are examining coverage data in the PureCoverage Viewer, select **Extract adjustments from all source files** from the Adjustments menu. This applies all current adjustments to the display.

You can then add appropriate adjustments to the source files interactively as your analysis proceeds. You can save the adjusted source file and the ~/.purecov.adjust file by selecting **Save** from the File menu in the Annotated Source window.

The main difference between this model and the full-usage model is that the user has to maintain the adjustments manually, since the makefiles do not take care of it automatically.

Note: If you use scripts instead of the Viewer to analyze coverage data, you can use the command:

```
purecov -extract *.c
```

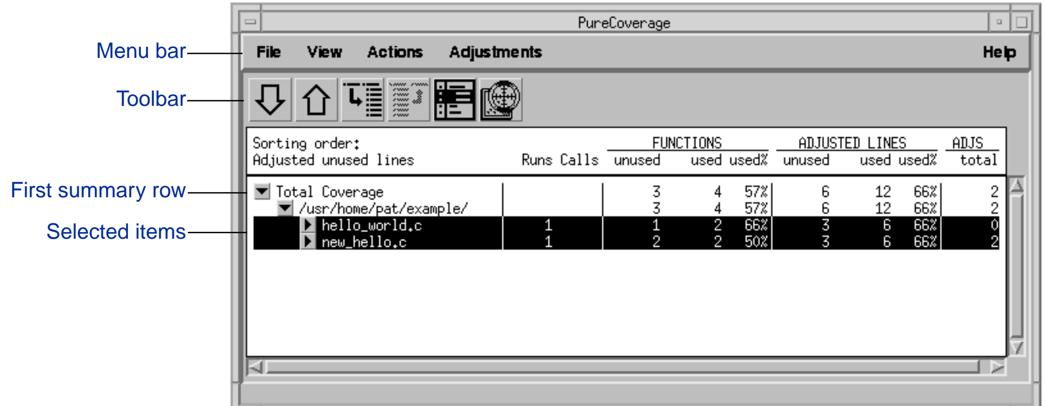
to extract the adjustments from your source files.

Using the PureCoverage Graphical Display

This chapter gives you a quick tour through features of the PureCoverage graphical display that were not introduced in Chapter 2 or Chapter 4.

PureCoverage Viewer

The PureCoverage Viewer displays coverage statistics for your application.



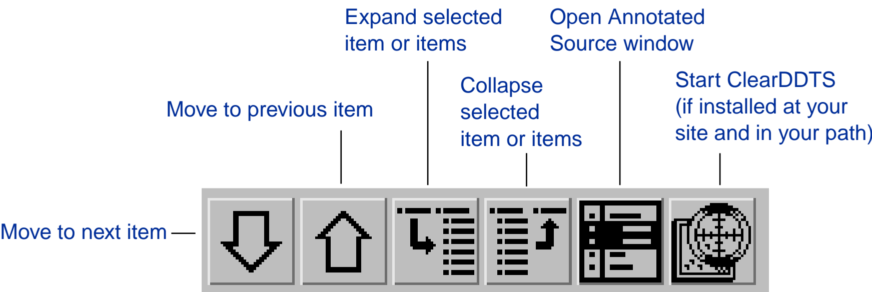
Selecting items

Operations in the Viewer apply to the currently selected item or set of items. The Viewer displays these items in inverse video.

To select an item, click left on it. To select a *set of items*, hold down the left mouse button and sweep the items you want to select.

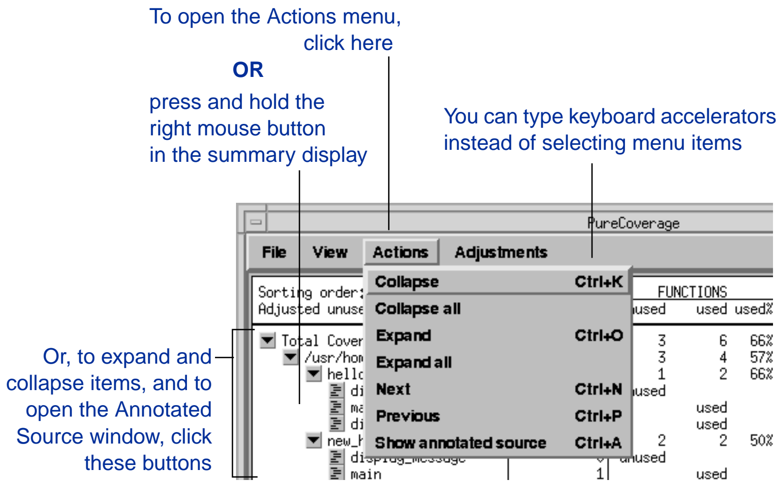
Using the toolbar

Click toolbar buttons to perform the following actions:

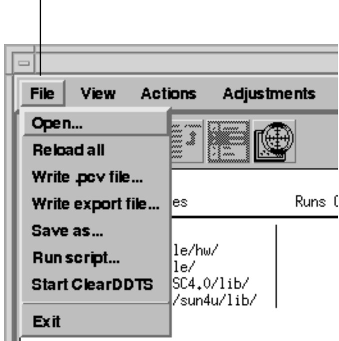


If you prefer, you can hide the toolbar by selecting **Hide toolbar** from the View menu, and perform all these operations in other ways. For example:

- **Next**, **Previous**, **Expand**, **Collapse**, and **Show annotated source** are available in the Actions menu.



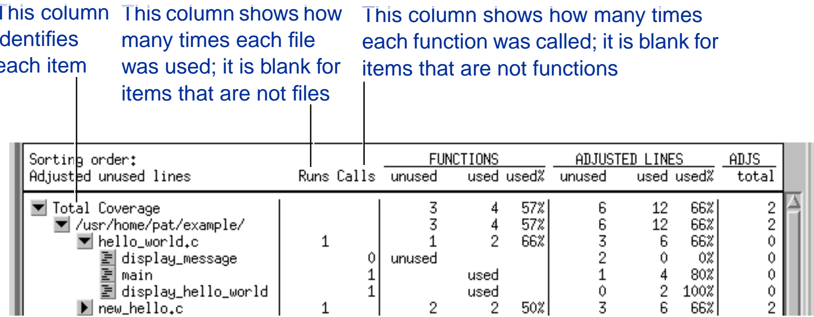
- **Start ClearDDTS** is available in the File menu.



Note: To display the toolbar again, select **Make toolbar visible** from the View menu.

Selecting Viewer columns

By default, the Viewer displays the following information:



Sorting order: Adjusted unused lines	Runs	Calls	FUNCTIONS unused	used	used%	ADJUSTED LINES unused	used	used%	ADJS total
Total Coverage			3	4	57%	6	12	66%	2
/usr/home/pat/example/			3	4	57%	6	12	66%	2
hello_world.c	1		1	2	66%	3	6	66%	0
display_message		0	unused			2	0	0%	0
main		1		used		1	4	80%	0
display_hello_world		1		used		0	2	100%	0
new_hello.c	1		2	2	50%	3	6	66%	2

Note: Each time you run a program containing a file, the `Runs` count for the file increases by one, even if the run did not use any code in the file.

Functions

The columns labeled FUNCTIONS give information for each file and for each function, as well as Total Coverage information for your entire application:

- For the entire application and for each file, the values show how many functions were used or unused, and what percentage of the total number of functions was used.
- For each function, a word in the appropriate column indicates whether the function was used or unused.

Meaningless percentages are displayed as "--" instead of as a number.

Adjusted Lines

The columns labeled ADJUSTED LINES indicate how many lines in the entire application, in each file, and in each function were unused and used, as well as what percentage of lines was used.

The heading ADJUSTED LINES does not necessarily imply that adjustments are present, but only that any current adjustments are included in the statistics. The ADJS column indicates whether adjustments are present, giving the total number of adjustments applied for the entire application, for each file, and for each function.

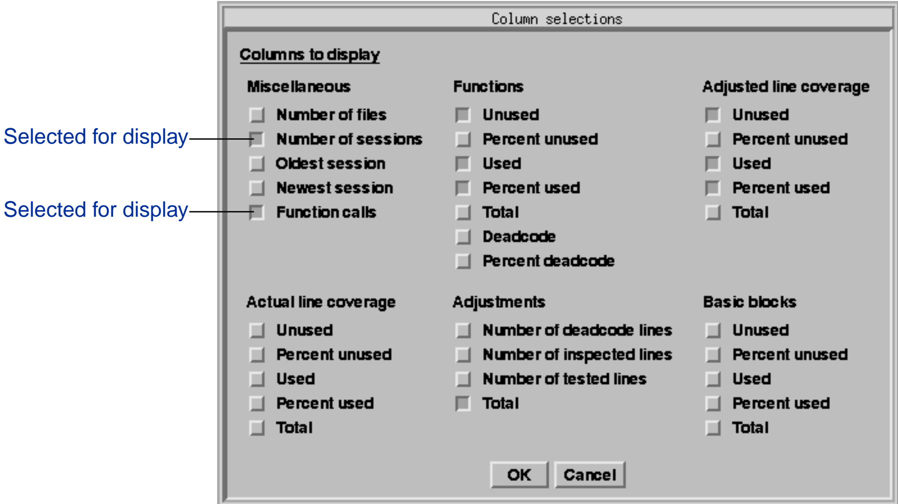
Note: The ADJS totals do not include unnecessary adjustments; see page 4-12.

For complete information about adjustments, see "Adjusting coverage on a line-by-line basis" on page 4-4.

Optional columns

The Viewer can display many types of columns in addition to the defaults. To customize the display, select **Select Columns** from the View menu, then select the columns you want to see.

To select a column for display, click the column description. A punched-down selection button shows that a column is selected.



- The `Miscellaneous` settings include:
  - `Number of files`, which adds a column indicating the number of files in each directory and in the entire Viewer. This column is blank for individual files and functions.
  - `Number of sessions`, which controls the `Runs` column, showing the number of runs for which data has been collected.
  - `Oldest session` and `Newest session`, which add columns showing the date and time of the oldest and newest run for which data has been collected for a given file or directory.
  - `Function calls`, which controls the `Calls` column, showing the number of times each function has been called.
- For lines, you can display `Actual Line Coverage` and `Adjusted Line Coverage`. `Actual Line Coverage` ignores adjustments. For `Basic blocks`, you can display only actual coverage.

  In each category, you can display percentages and counts for used and unused code. Selecting `Total` displays the total number of used and unused lines or blocks.

- The `Function` adjustments are similar to those for lines, but include additional options for displaying the number and percentage of `deadcode` adjustments. If all lines in a function are marked `deadcode`, the entire function is marked `dead` in the `used` and `unused` columns.
- The `Adjustments` columns provide line-based adjustment information for each item in the Viewer. The figures represent the number of lines adjusted, not the number of source lines marked. (To qualify as an adjustment, a line must both contain code and be unused.)

Using the Viewer File menu

The File menu allows you to control the Viewer's operation. This includes reading and writing Viewer data files.

- **Open** opens a dialog allowing you to select a `.pcv` file for merging into the current Viewer. The new coverage data from the file is merged with the data currently loaded in the Viewer.
- **Reload all** allows you to reload the data for all the `.pcv` files currently displayed in the Viewer. PureCoverage discards the data from the files already being viewed and loads updated coverage data.

  This option is useful if you instruct PureCoverage not to reload changed `.pcv` files, and not to inform you of future changes when you are running tests while the Viewer is open (see page 2-9).
- **Write .pcv file. . .** opens a dialog box allowing you to save the current coverage data in a new `.pcv` file for later analysis.
- **Write export file. . .** opens a dialog box allowing you to save the current merged coverage data in export format in an ASCII text file.
- **Save as. . .** opens a dialog box allowing you to save the current contents of the Viewer display to an ASCII text file.
- **Run script. . .** opens a dialog box allowing you to run one of PureCoverage's report scripts or a custom script. See page 3-10.

- **Start ClearDDTS** starts the ClearDDTS defect-tracking tool, if it is installed on your system.
- **Exit** exits the PureCoverage program and closes all PureCoverage windows.

Using the Viewer View menu

The View menu allows you to control the Viewer display.

- **Select columns. . .** allows you to display different types of data in the Viewer. See “Optional columns” on page 5-4.
- **Select sorting order** opens a pull-right menu allowing you to control how items in the Viewer are sorted. The Viewer identifies the current sorting order in the upper-left corner of the summary display.
- **Set display style for names. . .** opens the `Name style settings` dialog, allowing you to specify how to display directory names, pathnames, filenames and function names in the Viewer. For C++ applications, you can choose to suppress class names, argument lists, or operator prefixes.

  You can also specify a minimum width for the names column, which the Viewer maintains even when you make the Viewer narrower.
- **Vertical column separators** opens a pull-right menu allowing you to control how columns in the Viewer are separated.
- **Hide toolbar/Make toolbar visible** allows you to turn the toolbar display off and on.
- **Refresh display** refreshes all open PureCoverage windows.

- **Update ~/.purecov.Xdefaults** modifies your `~/.purecov.Xdefaults` file to reflect the current PureCoverage display.[1]
- **Write view settings to . . .** allows you to record the current graphical display settings in a file of your own choosing. Comments are not included.

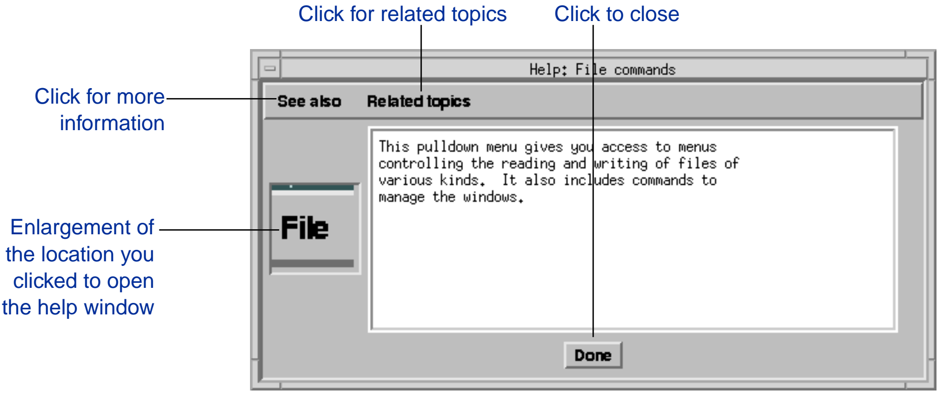
Using the Viewer Adjustments menu

This menu contains one item, **Extract adjustments from all source files.** Select it to extract adjustments from every source file named in the Viewer, and to apply the adjustments to the Viewer display.

Using the Viewer Help menu

The Help menu gives you access to information that can be useful while you are working in the PureCoverage graphical display:

- **On Context** turns the cursor into a question mark (?). Click the question mark on the area of your PureCoverage display that you want information about. The Help Outline dialog appears:



1. You can also modify your `~/.purecov.Xdefaults` file manually using your editor. The defaults apply the next time you start PureCoverage. The file uses typical `.Xdefaults` file format. For more information about editing an `.Xdefaults` file, see your *X Window System Users Guide*.

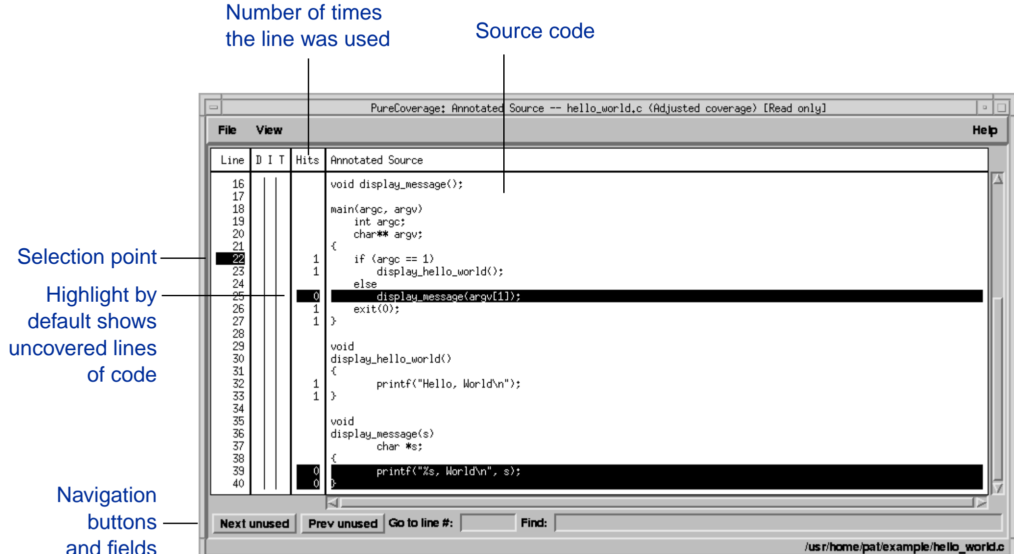
The remaining items return specific information:

- **On version** reports on your version of PureCoverage.
- **On license** displays a copy of the Simple License Agreement.
- **PureCoverage overview** gives you a quick introduction to the PureCoverage graphical display.
- **Release notes** displays the current `README` file, containing supplementary PureCoverage information.
- **Bug report** tells you how to file a bug report.

Annotated Source window

The Annotated Source window displays a selected source file with line-by-line coverage data. You can use this window to make coverage adjustments to the code, but not to change the code itself. See “Adjusting coverage on a line-by-line basis” on page 4-4.

For the adjustment information displayed in this window, see “Viewing adjusted displays” on page 4-8.



Using the Annotated Source File menu

The File menu allows you to control operations in the Annotated Source window.

- **Save** rewrites both the current source code file, including all adjustments displayed in the Annotated Source window, and the `~/.purecov.adjust` file.
- **Save source and annotations as . . .** opens a dialog box allowing you to save the current display, including line numbers and hit counts, to an ASCII text file. This file cannot be compiled. In content and format, it is equivalent to the report that the `pc_annotate` script produces.
- **Snapshot** allows you to take a read-only snapshot of the current Annotated Source window.

  The Annotated Source window changes when you load new coverage information. **Snapshot** allows you to compare line-by-line coverage data while you test a file.

  PureCoverage discards snapshots when you exit the program.
- **Close** closes the Annotated Source window, but does not exit PureCoverage.
- **Exit PureCoverage** exits the PureCoverage program and closes all PureCoverage windows.

Using the Annotated Source View menu

The View menu allows you to customize the display in the Annotated Source window. By default, the window displays line numbers and line counts, highlighting unused lines that have not been adjusted.

You can turn the following items on or off:

- **Show adjusted annotated source** allows you to choose between highlighting based on adjusted coverage and highlighting based on actual coverage.

- **Show line counts** displays the `Hits` column, which shows how many times each line of code has been executed.
- **Show line numbers** displays a column showing line numbers for each line of code.
- **Highlight unused lines** displays all lines of untested code in inverse video.
- **Highlight used lines** displays all lines of tested code in inverse video.

The following menu items allow you to turn on highlighting for the different types of adjustments:

- **Highlight 'purecov: deadcode' lines**
- **Highlight 'purecov: inspected' lines**
- **Highlight 'purecov: tested' lines**

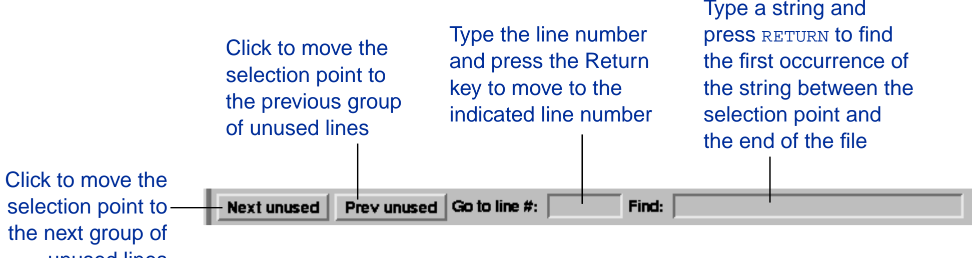
  You can assign colors to distinguish the types of adjustments by modifying your `~/.purecov.Xdefaults` file.

Using the Annotated Source Help menu

The Help menu items are the same as those in the Viewer. See "Using the Viewer Help menu" on page 5-8.

Navigating in the Annotated Source window

The navigation buttons and text fields are located along the bottom edge of the Annotated Source window:



To find each occurrence of a string in a file, begin searching at line #1. Press the Return key after each match. PureCoverage informs you when it finds no more matches.

The search is case-sensitive, and recognizes the wildcards `?` (a match for a single character) and `*` (a match for a series of zero or more characters).

Report Scripts

Report scripts allow you to format and process the data that PureCoverage has generated. PureCoverage includes a comprehensive set of report scripts that you can run as-is, modify for your environment, or use as a starting-point for writing your own custom report scripts.

PureCoverage report scripts

PureCoverage includes the following report scripts:

Report name	Script	Description	Page
Coverage summary report	`pc_summary`	Produces an overall summary	6-2
Low coverage report	`pc_below`	Reports low coverage	6-3
Low coverage mail report	`pc_email`	Mails a report to the last user who modified insufficiently covered files	6-3
Spreadsheet report	`pc_ssheet`	Produces a summary in spreadsheet format	6-4
Differences report	`pc_diff`	Lists files for which coverage has changed	6-5
Build differences summary report	`pc_build_diff`	Compares PureCoverage data from two builds of an application	6-6
Annotated source report	`pc_annotate`	Produces an annotated source text file	6-8
Annotated differences report	`pc_covdiff`	Annotates the output of `diff` for modified source code	6-9
Selected tests report	`pc_select`	Identifies the subset of tests required to exercise modified source code	6-11

The scripts are located in the `<purecovhome>/scripts` directory. You can add the directory to your path, or install links to the scripts from a location such as `/usr/local/bin`. See the *Installation & Licensing Guide* for details. The scripts are written in Perl, a version of which is included with PureCoverage in the `<purecovhome>/PERL` directory.

All scripts accept the `-force-merge` and `-apply-adjustments` options. For information about these and other options, see Chapter 7.

For information about writing custom report scripts, see page 6-15.

Coverage summary report

This report is an overall summary similar in format to the data the PureCoverage Viewer displays. The syntax is:

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

The `<prog>.pcv` files are the PureCoverage data files for the programs to be summarized.

The `-file=<name>` option controls which files appear in the summary. PureCoverage includes only the files you specify. When you do not specify a file, PureCoverage includes all source files referenced by the `.pcv` files.

Note: If you do not specify a `<prog>.pcv` filename, the script accepts export-format data on `stdin`.

An overall summary report is printed showing the coverage information for all the files listed on the command line.

```
% pc_summary a.out.pcv

PureCoverage Summary Report Thu Apr 11 12:50:18 EDT 1996  Page 1

File: /tmp/hello/hello_world.c
                         Funcs   Funcs  Times  Lines Lines  Lines
Function Name             Used   Used% Called Unused  Used  Used%
==============================  ====== ====== ====== =====  =====
** hello_world.c total **    3    100%             0    12   100%
display_hello_world                         1      0     3   100%
display_message                             1      0     3   100%
main                                        2      0     6   100%
```

Low coverage report

This report lists files where the percentage of line coverage is lower than a specified minimum. The syntax is:

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

This report lists only files with line coverage below the specified percentage. If you do not specify `<pct>`, PureCoverage uses 80% as the threshold.

The `<prog>.pcv` files listed are the PureCoverage data files for the programs to be reported.

```
% pc_below -percent=75 myprog.pcv
60    /usr/pat/work/message/utils.c
56    /usr/pat/work/database/utils.c
```

Low coverage mail report

This report is an extension of the low coverage report. It takes the low coverage report output, examines the RCS data for each file listed, and determines who most recently modified the files. The responsible developers are sent e-mail informing them that the coverage on the file is too low. The syntax is:

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

Mail is sent only for files that have a percentage of used lines below `<pct>`. If you do not specify `<pct>`, PureCoverage uses 80% as the threshold. If no files are below the requested percentage, no mail is sent.

The `<prog>.pcv` files that you list are the PureCoverage data files for the programs to be reported.

Developers receive mail listing all appropriate files:

```
% pc_email -percent=75 myprog.pcv
Date: Wed, 1 May 1995 07:45:34 -0800
From: terry@pure.com  (Terry Programmer)
To: pat@pure.com
Subject: Coverage too low
/usr/pat/work/message/utils.c is below 75% used lines
/usr/pat/work/database/utils.c is below 75% used lines
```

Note: The PureCoverage e-mail script supports only the RCS source control system. You can customize the script to support your own source code control system.

Spreadsheet report

This report is a file which can be read into a spreadsheet program for further analysis. The syntax is:

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

The `<prog>.pcv` files that you list are the PureCoverage data files for the programs to be reported.

The report format is tab-separated text. For each function, the report includes the directory name, file name, and function name. The numbers in the two columns on the right indicate the percentage of uncovered lines and the total calls to the function.

```
% pc_ssheet prog.pcv
/usr/pat/work/message/   utils.c   unused_function         100  0
/usr/pat/work/message/   utils.c   heavily_used_function   0    1000
/usr/pat/work/message/   utils.c   somewhat_used_function  0    25
/usr/pat/work/message/   shifty.c  empty_line              0    100
/usr/pat/work/message/   shifty.c  empty_line2             0    200
/usr/pat/work/database/  utils.c   unused_function         100  0
/usr/pat/work/database/  utils.c   heavily_used_function   0    10000
/usr/pat/work/database/  utils.c   somewhat_used_function  0    25
```

Note: When the number of total calls reaches 10,000, as for the second function from the last, the counter becomes *saturated*; the value stops incrementing.

Differences report

This report is useful for analyzing changes in coverage, typically because some change to the program or test suite was made between the time the old data was collected and the time the new data was collected. The syntax is:

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

You must specify two filenames when you use the `pc_diff` script. One of the filenames, however, can be "`-`"; this designates `stdin` as the source of exported information:

```
% purecov -export new.pcv | pc_diff old.pcv -
```

or

```
% purecov -export old.pcv | pc_diff - new.pcv
```

Note: When you run the script from the PureCoverage Viewer, you must enter "`-`" as one of the filenames.

The `old` and `new` files are the two PureCoverage data files to be compared. The resulting report shows the files for which coverage got worse or improved, and files which were added or deleted.

```
% pc_diff old.pcv new.pcv
Files which got worse:

Files which improved:
      /usr/sam/work/hash/ hash.c 71% changed to 96%
      /usr/sam/work/tests/ testHash.c 52% changed to 70%

Files which are new to new.pcv.
      /usr/chris/work/graphics/ view.c 100%
      /usr/chris/work/graphics/ menu.c 100%
      /usr/chris/work/graphics/ dialog.c 100%

Files from old.pcv which no longer exist:
      /usr/pat/work/database/ utils.c 56%
      /usr/pat/work/message/ shifty.c 100%
      /usr/pat/work/message/ utils.c 60%
      /usr/pat/work/message/ xmit.c 100%
```

Build differences summary report

This report shows how coverage has changed between builds of an application. The syntax is:

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

The script exports each `.pcv` file, takes a `diff` of the resulting data, and prints a summary of the changes. The differences are calculated in terms of lines of code, so the script works only if the `-g` option was used to compile.

Any prefixes are stripped from the left side of directory names so that data from nightly build areas can be compared even though the root directory of the builds varies from day to day. You can specify multiple `-prefix` arguments, but one prefix at most will be applied to each directory name. The prefixes are applied in left-to-right order.

Note: The prefix is a Perl regular expression anchored to the left by a caret (^).

As an example for `pc_build_diff`, suppose you do nightly builds or tests in directories named using the scheme `/usr/home/builds/daily.<date>`, and that one program you

produce as part of this build is called myprog and is built in the myprog subdirectory of the build tree. Suppose further that there is some shared code used by this and other programs. So, in your filesystem, you have:

```
/usr/home/builds/daily.960408/myprog/main.c
/usr/home/builds/daily.960408/myprog/apply_patch.c
/usr/home/builds/daily.960408/myprog/myprog.pure.pcv
/usr/home/builds/daily.960408/shared/strings.c
```

as well as other source files. You also have the corresponding trees for other days on which you have done builds.

To compare coverage between the April 8 and April 9 builds, go to the /usr/home/builds directory, and run:

```
% pc_build_diff -prefix=".*daily......./" \
daily.96040{8,9}/myprog/myprog.pure.pcv
```

This generates the following report:

```
*** Reduced coverage ***    Unused  Change   |  Used%   Change
    shared/ strings.c           61     +10   |    75%       -3

*** Improved coverage ***   Unused  Change   |  Used%   Change
    myprog/ main.c             255      -3   |    70%        0
            apply_patch.c      103      -1   |    75%        0

*** New files ***           Unused   Used%

*** No longer tested ***
```

In the command, which is written here in csh {} syntax, the prefix .*daily......./ strips out the /usr/home/builds/daily.<date>/ part of the filenames. (The series of dots in this Perl expression matches the 7 characters of date specifications such as .960408.) The two .pcv files are then specified. The Unused and Change columns are measured in lines, so 10 fewer lines of strings.c were tested in the Apr 9 build. This works out to a decrease of 3% of the used lines. In main.c, 3 additional lines were used (3 fewer unused), but this is too small a number to influence the %used figure.

Note: You must specify two filenames when using the `pc_build_diff` script. However, one of the filenames can be "-" to denote that exported information is to be supplied via `stdin`.

Annotated source report

This report produces an annotated source listing for specified files. The syntax is:

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

By default, PureCoverage produces the annotated source report at line granularity. You can specify `-type=block` to see basic block coverage instead.

The report annotates all files that match at least one of the `<basenames>` given in the `-file` argument. If you do not specify `-file`, the report annotates all files mentioned in the specified `.pcv` files.

The script compares the `<basename>` to the *end* of file and path names. For example, the `<basename>` `prog.c` matches `1prog.c` and `2prog.c`. It also matches `path/prog.c` and `/some/path/prog.c`.

```
% pc_annotate hello.pcv
        |   main()
1       |   {
1       |       print_hello(); print_hello(); print_goodbye(); print_goodbye();
1       |   }
        |
        |   print_hello()
2       |   {
2       |       printf ("hello!\n");
2       |   }
        |
        |   print_goodbye()
2       |   {
2       |       printf ("bye bye!\n");
2       |   }
```

This example shows the report when -type=block.

```
% pc_annotate -type=block hello.pcv
        |   main()
1       |   {
1       |       print_hello(); print_hello(); print_goodbye(); print_goodbye();
1       +
1       +
1       |   }
        |
        |   print_hello()
2       |   {
2       |       printf ("hello!\n");
2       |   }
        |
        |   print_goodbye()
2       |   {
2       |       printf ("bye bye!\n");
2       |   }
```

Annotated differences report

The pc_covdiff script generates a source code differences report that includes coverage information. This allows you to see the coverage of the lines of code that you have just changed without having to consider the coverage for the unchanged portion. The syntax is:

```
% 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> <file>.pcv ...
```

The stdin for the pc_covdiff script is a description in diff(1) format of the changes made to a file specified in <name>. The value for yourdiff depends on the code management tool you are using. If you are using RCS, for example, yourdiff is rcsdiff.

Specify the same file for the -file option. This script calls the pc_annotate report script, so you can use any form of the file name that pc_annotate recognizes. List the .pcv files that contain the coverage data for the new version of the file. If you specify the -force-merge option here, the script passes it to pc_annotate.

You can use this script only on one file at a time. You cannot run it from the PureCoverage Viewer.

The other options control the format of the output. There are two output styles: `diff` and `sdiff`. In `sdiff` style, the old and new versions of the source are displayed side-by-side, with a column between them indicating whether the lines were added, deleted, or changed. In `diff` style, the old and new are listed sequentially, with code lines indicating what happened. The default is `sdiff`.

The option `-format` allows you to specify the style:

`-format=diff`	`diff` style
`-format=side-by-side`	`sdiff` style
`-format=new-only`	`sdiff` style, but without the "old" column

In both `diff` and `sdiff` styles, the tabs are stripped. Use the `-tabs` option if your text editor normally expands tabs to a value other than 8 character stops.

In both styles, you can request that additional context lines be included in the output. The default is 2, but set it to 0 if you want to produce output understandable to other tools that read `diff`.

In `sdiff` style, use the `-lines` option (default `TRUE`) to indicate whether line numbers should be included in the output.

In `sdiff` style, you can use the option `-width` to specify the width of each column of source. Normally, width is adjusted automatically to accommodate the input. If you specify an explicit value, the sources are truncated if they are too wide. Note that the width is for each column of the source, not the overall width, which includes other items such as coverage counts, the dividing column, and line numbers, depending on the options that you use.

The following sample shows the use of the `pc_covdiff` script.

```
% rcsdiff foobie.c | pc_covdiff -file=foobie.c -width=29 -lines=no foobie.pcv
===================================================================
RCS file: foobie.c,v
retrieving revision 1.3
diff -r1.3 foobie.c
  Change            Old Version                      New Version          Usage
=========== =============================   ============================= =====
                   unsigned long                    unsigned long
            {                               {                                 2
      15a16                               >     int             result =      2
                unsigned long   i;              unsigned long   i;

                if (argc > 1) {                 if (argc > 1) {               2
      19c20         for (i=0; i < num_ele |         for (i=0; (i < num_el     5
                        if (MY_STR_EQ(tab               if (MY_STR_EQ(tab     3
21,23c22,25                 printf("runni |                 printf("Proce     1
                            return((*tabl |                 printf("walki     1
                            printf("back  |                 result = (*ta     1
                                          >                 printf("back      1
                        }                               }                     3
                    }                               }
                }                               }

      28c30     return(-1);               |     return(result);               2
            }                               }                                 2

........... .............................   .............................  ....
                int i;                          int i;                        0

      48d49         printf("barsoom: %s\n <
                    printf("bim bam: %s\n           printf("bim bam: %s\n     0
      49a51                               >         printf("bang, bang! %     0
                }                               }                             0
                return(1);                      return(1);                    0
........... .............................   .............................  ....

            DispatchTable my_cmds[] = {     DispatchTable my_cmds[] = {
   54a57,58                               >     { "foo", foo },
                                          >     { "joe", foo },
                { "bar", bar }                  { "bar", bar }
            };                              };
```

Selected tests report

The `pc_select` script addresses the problem of selecting an appropriate subset of tests to run in order to test a limited change made to the code. Most of the time, it is too time consuming to run the entire test suite after each change, so the typical practice is for the user to update the master source code without running any of the test suite. Then, as part of an overnight build process, the full test suite is run and reported.

There are problems with this practice, for example:

- If the update introduces severe problems, the rest of the development group may be "stuck" while they wait for a repair.
- If several developers have made updates, it may not be clear which code changes caused things to break.
- If several developers have introduced bugs, one set of problems may mask the other, causing a series of delays between the initial update of the code and the eventual discovery of problems introduced.
- When problems are eventually discovered, the developer may already be involved in other changes, making it inconvenient or difficult to stop and fix those problems.

The `pc_select` script gives developers a way to run just that subset of the test suite which is necessary to exercise the code changes they have just made. It can greatly reduce the testing time required, making it much more feasible to do the appropriate testing before the code update.

Even though some test harnesses support the selection of an arbitrary subset of the tests to run, they still do not address the issue of determining just which subset of the tests is the appropriate subset to exercise a given set of changes. The `pc_select` script addresses this problem. It assumes that:

- Your test harness supports specification of an arbitrary subset of tests to run.
- There is a set of `.pcv` files available, one per testcase, that might need running; and further that you have some way to determine from the name of a `.pcv` file which testcase corresponds to it.
- You can identify which source files were modified.
- There is some way to determine for each source file the full set of `diff` output identifying the changes made since the coverage data for each test was collected.

When `pc_select` runs, it searches all the coverage files, and identifies each coverage file that exercises any function that was

modified. The resulting list of coverage files is output. These can then be converted into whatever form you need to run just the corresponding tests from the test suite.

```
% <changed files> | pc_select [-diff=<rule>] \
  [-canonicalize=<rule>] <file>.pcv ...
```

The `<changed files>` list piped to `pc_select` via `stdin` should consist of filenames, one per line, identifying each file modified since the coverage data was collected. For example, if you have been making the changes for the last 3 days, you can use a `find` command to produce the correct list, and then pipe the list into the `pc_select` command:

```
% find * -type f -mtime -3 -print | pc_select
```

The list of `<.pcv>` files on the command line should be the full list of coverage data corresponding to the test suite. For example,

```
% /usr/builds/951211/tests/*.pcv
```

The `-diff <rule>` option is used to specify exactly how to obtain the `diff` output for each file. It is a generic Perl expression that is applied (via `eval`) to each file on `stdin` in turn to obtain the `diff` output for each file. When the expression is evaluated, the variable `$file` contains the name as it was found on `stdin`, and for convenience the variable `$_` is also set to the same value.

Note: Since the shell and Perl use some of the same metacharacters, it is very important to apply appropriate quotation marks and escapes to make sure the correct program parses the value.

The default value for the `-diff <rule>` is

```
-diff='`rcsdiff $file`'
```

which says to use `rcsdiff` on each file to look up the changes in the RCS data. Another example value is

```
-diff='`diff $file.original $file`'
```

which says to look for a file foo.original from which the new version of foo is descended, then to use the standard diff program to calculate the changes.

Note: While the script can detect syntax errors in the specification, it cannot detect various other problems, such as reversing the order of the two files and argument errors to the command which cause it to produce no output.

The -canonicalize <rule> is used in the case where the fullpaths in the .pcv files do not match those of the files which were being modified. For example, suppose that the name of the original file.c when it was built for the overnight full test run was:

```
/usr/builds/951211/src/file.c
```

while the fullpath in your development area is now:

```
/usr/home/pat/work/file.c
```

In this case, the pc_select script looks for coverage information for the file under the latter name, but does not find it because all the information in the .pcv file is under the former name.

The -canonicalize <rule> is a generic Perl expression that is applied (via eval) to each file in turn in order to determine what name to use to locate coverage for it in the .pcv files. When the expression is evaluated, the variable $file contains the name as it was found on stdin, and for convenience the variable $_ is also set to the same value.

The default value of the -canonicalize <rule> is:

```
-canonicalize='chop(local($here)=`pwd`); "$here/$file"'
```

This converts the relative pathname to an absolute pathname, using pwd to calculate the fullpath. For the example shown here, if the filename is just the basename of the file, you can use:

```
-canonicalize='"/usr/builds/951211/src/$file"'
```

Note: While the script can detect syntax errors in this specification, it cannot detect problems such as incorrect use of

automounter prefixes or specification of a fixed value instead of a variable value.

Custom reports

You can use the PureCoverage export format to write your own scripts to produce custom reports.

The PureCoverage export format is in ASCII and is used to transfer coverage data to other programs. The export format contains nested summary information about directories, files, functions, lines, and blocks. For details, see Appendix B, “Export Format.”

To create a custom report:

- Convert the appropriate PureCoverage data files to export format. To obtain export data, use the command `purecov -export`. See page 7-17 for details about the use of the `-export` mode option.
- Read the export data and process it into the desired form.

A sample custom report script

This example shows how to write a simple report script `onefunc` that prints the line coverage percentage for a specific function within a program. The script syntax is:

```
onefunc <program> <function>
```

where `<program>` is the name of the program to search, and `<function>` is the name of the function to display. The script automatically appends the `.pcv` extension to the program name, and it ignores the possibility of invalid parameters.

The custom report script `onefunc` is shown here. You can use it as a template for writing scripts of your own.

```
#!/usr/local/bin/perl
#   Usage: onefunc program function
#   Reports coverage for "function" which is used by "program".
#   Example: onefunc myprog main

open(EXPORT, "purecov -export $ARGV[0].pcv |");
while (<EXPORT>) {
    if (/^di/) { ($key, $curdir)  = split(/\t/); }
    if (/^fi/) { ($key, $curfile) = split(/\t/); }
    if (/^fu/) {
        ($key, $curfunc, $ign, $ign, $unused, $lines) =
            split(/\t/);
        if ($curfunc eq $ARGV[1]) {
            $count    = $lines - $unused;
            $percent = ($count * 100) / $lines;
            push(@matches,
                sprintf("%3d%% %s%s:%s\n", $percent, $curdir,
                    $curfile, $curfunc));
        }
    }
}
close(<EXPORT>);
print sort @matches;
```

The basic script structure uses `purecov` to produce the export data, and then extracts the function information and records it in the array `@matches`. It uses `sort` to organize the data, in case the function name is used more than once in the program.

The command line to `purecov` uses the `-export` option with `$ARGV[0]`, the program name, to export data from the program you specify when you run the script.

The `while` loop has three major pattern-matching sections. This report handles three different types of lines from the export data:

- The name of the directory containing the file
- The name of the file containing the function
- The name of the function plus the coverage information for the function

The caret (^) in front of the keywords makes the patterns match only at the beginnings of lines. For the file and directory lines, you only need to record the name. For the function line, you also want to record the line coverage data from `$unused` and `$lines`, calculate the coverage percentage, and print function information.

When you run the script, you get coverage information for the function you specify.

```
% chmod +x onefunc
% onefunc myprog myfunc
  70% /home/pat/dir1/file1.c:myfunc
```

7 PureCoverage Options

PureCoverage options let you customize PureCoverage operations. By specifying build-time or run-time options in the link line or in environment variables, you can control how your program is instrumented and linked, and how data is collected. By specifying options after running the program, you can choose how coverage data is presented for analysis.

Option tables

Build-time options	Default	Page
-always-use-cache-dir	no	7-7
-auto-mount-prefix	/tmp_mnt	7-10
-cache-dir	<purecovhome>/cache	7-7
-collector	none	7-9
-forbidden-directories	system-dependent	7-8
-ignore-run-time-environment	no	7-6
-linker	system-dependent	7-9

Run-time options	Default	Page
-auto-mount-prefix	/tmp_mnt	7-10
-counts-file	%v.pcv	7-12
-follow-child-processes	no	7-12
-force-merge	no	7-15
-handle-signals	none	7-13
-ignore-signals	none	7-13
-log-file	stderr	7-11
-program-name	argv[0]	7-11
-run-at-exit	none	7-14
-user-path	none	7-10

Analysis-time options	Default	Page
`-apply-adjustments`	`yes`	7-15
`-force-merge`	`no`	7-15
`-user-path`	none	7-10

Analysis-time mode options	Page
`-export`	7-17
`-extract`	7-17
`-merge`	7-18
`-view`	7-18

Informational options	Page
`-print-home-dir`	7-19
`-version`	7-19

Using PureCoverage options

PureCoverage option syntax

Each PureCoverage option is a word or phrase that begins with a hyphen, for example, `-force-merge`.

- The leading hyphen is required.
- PureCoverage ignores case, hyphens, and underscores in the option name. For example, the option `-force-merge` is equivalent to `-FORCE_MERGE` and `-ForceMerge`.
- For options that take a list of directories, use spaces or colons (`:`) to separate directory names.
- Many options have default values. You can override these values using the syntax `-option_name[=[value]]`.
- Do not use a space on either side of an equal sign (`=`).

Using conversion characters in filenames

When you specify filenames for options such as `-log-file` and `-counts-file`, you can include conversion characters that expand to the program name or to the process id (pid):

Character	Converts to
%V	Full pathname of the program ("/" replaced by "_").
%v	Program name.
%p	Process id (pid).

If the filename is *unqualified* (does not contain "/"), PureCoverage writes it to the directory where the program resides. *Qualified* filenames can be absolute or relative to the current working directory. For example, if you specify the option

```
-log-file=./%v.plog
```

PureCoverage writes the log file to the current working directory. If the program is called `test`, the log file is called `./test.plog`.

PureCoverage option types

PureCoverage supports two types of options: boolean and string:

- Boolean options take the values `yes` or `no`.
  - The value `yes` applies if you specify an option but no value. For example, `-force-merge` is equivalent to `-force-merge=yes`.
  - The default value for an option applies if you do not specify the option at all.
  - For the `-apply-adjustments` option, you must specify `-apply-adjustments=no` to turn the option off.
- String options can be a string of any kind. If you specify an option without an explicit value, the value is cleared.

  For example, the option `-log-file=./purecovout` routes PureCoverage messages to the file `purecovout` in the current directory. The option `-log-file` without a value clears any specification of a logfile.

PureCoverage option processing

You can specify PureCoverage options in several locations:

- In the environment variables `PURECOVOPTIONS` or `PUREOPTIONS`
- On the link line
- On the `purecov -view`, `purecov -merge`, or `purecov -export` command line

Specifying options in environment variables

You can specify PureCoverage options in the environment variables `PURECOVOPTIONS` or `PUREOPTIONS`. Values in `PUREOPTIONS` apply to PureCoverage, Purify, and Quantify software products. Values in `PURECOVOPTIONS` take precedence over values in `PUREOPTIONS`.

PureCoverage applies *build-time* options specified in environment variables when it builds instrumented applications. Any build-time options on the link line override build-time environment variables.

PureCoverage applies *run-time* options specified in environment variables when you run instrumented applications. The run-time environment values in force when you run the program override any values specified on the link line.

Note: If an option is specified more than once in an environment variable, PureCoverage applies the *first* value. To add an overriding value for the `-log-file` option without changing other options specified, use a command like:

```
csh   %   setenv PURECOVOPTIONS "-log-file=new \
              $PURECOVOPTIONS"
ksh   $   export PURECOVOPTIONS="-log-file=new \
              $PURECOVOPTIONS"
```

Using the PUREOPTIONS environment variable

You can use the `PUREOPTIONS` environment variable to set options that apply to all users of PureCoverage, Purify, and Quantify

software products. PUREOPTIONS is convenient for specifying defaults that apply to all three applications.

For example, if your site has a central shared file that is sourced by all users' .cshrc or .profile files, you can set -cache-dir=/alternate/dir in the environment variable PUREOPTIONS to apply to all users.

Setting options for Purify and PureCoverage

When using Purify and PureCoverage together, you can use a command such as:

```
% purify <purifyoptions> purecov <purecovoptions>\
      $CC ...
```

In this case, both <purifyoptions> and <purecovoptions> are used, as well as the values of the environment variables PURIFYOPTIONS, PURECOVOPTIONS, and PUREOPTIONS, in that order of precedence. Where there are conflicting values, the first value seen is used. A value in <purifyoptions> overrides a value in <purecovoptions>. These options override environment variables.

Specifying options on the link line

You can specify build-time and run-time options on the link line. For example:

```
% purecov -cache-dir=${HOME}/pcache \
      -always-use-cache-dir $CC ...
```

Build-time options apply to the PureCoverage build command being run.

Run-time options are built into the executable and become the default values for the instrumented executable. This provides a convenient way to build a program with nonstandard default values for run-time options. For example:

```
% purecov -force-merge $CC ...
```

Using the ignore-run-time-environment option

You can use the `-ignore-run-time-environment` option when you build your executable to make sure that the run-time options you specify remain in effect whenever the executable is run.

The `-ignore-run-time-environment` builds into an executable all the run-time options specified on the link line along with any run-time options specified in the `PURECOVOPTIONS` and `PUREOPTIONS` environment variables.

When the instrumented program runs, PureCoverage ignores the current option values set in environment variables in preference to the built-in values.

Use the `-ignore-run-time-environment` option when:

- You want someone else to run your program without their run-time environment modifying your run-time option specifications
- Your program is started automatically by another program and you cannot set the environment variable for that program
- You have several instrumented programs running at one time and you cannot specify options for each program

To find out what options are built into a PureCoverage instrumented program, use:

```
% <purecovhome>/purecov_what_options <program name>
```

Note: Use the `-ignore-run-time-environment` option at build time only. PureCoverage ignores this option if you specify it at run time.

Using analysis-time mode options

When using the `-view`, `-export`, `-merge`, or `-extract` mode options, you can specify additional options on the command line. These additional options have precedence over any options obtained from the environment variables `PURECOVOPTIONS` or `PUREOPTIONS` for that operation.

The table beginning on page 7-17 lists the options that are compatible with each mode.

Build-time options

Options for caching

To improve build-time performance, PureCoverage caches its versions of all the libraries and object files used to build the program. By default, files instrumented by PureCoverage are written into the original file's directory if it is writable. If this directory is not writable, PureCoverage writes into the central cache directory.

If you need to manage disk space, you can safely remove the cached files by using the script `pure_remove_old_files`. PureCoverage rebuilds any shared libraries required at program run-time. See page 2-12.

Option	Default
-cache-dir	**<purecovhome>/cache**
This option sets the central cache directory for caching instrumented versions of files.	
-always-use-cache-dir	**no**
This option forces PureCoverage to write *all* instrumented versions of files to the central cache directory.	

Option	Default
-forbidden-directories	**system-dependent**

You can use this option to specify a list of directories, each separated by a colon, into which files are never written, even if the directories listed are writable. All the subdirectories of forbidden directories are also forbidden. The default values are system-dependent:

`/lib:/opt:/usr/lib:/usr/5lib:/usr/ucb/lib:/usr/lang:/usr/local`

`/lib:/opt:/usr/lib:/usr/4lib:/usr/ucblib:/usr/lang:/usr/local`

`/lib:/usr/lib:/usr/local`

Options for linker and collector

Option	Default
-linker	**system-dependent**

This option specifies the name of the linker for PureCoverage to invoke to produce the executable. Use this option only if you need to bypass the default linker.

The defaults are:

```
/bin/ld
```

```
/usr/ccs/bin/ld
```

Note: Do not use this option to specify PureLink. For instructions on using PureLink with PureCoverage, see page 3-12.

Option	Default
-collector	**none**

This option specifies the name of the collector program to be used to sequence and collect static constructors in C++ code. It is *mandatory* to set this option to the name of the collector program if you are using the g++ compiler.

You can find the name of the collector program used by the g++ compiler with the command:

```
% g++ -v foo.cc
```

If the collector program is

```
/usr/local/lib/gcc-lib/sun-sparc-sunos4/2.1/ld
```

use the command:

```
% purecov -g++=yes -collector=\
/usr/local/lib/gcc-lib/sun-sparc-sunos4/2.1/ld \
g++ foo.cc
```

Solaris 2 does not use a collector for C++ programs, so PureCoverage on Solaris ignores this option.

Run-time options

Options for file identification

Option	Default
-auto-mount-prefix	**/tmp_mnt**

This option is used to specify the directory prefix used by the file system automounter, usually /tmp_mnt, to mount remote file systems in NFS environments. Use this option to strip the prefix, if present, to improve the readability of source filenames in PureCoverage reports.

If your automounter *alters* the prefix, instead of *adding* a prefix, use the syntax -auto-mount-prefix=/tmp_mnt/home:/homes to specify that the real file name is constructed from the apparent by replacing /tmp_mnt/home with /homes.

If you do not set this option correctly, PureCoverage may be unable to access files on automounted filesystems. The automounter may not recognize their names.

Option	Default
-user-path	**none**

This option specifies a list of directories to search for source code.

When searching for source code to build annotated source listings, PureCoverage looks for the file in the full pathname specified in the debugging data, then in directories listed in -user-path, and finally in the current directory.

PureCoverage can also use -user-path at instrumentation time to help resolve source-code path names if the debugging data is incomplete or the files have been moved. The resolved name is built into the executable and is used in the export data and at analysis time.

The directory where the object file resides and the current working directory are automatically appended to the end of the -user-path directory list to help find source files with missing or incorrect directory names.

Option	Default
-program-name	**argv[0]**
This option specifies the full pathname of the instrumented program. Use this option if argv[0] contains an undesirable or incorrect value—for example, when your program is invoked by an exec call whose path differs from the argument it passes as argv[0] to your program. In such cases, PureCoverage cannot find the program file, and therefore cannot match addresses to function names.	
-log-file	**stderr**
If this option is not set, PureCoverage run-time messages are sent to the program's stderr stream. If this option is set to a filename, PureCoverage run-time messages are sent to the named file. Sometimes it is more important that the PureCoverage instrumented application run exactly like the uninstrumented application, with no additional output to stderr, than it is to be reminded that coverage data is being collected. You can silence run-time messages by specifying the option -log-file=/dev/null. If your application is instrumented for both PureCoverage and Purify, PureCoverage's start-up and data recording messages go to the Purify Viewer. If the option -log-file is set, the same logfile receives both PureCoverage and Purify output. If you set PURIFYOPTIONS -log-file=plog and PURECOVOPTIONS -log-file=xlog, the value in PURIFYOPTIONS has precedence. The -log-file option can be used with filename conversion characters. See "Using conversion characters in filenames" on page 7-3.	

An option for saving data

Option	Default
-counts-file	**%v.pcv**

This option specifies the pathname and filename for saving the accumulated coverage counts. The default filename is the name of the program with a .pcv extension; the default location is the directory where the program resides.

The option -counts-file can be used with filename conversion characters. See "Using conversion characters in filenames" on page 7-3.

An option for data collection

Option	Default
-follow-child-processes	**no**

This option controls whether PureCoverage monitors child processes created when an instrumented program forks.

If this option is not specified, PureCoverage does not accumulate counts for child processes, and does not write counts data for the child process to the .pcv file.

If this option is set to yes, PureCoverage accumulates coverage data for the child process. When the child process exits or executes another process, PureCoverage writes the accumulated counts in the child to the .pcv file.

Likewise, when the parent process exits, PureCoverage writes counts accumulated in the parent to the .pcv file. By default, PureCoverage writes both parent and child counts to the same .pcv file (%v.pcv), so that viewing that file shows the combined coverage data.

To separate the counts for parent and child, you can use the option -counts-file=%v.%p.pcv to specify a counts file including the process pid in the filename, or call the API function purecov_set_filename() in either or both the parent or child process to control where the data is written.

Options for signal handling

Option	Default
-handle-signals	**none**
-ignore-signals	**none**

PureCoverage installs a signal handler for many of the possible software signals which are delivered to an instrumented process. The signal handler prints an informative message and saves coverage data to the `.pcv` file in case the process crashes.

The signal handler installed by PureCoverage outputs a signal message to `stderr`, or to the Purify Viewer if the process is Purify'd. If the signal is a fatal signal such as a `SEGV`, and no user signal handler has been installed by the program to catch such a condition, the coverage data is written out to the `.pcv` file before the normal signal termination occurs. If the instrumented program has installed a signal handler, the PureCoverage handler will pass control to that handler instead.

The default list of signals that are handled by PureCoverage is:

```
SIGHUP SIGINT SIGQUIT SIGILL SIGIOT SIGABRT SIGEMT SIGFPE SIGBUS
SIGSEGV SIGSYS SIGPIPE SIGTERM SIGXCPU SIGXFSZ SIGLOST SIGUSR1 SIGUSR2
```

```
SIGHUP SIGINT SIGQUIT SIGILL SIGIOT SIGABRT SIGEMT SIGFPE SIGBUS
SIGSEGV SIGSYS SIGPIPE SIGTERM SIGUSR1 SIGUSR2 SIGPOLL SIGXCPU SIGXFSZ
SIGFREEZE SIGTHAW SIGRTMIN SIGRTMAX
```

```
SIGHUP SIGINT SIGQUIT SIGILL SIGABRT SIGEMT SIGFPE SIGBUS SIGSEGV
SIGSYS SIGPIPE SIGTERM SIGUSR1 SIGUSR2 SIGLOST SIGRESERVE SIGDIL
SIGXCPU SIGXFSZ
```

To skip handling signals in this list, set the value of the option `-ignore-signals` to a comma-separated list of the signal names not to be handled. For example:

```
-ignore-signals=SIGSEGV,SIGBUS
```

Option description continued on next page.

To handle additional signals, set the value of the option `-handle-signals` to a comma-separated list of the additional signal names to be handled. For example:

```
-handle-signals=SIGALRM,SIGCHLD
```

These signals are added to the current list of handled signals.

PureCoverage does not handle `SIGKILL`, `SIGSTOP`, or `SIGTRAP`, since doing so interferes with normal program operation. If you specify these signals in `-handle-signals`, they are silently ignored.

The default action upon delivery of `SIGALRM` is to terminate the process. PureCoverage does not handle this signal by default, since functions like `sleep` use it internally. However, if you see a process terminated with a message like `Alarm clock`, you can set `-handle-signals=SIGALRM` to save data when the program is terminated.

See the man pages for `signal` and `sigmask`, and the files `/usr/include/signal.h` and `/usr/include/sys/signal.h` for more information on signals.

An option for exit processing

Option	Default
`-run-at-exit`	**none**

This option specifies an arbitrary shell command to be run when your program exits or otherwise terminates. In addition to filename conversion characters, you can also use the directive `%x` for the program's exit status (0 if the program did not call exit).

You can use this option to log the coverage data that results from each session of a program into a testing database system. You can use it to trigger a script such as `pc_email` at the end of a test run.

Analysis-time options

An option for handling adjustments

Option	Default
`-apply-adjustments`	**`yes`**

This option specifies whether PureCoverage applies coverage adjustments when you export coverage data.

This option applies only to the command `purecov -export` and to the report scripts. Adjustments are always applied in the Viewer, though you can choose to see the unadjusted statistics in the Viewer and the unadjusted source code in the Annotated Source window.

An option for merging

Option	Default
`-force-merge`	**`no`**

The `-force-merge` option controls PureCoverage's behavior when combining data obtained from different versions of the same object file. PureCoverage uses an object-code checksum to detect when the code has changed. By default, if a file appears to have changed, PureCoverage uses the data from the most recent version and discards the data from the earlier version.

PureCoverage merges and discards data on an object-file basis. If you change one part of an application, re-compile and make additional runs, PureCoverage discards data only for the changed code and continues to accumulate it for the rest of the application.

You can combine data from different versions of an object file. For example, suppose a program varies only slightly owing to conditional compilation:

```
#ifdef VERSION1
     do_something();
#else
     do_something_else();
#endif
```

To combine data from these two different versions of the object file, use the option `-force-merge`. PureCoverage combines the data even if it is not from the same program.

Option description continued on next page.

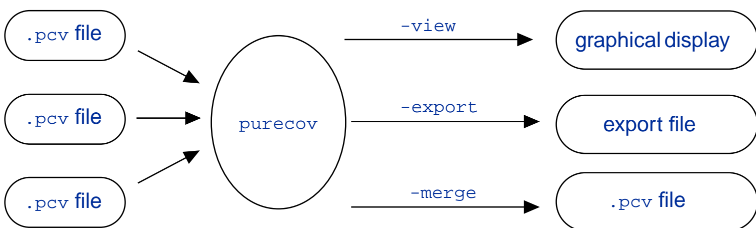
For example:

```
% purecov cc -g -DVERSION1 myprog.c -o myprog1
% purecov cc -g -DVERSION2 myprog.c -o myprog2
% myprog1
% myprog2
% purecov -view -force-merge myprog1.pcv myprog2.pcv
```

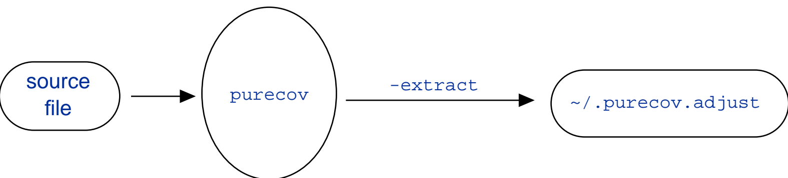
Note: Do not use this option unless you are absolutely certain that the two versions were both built from precisely the same source.

Analysis-time mode options

Three of the four analysis-time mode options, `-view`, `-export`, and `-merge`, work with the same input, `.pcv` files, but generate different output:



The fourth analysis-time mode option, `-extract`, uses source files as input, and generates or updates the file `~/.purecov.adjust`.



Mode option

`-export`

This option instructs PureCoverage to generate export data from one or more `.pcv` files. For example, use the command

```
% purecov -export=result.export a.pcv
```

to save export data from the `a.pcv` file in the `result.export` file. Use the following command to save export data from the merged `a.pcv` and `b.pcv` files in the `result.export` file:

```
% purecov -export=result.export a.pcv b.pcv
```

Note: When you use the `-export` option on multiple files, PureCoverage performs an implicit merge, but does not save the results in a merged `.pcv` file.

If you do not set a value, PureCoverage sends its output to `stdout`. For example, use the command

```
% purecov -export a.pcv b.pcv
```

to send export data for the merged `a.pcv` and `b.pcv` files to `stdout`.

Compatible Options: You can use the following option on the command line when you start PureCoverage in export mode: `-apply-adjustments`.

`-extract`

This option extracts coverage adjustments from specified source files and stores them in the file `~/.purecov.adjust`. For example, use the command

```
% purecov -extract file1.c file2.c
```

Note: Choosing **Extract adjustments from all files** from the Viewer Adjustments menu is equivalent to running this option for all source files listed in the `.pcv` data in memory.

Compatible Options: None.

Mode option

-merge

This option instructs PureCoverage to merge the coverage data from multiple .pcv files, and write a merged .pcv file. For example, use the command

```
% purecov -merge=result.pcv a.pcv b.pcv
```

to merge a.pcv and b.pcv into result.pcv.

Note: You must specify an output file with this option.

Compare also the two following examples:

```
% purecov -merge=result.pcv a.pcv
```

```
% purecov -merge=result.pcv result.pcv b.pcv
```

The first example is just a way to copy a.pcv to result.pcv. The second instructs PureCoverage to read result.pcv completely, and then combine it with b.pcv *before* overwriting result.pcv with the new, merged data.

Compatible Options: You can use the following option on the command line when you start PureCoverage in merge mode: -force-merge.

-view

This option displays coverage data for one or more .pcv files in the PureCoverage Viewer. For example, use the command

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

to display the a.pcv data in the Viewer. Use the following command to display data from the merged a.pcv and b.pcv files in the Viewer:

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

Note: When you use the -view option on multiple files, PureCoverage performs an implicit merge, but does not save the results in a merged .pcv file.

Note: By default, PureCoverage does not invoke the Viewer at run time. You must invoke it after collecting coverage data from one or more test runs.

Compatible Options: You can use the following options on the command line when you start PureCoverage in view mode: -force-merge, -user-path.

Informational options

The informational options each return a single item of information. You can specify these options, one at a time, on a command line after `purecov`. You can also embed them in other commands for evaluation before the commands are executed.

Option	Default
-print-home-dir	
This option instructs PureCoverage to print the name of the directory where PureCoverage is installed, and then exit. You can use this option, for example, to build the compiler command when including the `purecov.h` file from the installation directory: `$CC -c $CFLAGS -I`purecov -print-home-dir` foo.c`	
-version	

This option instructs PureCoverage to print its version number string to `stdout` and then exit. You can use this option, for example, to identify which version of PureCoverage is in use while running a test suite, by incorporating this line in your test harness scripts:
`#!/bin/sh`
`...`
`echo "Run monitored by PureCoverage: \`
`    `purecov -version`"..`	

PureCoverage API

Use the PureCoverage application programming interface (API) for fine-grained control over the collection and management of PureCoverage data. The API functions let you control what part of your code has data collected, when the data is saved, and where it is saved.

By default, PureCoverage collects data from the beginning of the program through the exit of the program, and saves it at the exit of the program regardless of whether the program completes successfully. The API functions enable you to change this behavior to suit your needs.

API Function	Page
`int purecov_set_filename (const char *file_name)`	8-3
`int purecov_save_data (void)`	8-3
`int purecov_clear_data (void)`	8-3
`int purecov_disable_save (void)`	8-3
`int purecov_enable_save (void)`	8-4
`int purecov_is_running (void)`	8-4

Calling PureCoverage API functions from your program

You can call PureCoverage functions directly from your program. Use `#include <purecov.h>` to include the file `purecov.h` in your C or C++ programs.

The header file is located in the PureCoverage home directory. Add the compiler option `-I<purecovhome>` in your makefile, if necessary.

If you add calls to the PureCoverage functions to your code, you can add the library `<purecovhome>/purecov_stub.a` to your link

line. This is a small library that stubs out all the API functions. When you are *not* using PureCoverage, these stubs satisfy the linker; when you *are* using PureCoverage, these stubs are ignored.

```
PFLAGS              = -force-merge
PDIR                = `purecov -print-home-dir`
PURECOV             = purecov $(PFLAGS)
PSTUBS              = $(PDIR)/purecov_stubs.a

# General flags
CC                  = cc
CFLAGS              = -g -I$(PDIR)

# Targets
all: a.out a.out.pure

a.out: hello_world.c
        $(CC) $(CFLAGS) -o $@ $? $(PSTUBS)

a.out.pure: hello_world.c
        $(PURECOV) $(CC) $(CFLAGS) -o $@ $? $(PSTUBS)
```

Calling PureCoverage API functions from a debugger

You can use the PureCoverage API functions by setting breakpoints in the debugger running your instrumented program and then calling the appropriate function when the breakpoint is reached.

```
(gdb) call purecov_clear_data()
(dbx) print purecov_clear_data()
```

Data collection API functions

int purecov_set_filename (const char *filename)

This function sets the name of the file for saving or merging the accumulated counts when the program exits or executes another program. This call does not actually write data or clear the counts.

The filename is subject to expansion of conversion characters such as `%v`, `%V`, `%p`. For details, see "Using conversion characters in filenames" on page 7-3.

If the filename string is `NULL`, the default filename is restored. This is typically the name of the program with the `.pcv` extension (`%v.pcv`), unless you specify the `-counts-file` option.

If the filename is unqualified (contains no "/"), it is written in the directory where the program lives. A qualified filename can be either absolute or relative to the current working directory.

int purecov_save_data (void)

This function takes the accumulated counts at the time of the call and writes or merges them into the counts file. It then clears the counts to zero.

You can use this function with a long-running program to checkpoint coverage data incrementally. Alternatively you can use it to distribute counts for different phases of a program's run to different files.

int purecov_clear_data (void)

This function resets all data accumulators to zero. This permits you to reset coverage information after a section of the code has failed, while continuing to collect information about the session after the failure. You can also use this to ignore coverage up to a certain point—for example, if you have a function used by several parts of the code but you are specifically interested in the coverage obtained by one section of the code.

int purecov_disable_save (void)

This function sets a flag which prevents PureCoverage from writing the accumulated data when the program exits or executes another program.

For example, you can call this function within a self-testing program. If the program detects a failure, the function keeps PureCoverage from writing coverage data automatically when the failed program exits. A signal handler detecting a fatal signal can call this function before exiting.

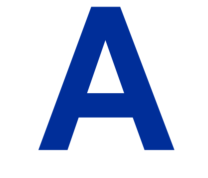
`int purecov_enable_save (void)`

This function clears the flag set by `purecov_disable_save`, permitting PureCoverage to write the data when the program exits or executes another program.

For example, a self-test program can disable saving at start-up, and only re-enable saving upon positive confirmation of successful completion.

`int purecov_is_running (void)`

This function returns `1` if the executable is instrumented with PureCoverage, `0` if not.



Common Questions

Customizing coverage

How do I keep PureCoverage from instrumenting specific lines of code?

PureCoverage allows you to mark specific lines in order to exclude them from the coverage statistics and prevent them from being highlighted as untested in the Annotated Source window. See "Adjusting coverage on a line-by-line basis" on page 4-4.

You can exclude a function or file by marking all lines in the function or file.

For details on the `exclude` directive, which allows you to exclude groups of files, see "Customizing data collection" on page 4-2.

Is there any way I can exclude some libraries after I build my executables?

No, because `exclude` directives are read at build-time. You need to re-link your program for new directives to take effect.

I excluded `libfoo.a` and rebuilt my application. Why do I still see coverage data for this file when I use `purecov -view` to examine the results?

You are probably seeing data in the `.pcv` file that was collected before you excluded the library. Remove the `.pcv` file and start over. You should no longer see data collected for this library.

If you still see data, your `exclude` directive probably does not match the exact pathname as shown in the coverage summary. If the pathname is very long, expand the width of the Viewer to see it all. Or use the script `pc_summary` to generate an ASCII report in which pathnames are not truncated.

Make sure your `exclude` directive matches the pathname as recorded here. It is best to use wildcards for the root of the pathname in the directive, or to use unqualified file names. For details on the `exclude` directive, see “Customizing data collection” on page 4-2.

How come, if I exclude a library, it still gets instrumented?

The OCI technology that PureCoverage uses requires that all files receive a minimum level of instrumentation. The `exclude` directive provides this minimum level without the additional instrumentation for the coverage collection that PureCoverage normally performs.

This partial instrumentation is required so that the fully instrumented parts of the program can successfully call the excluded parts. Overhead for partial instrumentation is minimal—significantly less than counting the coverage data at run time.

General questions

What is a basic block?

A basic block is an indivisible sequence of instructions always executed together in succession.

An example is:

```
1     a++;
2     b++;
3     c = a+b;
4     if (a == b) { c = 0;
5         d = 0;
6     }
7     e = 0;
```

The statements on line `1`, `2`, and `3`, and the comparison on line `4`, all form one *basic block* that ends with the conditional jump implementing the `if` statement.

The assignments on line `4` and `5` form a *second basic block*.

Line `7` begins a *third basic block*, because the assignment `e = 0;` is not indivisibly attached to those on lines `4` and `5`, since the `if` statement on line `4` may cause those assignments to be skipped.

Do I have to specify the `-g` debugging option when I compile my code in order to obtain coverage data? If I don't compile with the `-g` option, what kind of coverage will PureCoverage provide?

For code compiled with the `-g` option, PureCoverage provides both line-level and function-level coverage information. For code compiled without `-g`, PureCoverage provides only function-level coverage information.

Can I use both the -g and -O options when I compile?

This is not useful when you are working with PureCoverage, even if your compiler allows it. The coverage data collected from such a file is confusing because the instructions for each line are sometimes deleted, repeated, or interleaved with other lines.

I notice differences in annotation when I use different compilers or move to a different platform. Why is this?

Compilers add debugging data differently. Since PureCoverage uses the debugging data to provide coverage information, you will notice differences in annotation. For details, see Appendix C, "Annotation Variations."

I have several libraries, parts of which get exercised by different executables. How do I get a coverage report showing total coverage?

You can combine the `.pcv` files for all your executables using `purecov -merge`. You can also simply list them all on the command line to `purecov -view` or one of the coverage analysis scripts. If the shared code is built with different conditional compilation options, try using the option `-force-merge`, but only if you are *certain* that the files you are merging share identical source code.

For details see "An option for merging" on page 7-15.

Performance issues

What is the overhead when using PureCoverage?

Your PureCoverage instrumented program should typically run 1.5 to 3 times slower than your non-instrumented program.

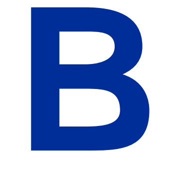
Code compiled with debugging data (`-g`) runs more slowly than code without, because PureCoverage adds many additional counters to note line coverage data.

When the program exits and saves data, additional overhead is incurred, roughly proportional to the size of the program. To minimize this overhead, exclude libraries such as `libc`, `libC`, `libX`, and `libm` from coverage.

The data from these libraries is not likely to be useful, and the time taken to save coverage data for all the unused code in the library may be substantial. This is particularly important when the libraries are shared libraries, and *all* functions in the library are present, not just the ones your application requires.

Will the overhead of Purify and PureCoverage be combined if I instrument my program with both products?

The overhead when using Purify and PureCoverage is less than the combined overhead for using these products separately.



Export Format

Export format description

Once you have collected coverage data, you can convert it to PureCoverage export format for processing by a script. The PureCoverage export format is in ASCII. To convert data to export format, use the syntax:

```
% purecov -export=tests.export test1.pcv test2.pcv test3.pcv...
```

In export format, data records appear one per line, with a keyword defining the type of data followed by data values. The keyword is always the first word on the line. The values are positional, and fields are separated by tabs. Blank lines can appear in the file.

Each line in the export file contains one type of information:

Keyword	Line	Type of information
`to`	total	Summary information about all the data
`di`	directory	Start of information about a source directory
`edi`	end directory	End of information about a source directory
`fi`	file	Start of information about a file
`efi`	end file	End of information about a file
`fu`	function	Start of information about a function
`efu`	end function	End of information about a function
`li`	line	Information about a single line of source code
`bl`	block	Information about a single basic block of object code

Note: You can use the # symbol to add comments or other documentation to an export file.

Each export file contains a header describing the key and field meanings:

```
##############################################################################
# This is a PureCoverage export file, created 18-Apr-1996 08:52:56 AM        #
# Coverage data type: adjusted.                                              #
#                                                                            #
# Summary of keys and field meanings:                                        #
#                                                                            #
# to unused-blocks blocks unused-lines lines unused-funcs funcs files dirs   #
# di name unused-blocks blocks unused-lines lines unused-funcs funcs files   #
# fi name unused-blocks blocks unused-lines lines unused-funcs funcs sessions #
# fu name unused-blocks blocks unused-lines lines calls                      #
# li line_no hits                                                            #
# bl line_no hits                                                            #
# efu name                                                                   #
# efi name                                                                   #
# edi name                                                                   #
#                                                                            #
# In all of these entries, the character separating the values is a TAB, not #
# a space.                                                                   #
#                                                                            #
# Values for hits, calls, and sessions stop accumulating once they reach     #
# 10000, so a value of 10000 should be taken as 10000 or greater.            #
#                                                                            #
# 'to'  -- Total coverage summary.                                           #
# 'di'  -- Coverage summary for one directory, followed by the files in the  #
#          directory.  Repeated for each directory.                          #
# 'edi' -- End of information for named directory.                           #
# 'fi'  -- Coverage summary for one file, followed by the functions in the   #
#          file.  Repeated for each file within a directory.                 #
# 'efi' -- End of information for named file.                                #
# 'fu'  -- Coverage summary for one function, followed by line and block     #
#          coverage information for the function.  Repeated for each function #
#          within the file.                                                  #
# 'efu' -- End of information for named function.                            #
# 'li'  -- Use count for the indicated line.  Repeated for each line in the  #
#          function.  If coverage adjustments are NOT applied, then the      #
#          hits figure is the true hits figure for the line.  If coverage    #
#          adjustments ARE applied, then inspected and tested lines will     #
#          indicate INSP or TEST as appropriate, while other used and unused #
#          lines will indicate their counts.  Deadcode lines do not appear   #
#          in export format when adjustments are applied.                    #
# 'bl'  -- Use count for the basic block.  Repeated for each block in the    #
#          function.  Note that a single line may contain multiple blocks,   #
#          and that a single block may span multiple lines.                  #
##############################################################################
```

The information in the export file is nested hierarchically. However, the data is not indented in the export format to show this nesting:

```
to      0       8       0       9       0       3       1       1
di      /usr/home/pat/demo/   0       8       0       9       0       3       1
fi      hello_world.c 0       8       0       9       0       3       2
fu      display_hello_world  0       2       0       2       1
bl      32      1
bl      33      1
li      32      1
li      33      1
efu     display_hello_world
fu      display_message       0       2       0       2       1
bl      39      1
bl      40      1
li      39      1
li      40      1
efu     display_message
fu      main    0       4       0       5       2
bl      22      2
bl      23      1
bl      25      1
bl      26      2
li      22      2
li      23      1
li      25      1
li      26      2
li      27      2
efu     main
efi     hello_world.c
edi     /usr/home/pat/demo/
```

In this example, the directory `/usr/home/pat/demo` contains the file `hello_world.c`. The file `hello_world.c` contains the functions `display_hello_world`, `display_message`, and `main`, and each function contains a number of blocks and lines.

Comments

Comments are indicated by a `#` character. A comment runs from the `#` character to the end of the line. Comments always appear on a separate line.

Total — "to"

There is only a single `to` line in the entire format, and it is always the first information line.

The data fields in a total line are:

```
to <# unused blocks> <# blocks> <# unused lines> \
      <# lines> <# unused funcs> <# funcs> \
      <# files> <# dirs>
```

- The `<# unused blocks>` field indicates the number of unused basic blocks for all source files in any of the test sessions.
- The `<# blocks>` field indicates the total number of basic blocks in the source files, including both used and unused blocks.
- The `<# unused lines>` field indicates the total number of lines that were not used in any of the test sessions, summed across all the source files.
- The `<# lines>` field indicates the total number of source lines in the source files, including both used and unused lines.
- The `<# unused funcs>` field indicates the total number of functions that were not used in any of the test sessions, summed across all the source files.
- The `<# funcs>` field indicates the total number of functions in all the source files.
- The `<# files>` field indicates the total number of source files in all the directories.
- The `<# dirs>` field indicates the total number of directories in the data.

Directory — “di”

There is one directory line for each directory containing source code included in the PureCoverage data being exported. The directory line is followed by information for each file within the directory, and eventually by a matching `edi`, which indicates the end of directory data. No new directory can appear before the matching `edi`.

The data fields in a directory line are:

```
di <name> <# unused blocks> <# blocks> <# unused lines> \
     <# lines> <# unused funcs> <# funcs> <# files>
```

- The `<name>` field indicates the pathname of the directory. As much of the pathname as is known to PureCoverage will be included, usually the full pathname.
- The remaining fields are the same as in a `to` line, except that they are summed across the directory rather than across the entire set of source files.

End Directory — “edi”

An `edi` line follows the information for files, functions, and lines within a directory. It has a single data field:

```
edi <name>
```

- The `<name>` field repeats the name from the matching `di` line.

File — “fi”

A `fi` line provides information about a single file within the containing directory. It is followed by information about functions and lines, and eventually by a matching `efi`, indicating the end of file data. No new file can appear before the matching `efi` line.

The data fields of the file entry are:

```
fi <name> <# unused blocks> < blocks> <# unused lines> \
      <# lines> <# unused funcs> <# funcs> <# sessions>
```

- The `<name>` field contains the name of the file, without any directory information.
- The `<# sessions>` field indicates the number of sessions over which the data for the file was collected. Note that this is not the same for all files if some files were modified more recently than others.
- The remaining fields have the same interpretation as they do in `di` and `to` entries, except they are summed only across all functions or lines within the file.

End File — “efi”

An `efi` line follows the information for functions and lines within a file. It has only a single data field:

```
efi <name>
```

- The `<name>` field repeats the name from the matching `fi` line.

Function — “fu”

A `fu` line provides information about a single function within a file. It is followed by information about lines within the function and eventually by a matching `efu`, indicating the end of function data. No new function can appear before the matching `efu` line.

The data fields of the function entry are:

```
fu <name> <# unused blocks> <# blocks> <# unused lines> \
      <# lines> <# calls>
```

- The `<name>` field contains the name of the function.
- The `<# calls>` field contains the number of times the function was entered.
- The other fields have the same interpretation as in previous examples, except summed only within this specific function.

End Function — “efu”

The `efu` line follows the line information within a single function. It has a single data field:

```
efu <name>
```

- The `<name>` field repeats the function name from the matching `fu` entry.

Line — “li”

The `li` entry contains information about a specific line of source code, which is inside the function named in the surrounding `fu` and `efu` lines. It contains the data fields:

```
li <source line #> <# hits>
```

- The `<source line #>` field gives the line number within the source file. Because it is possible to have multiple functions on the same line of code, source line numbers are not necessarily unique within the export file.
- The `<# hits>` field gives the number of times that specific source line was used in all the sessions contributing to the total for that file.

Block — “bl”

The `bl` entry contains information about a specific basic block of object code, which is inside the function named in the surrounding `fu` and `efu` lines. It contains the data fields:

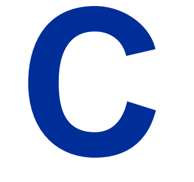
```
bl <source line #> <# hits>
```

- The `<source line #>` gives the first line number which contributed instructions to the basic block. It is possible that other lines following this line have also contributed instructions to the basic block. Because it is possible to have multiple basic blocks on a given line, line numbers are not necessarily unique within the export file.
- The `<# hits>` field gives the number of times that specific basic block was used in all the sessions contributing to the total for that file.

The effect of coverage adjustments on export

When you adjust coverage of a program with the techniques discussed in Chapter 4, the outline-level items of the export format—`to`, `di`, `fi`, `fu`, `li`, and `bl`—are affected in different ways:

- The totals reported for `to`, `di`, `fi`, and `fu` entries reflect the adjustments.
- The `li` entries are affected as follows:
  - For lines marked *tested*, the export format includes the `li` entry, but displays the word `TEST` instead of a use count.
  - For lines marked *inspected*, the export format includes the `li` but displays the word `INSP` instead of a use count.
  - For lines marked *dead code*, the export format suppresses the `li` entry completely.
- The `bl` entries always show unadjusted, or *true*, coverage totals.



Annotation Variations

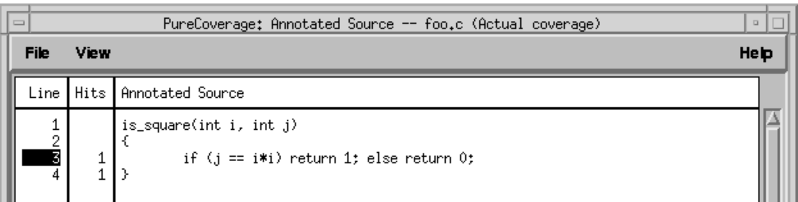
PureCoverage uses debugging information in object files to determine line coverage information. This debugging information is created by the compiler when invoked with the option `-g`.

The debugging information recorded in object files varies from compiler to compiler. There is no single standard. This section identifies some of the differences in how compilers annotate source code.

Note: For readability, this appendix shows Annotated Source windows customized to exclude the adjustment columns and display actual coverage. You can customize your own display by selecting and deselecting items in the window's View menu.

Complex source lines

If a single line contains parts which execute at different times, PureCoverage does not show which part of the line was executed. This is because the compiler does not provide this level of detail in the object file.

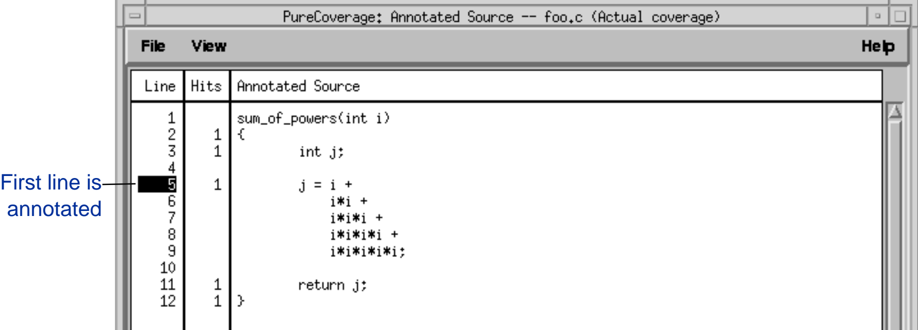


When there are multiple basic blocks on one line, PureCoverage collects separate counts for each basic block. The Annotated Source window, however, displays only the highest count. A partially used line, therefore, is always shown as used.

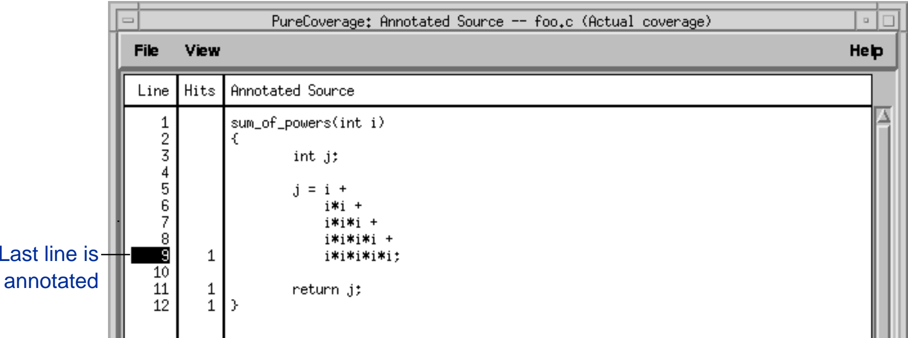
Note: To view separate basic block counts, run the script `pc_annotate` with the option `-type=block`. For details, see "Annotated source report" on page 6-8.

Multi-line statements

Some compilers annotate the *first* line of multi-line statements:



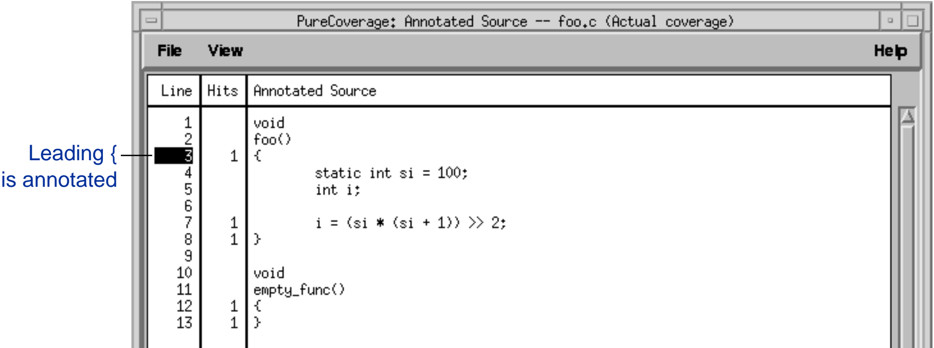
Some compilers annotate the *last* line of multi-line statements:



Function entry points

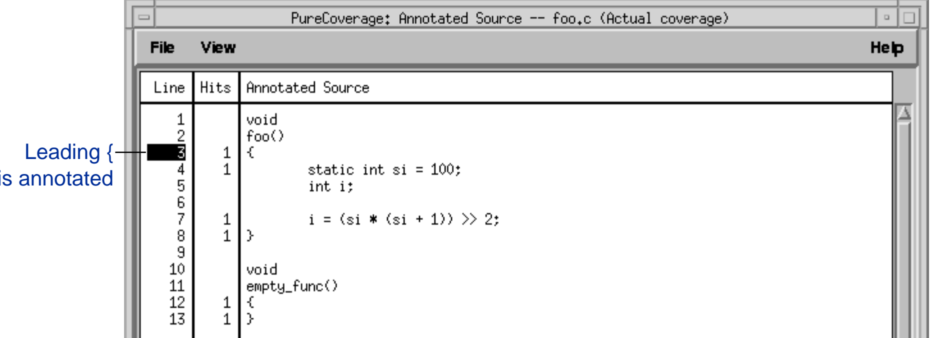
Compilers indicate function entry points in a wide variety of ways:

Some compilers annotate only the leading { as the function entry point:



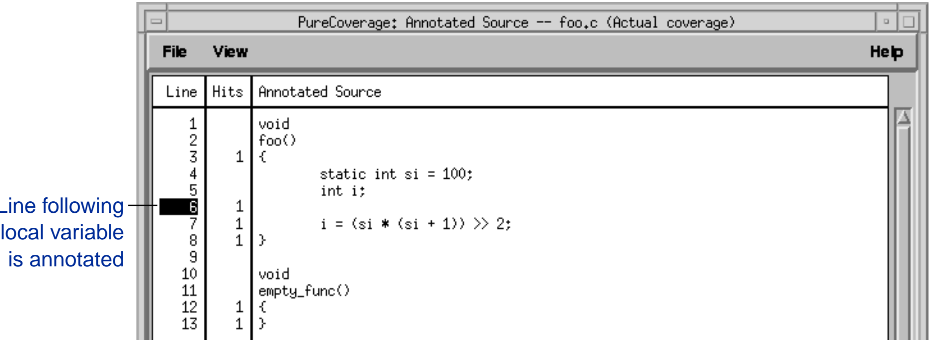
Some compilers annotate the first executable statement as the entry point. If there are no executable statements, as for example in an empty function, the leading { is annotated instead. Variable declarations without initialization clauses do not count as executable statements in this sense.

Some compilers annotate the leading { *and* the first local variable declaration:



Some compilers annotate the first local variable declaration as the function entry point. It does not matter if the variable is declared static or not. If there are no local variables, the leading { is annotated as the function entry point.

Some compilers annotate the line containing the first statement and the line following local variable declarations, even if that line is blank:



Some compilers annotate the function declaration and the first statement.

Some compilers annotate the leading { and the first executable statement.

Local variable declarations

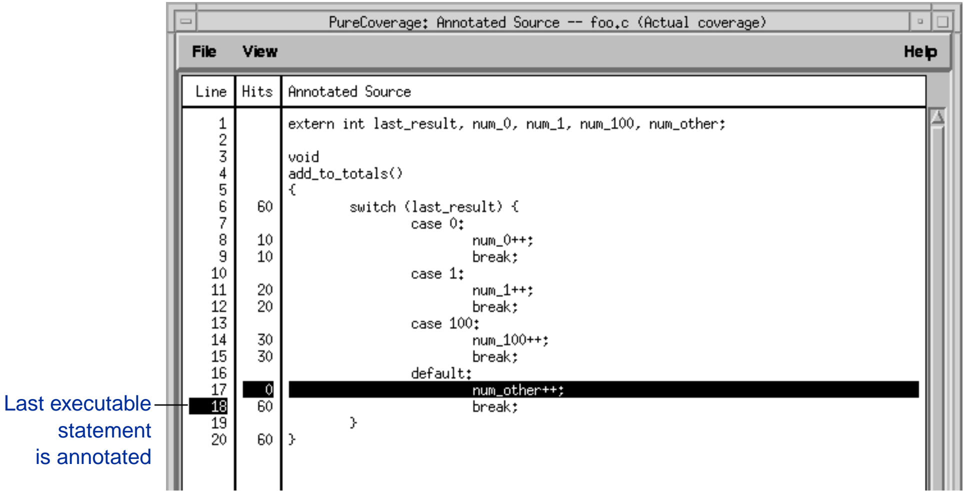
Local variable declarations typically are not annotated unless they are initialized at declaration time. This does not apply to static local variables. In this case initialization occurs during the compilation phase instead of at run-time.

An exception to this is if a compiler marks the first local variable declaration as a function entry point.

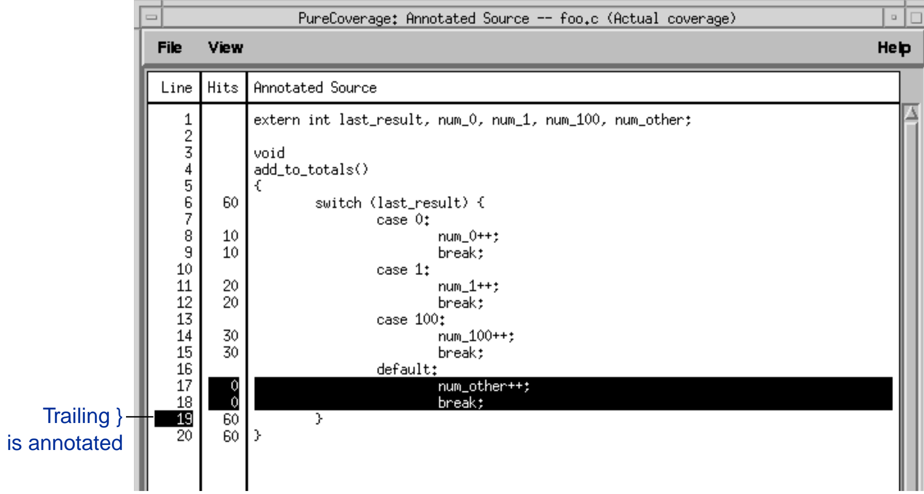
Switch statements

All compilers accumulate data at the switch keyword, but they vary on whether they accumulate data for all passes through a switch statement at the end of the switch statement. The way they accumulate data varies as well.

Some compilers accumulate data at the last executable statement of a switch statement:



Some compilers accumulate data at the trailing } of the switch statement:



Some compilers do not accumulate data at the end of the switch statement:

PureCoverage: Annotated Source -- foo.c (Actual coverage)

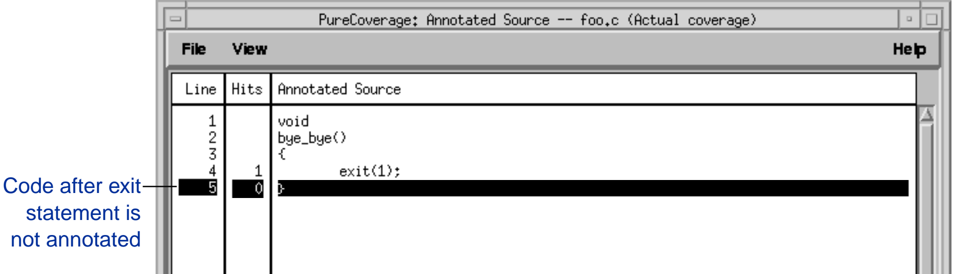
File View Help

```
Line Hits Annotated Source
   1      extern int last_result, num_0, num_1, num_100, num_other;
   2
   3      void
   4      add_to_totals()
   5   60 {
   6   60         switch (last_result) {
   7                      case 0:
   8   10                         num_0++;
   9   10                         break;
  10                      case 1:
  11   20                         num_1++;
  12   20                         break;
  13                      case 100:
  14   30                         num_100++;
  15   30                         break;
  16                      default:
  17    0                         num_other++;
  18    0                         break;
  19              }
  20   60 }
```

End of switch statement is not annotated

exit() statements

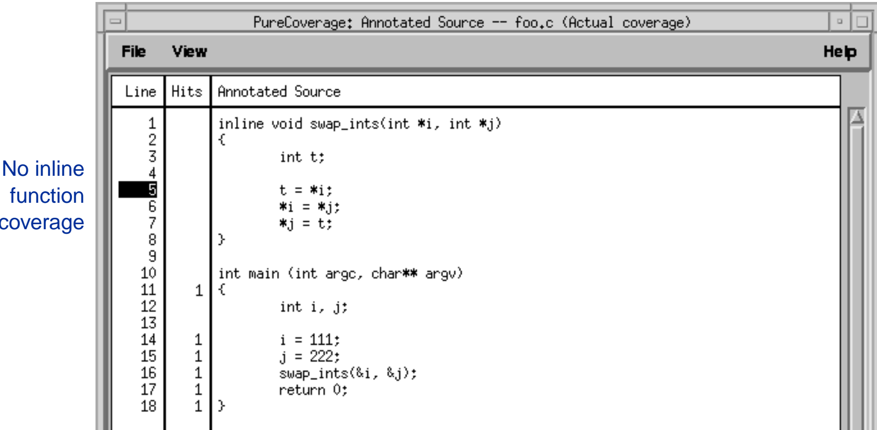
Normally, when an `exit()` statement is executed the program stops and control is passed back to the invoking process. This means that the function return is not reached and therefore does not accumulate run data:



When the `exit(XXX)` occurs in `main()`, some compilers treat it as equivalent to `return(XXX)`, so this annotation variation does not necessarily appear consistently under all circumstances.

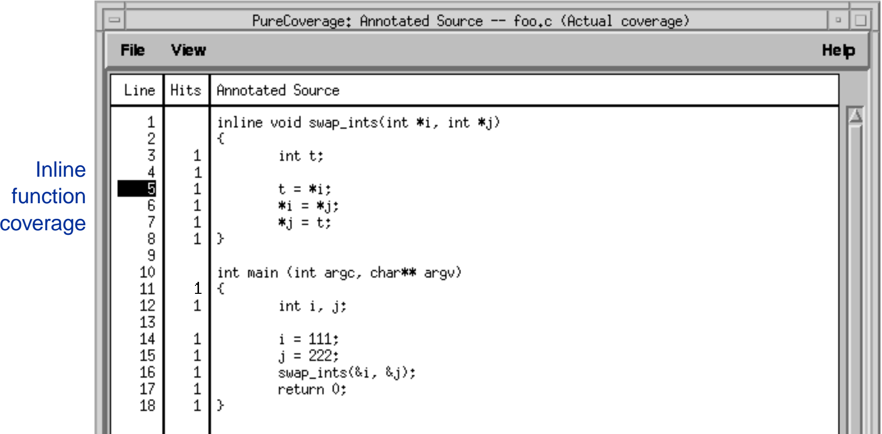
C++ inline functions

Some compilers do not record debugging information for C++ inline functions, and therefore do not annotate inline functions:



Some compilers annotate inline functions:

Inline function coverage

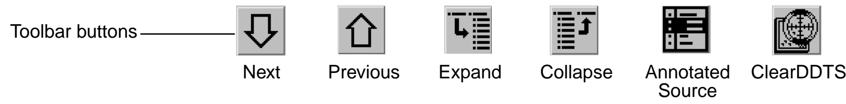


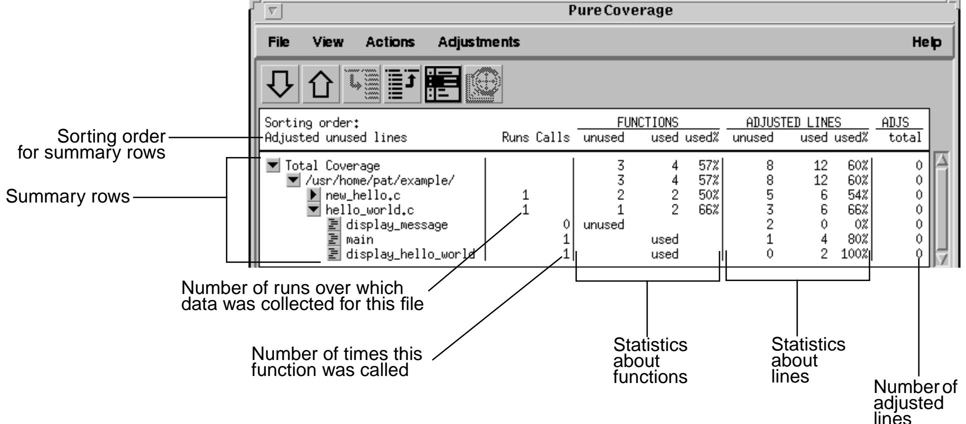
Some compilers that annotate inline functions do not generate any code for an *unreachable* inline function—one that is not used anywhere. In this case, the function does not appear in the Viewer, and no count appears next to it in the Annotated Source window.

PureCoverage 4.1 Quick Reference

PureCoverage Viewer

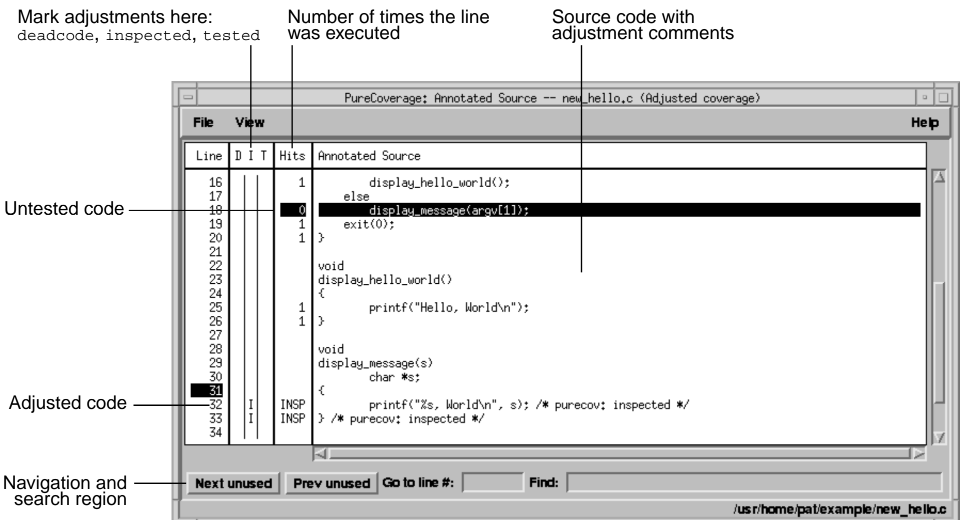
Compile a program with the command `% purecov cc -g <filename>.c`
Run the program, then use the command `% purecov -view a.out.pcv` to open the Viewer.





PureCoverage Annotated Source Window

In the Viewer, click the ▤ button next to any function name to open the Annotated Source window.



PureCoverage 4.1 Quick Reference

Build-time options

Set build-time options on the link line when you instrument programs with PureCoverage. For example:

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

Commonly used build-time options	Default
-always-use-cache-dir Forces all PureCoverage instrumented object files to be written to the global cache directory	**no**
-auto-mount-prefix Removes the prefix used by file system automounters	**/tmp_mnt**
-cache-dir Specifies the global directory where PureCoverage caches instrumented object files	**<purecovhome>/cache**
-collector Specifies the collect program to handle static constructors (for use with `gcc`, `g++`)	**none**
-ignore-run-time-environment Prevents the run-time PureCoverage environment from overriding the option values used in building the program	**no**
-linker Specifies a linker other than the system default for building the executables	**system-dependent**

Run-time options

Set run-time options on the link line or by using the PURECOVOPTIONS environment variable. For example:

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

Commonly used run-time options	Default
-counts-file Specifies an alternate file for writing coverage count data in binary format	**%v.pcv** **Note**: Can use filename conversion characters
-follow-child-processes Controls whether PureCoverage is enabled in forked child processes	**no**
-log-file Specifies a log file for PureCoverage run-time messages	**stderr** **Note**: Can use filename conversion characters
-program-name Specifies the full pathname of the PureCoverage instrumented program	**argv[0]**
-user-path Specifies a list of directories to search for source code	**none** **Note**: Can also be used in `-view` mode

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` files) obtained from different versions of the same object file	**no**

PureCoverage 4.1 Quick Reference

Analysis-time mode options

Command-line syntax: `% purecov -<mode option> [analysis-time options] <file1.pcv file2.pcv ...>`

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

Using PureCoverage with other Rational products

Product	Command line syntax
Purify	**% purify <purifyoptions> purecov <purecovoptions> cc ...**
PureLink	**% purelink <purelinkoptions> purecov <purecovoptions> cc ...**
Quantify	Cannot instrument for PureCoverage and Quantify simultaneously

Conversion characters for filenames

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

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

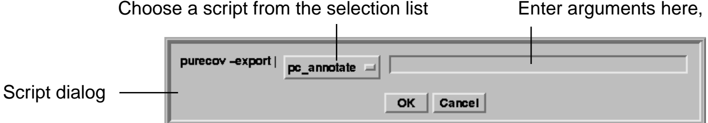
Viewer keyboard accelerators

Key	Action	Menu equivalent
Control-a	Open Annotated Source window	**Show annotated source** in the Actions menu
Down arrow or Control-n	Move to the next block of uncovered code	**Next** in the Actions menu
Up arrow or Control-p	Move to the previous block of uncovered code	**Previous** in the Actions menu
Control-o	Expand the selected row	**Expand** in the Actions menu
Control-k or DEL	Collapse the selected row	**Collapse** in the Actions menu
Space or Return	Toggle the selected row between expanded and collapsed	**Expand** in the Actions menu **Collapse** in the Actions menu

PureCoverage 4.1 Quick Reference

Report scripts

Run report scripts from the command line, or choose **Run script** from the Viewer File menu to open the script dialog.



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...]
```

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 be 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 user 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...]
```

API functions

Include <purecovhome>/purecov.h in your code and always link with <purecovhome>/purecov_stubs.a
Useful compile or link options include: -I`purecov -print-home-dir` and -L`purecov -print-home-dir`

Function	Description
int purecov_clear_data (void)	Clears and resets all coverage accumulators to zero
int purecov_disable_save (void)	Prevents coverage counts from being written when the program exits or executes another program
int purecov_enable_save (void)	Permits coverage counts to be written when the program exits or executes another program
int purecov_is_running (void)	Returns 1 if the program is PureCoverage-instrumented
int purecov_save_data (void)	Merges and writes coverage counts accumulated before this function call
int purecov_set_filename (const char *file_name)	Sets the file for writing coverage data (default is prog.pcv) **Note**: Can use filename conversion characters

Index

Symbols

%p 7-3
%V 7-3
%v 7-3
*.pcv files
  introduced 2-5
  merging 3-4
  separating 3-3
  specifying 3-3

A

accelerators, keyboard 5-2
Action menu (Viewer), opening 5-2
actual coverage data
  displaying 5-5–5-6
  explained 4-4
ADJS column (Viewer) 2-6, 5-4
adjusted coverage data
  displaying 5-5–5-6
  explained 4-4
ADJUSTED LINES columns (Viewer) 2-5, 5-4
adjustments 4-4–4-18
  adjustment file format 4-14
  adjustment files 4-7
  Adjustments menu 5-8
  Annotated Source window display 4-9
  blocks 4-6, 4-9
  comment format 4-6
  distinguishing types by color 5-11
  extracting from source files 4-7
  files 4-7
  FORTRAN limitations 4-11
  highlighted columns 4-12
  implicit end comment 4-6
  interactive 4-10
  manual 4-5
  multi-line statements 4-12
  non-nesting 4-6
  removing 4-13

adjustments, *continued*
  saving 4-12
  sorting order 4-11
  strategies for use 4-15–4-18
  unnecessary 4-12
  updating 4-8
  Viewer display 4-8
Adjustments menu (Viewer) 5-8
-always-use-cache-dir 2-12, 7-7
analysis-time mode options 7-2, 7-16–7-18
analysis-time options 7-2, 7-15–7-18
annotated differences report 6-9
annotated source code
  adjustments to 4-4
  examining 2-8
annotated source report 6-8
Annotated Source window 5-9–5-12
  adjusted display 4-9
  components 5-9
  File menu 5-10
  navigating 5-11
  removing adjustments 4-13
  saving source files 4-12
  selecting a line 4-13
  unavailable for code compiled without -g 2-14
  View menu 5-10
  viewing unused code 2-8
annotations
  C++ in-line functions C-7
  complex source lines C-1
  function entry points C-3
  local variable declarations C-5
  local variables C-4
  multi-line statements C-2
  switch statements C-5
a.out.pcv 2-5
  See also *.pcv files
API functions
  calling interactively 8-2
  data collection 8-3

API functions (by name)
purecov_clear_data 8-3
purecov_disable_save 2-17, 8-3
purecov_enable_save 8-4
purecov_is_running 8-4
purecov_save_data 8-3
purecov_set_filename 2-16, 2-17, 8-3
-apply-adjustments 7-15
arguments for scripts. entering 3-10
ASCII format, used for adjustments 4-14
automatic discarding of data 3-5
automatic reloading of changed data 2-9
automating data collection 3-1
-auto-mount-prefix 7-10

B

basic blocks
defined 2-13
example A-2
bl, export format directive B-8
block adjustments
display 4-9
marking manually 4-5–4-6
block, export format keyword B-8
Bug report (Help menu) 5-9
build differences summary report 6-6
builds, makefiles for 3-2
build-time options 7-1, 7-5, 7-7–7-9
built-in options, finding 7-6

C

C++
adjustment comment format 4-11
in-line function annotation C-7
-cache-dir 7-5, 7-7
caching
directory 2-12, 7-7
files 2-12
options 7-7
removing old files 2-12
Calls column (Viewer) 5-3
canonical pathnames 4-3
checksum, used to discard data 3-5
child processes 2-15
ClearDDTS, using with PureCoverage 3-12, 5-3
Close (Annotated Source File menu) 5-10
-collector 7-9
color, to distinguish adjustment types 5-11
columns (Viewer) 5-3–5-6
selecting 5-4
combining data for multiple runs 3-4
commands
for compiling and linking with PureCoverage vii
for displaying coverage data 2-5
shown using csh(1) syntax ix
syntax for specifying options 7-4
comments
adjustments 4-6
C++ format 4-11
export format B-3
compiler options 2-13, C-1
compiler-based annotation variations C-1–C-8
compiling with PureCoverage vii
separately from linking 2-3
complex source line annotation C-1
context help ix
control key combinations 5-2
conventions used in this guide viii
conversion characters 7-3
counters, saturating 2-18
-counts-file 2-16, 3-3, 7-3, 7-12
coverage
excluding files and directories 4-2
excluding files by source filename 4-4
excluding libraries 4-1–4-2
coverage data
adjusting 4-4–4-18
and API functions 8-3
automatic discarding 3-5
automating collection 3-1
collection options 7-12
combining for multiple runs 3-4
composite, for libraries A-3
converting to export format B-1
customizing collection 4-2–4-4
detail level 2-6
discarded 3-5–3-6, 7-15
displaying (general) 2-5–2-7

coverage data, *continued*
displaying actual coverage 5-5
displaying adjusted coverage 5-5
expanding detail level 2-6
exporting 3-8
file level 2-6
for excluded libraries A-1
for failed tests 3-6
forcing a merge 3-6
function level 2-7
graphically inspecting 2-5, 5-3
handling options 2-9
line level 2-8
location of files 7-12, 8-3
maximum counts 2-18
merged 3-4, 7-15
output files 2-5
reloading in the Viewer 5-6
removing collected data for excluded files 4-3
saving options 7-12
separating child processes 2-16
separating for individual test runs 3-3
coverage level
expanding 2-6
file level 2-6
function level 2-7
line level 2-8
coverage summary report 6-2
csh(1) syntax, used for command examples ix
custom reports
accessing in the Viewer 3-11
writing scripts for 6-15–6-17
customizing the Viewer display 5-4

D

data
See coverage data
data handling options 2-9
dbx debugger, used for debug examples ix
deadcode adjustment type
applied to all lines in a file 4-9
applied to all lines in a function 4-8
defined 4-5
effect on Viewer display 4-8
debug examples, dbx format ix
debugger, calling API functions 8-2
debugging option -g 2-13
default Viewer settings 5-8
deleting coverage data 4-3
detail level of coverage data
expanding 2-6
file level 2-6
function level 2-7
line level 2-8
di, export format directive B-5
differences report 6-5
directives
exclude 4-2
full pathname 7-3
process id 7-3
program name 7-3
directories
excluding from coverage 4-2
for custom scripts 3-11
location of cache files 2-12
PureCoverage home directory viii
specified in -user-path option 2-15
directory, export format keyword B-5
discarding data
failed tests 3-6
PureCoverage procedure 3-5
displaying
actual coverage 5-5
adjusted coverage 5-5
PureCoverage toolbar 5-3

E

edi, export format directive B-5
efi, export format directive B-6
efu, export format directive B-7
eliminating old coverage data 4-3
e-mail, automatic, for low coverage reports 6-3
end directory, export format keyword B-5
end file, export format keyword B-6
end function, export format keyword B-7
environment variable options, precedence 7-4
examples
annotated differences report 6-11
annotated source report 6-8–6-9
basic block A-2

examples, *continued*
build differences summary report 6-7
coverage summary report 6-3
custom report script 6-15–6-17
differences report 6-6
export format B-3
hello_world.c 2-1–2-11
low coverage mail report 6-4
low coverage report 6-3
makefile 2-11, 3-2
spreadsheet report 6-5
test script for discarding data automatically 3-7
using PureCoverage 2-1–2-11
exclude directives 4-2, A-4
wildcards 4-3
excluding
by filename or directory 4-2
by source filename 4-4
functions and files A-1
libraries A-1
exec
invoked by threads 2-17
using 2-16
Exit (Viewer File menu) 5-7
Exit PureCoverage (Annotated Source File menu) 5-10
exit statement annotation C-7
exit, invoked by threads 2-17
exiting PureCoverage 2-11
expanding coverage detail level 2-6
-export 3-8, 7-7, 7-17
export format
comments B-3
data fields B-4
example B-3
writing custom reports 6-15
export format keywords
block B-8
defined B-2
directory B-5
end directory B-5
end file B-6
end function B-7
file B-6
function B-7
line B-8
list of B-1
total B-4
exporting data 3-8, 7-17
-extract 7-17
in makefile 4-8
running twice on one file 4-7
Extract adjustments from all source files (Viewer Adjustments menu) 5-8
extracting adjustments 4-7

F

failed tests, discarding data 3-6–3-8
features, PureCoverage 1-3
new features ix
fi, export format directive B-6
file level coverage data 2-6
File menu
Annotated Source window 5-10
Script Output window 3-11
Viewer 5-6
file options 7-10
filename expansion 7-3
files
adjustment 4-7
adjustment file format 4-14
a.out.pcv 2-5
cache files 2-12
excluding from coverage 4-2, A-1
export format keyword B-6
location of .pcv files 7-12, 8-3
merging output files 3-4
object files 2-12
redirecting coverage output to file 7-11
saving adjustments 4-12
source 2-14
specifying output file names 3-3
-follow-child-processes 2-16, 2-17, 7-12
fonts, in User's Guide viii
-forbidden-directories 7-8
-force-merge 3-6, 7-15, A-3
-force-rebuild 4-4
fork 2-15
format
adjustment comments 4-6, 4-13
adjustment file 4-14
export B-1–B-9
FORTRAN and interactive adjustments 4-11
fu, export format directive B-7

function, export format keyword B-7
function-level coverage
example 2-7
for code compiled without -g 2-14
functions
entry point annotation C-3
excluding from coverage A-1
usage statistics 2-5
FUNCTIONS columns (Viewer) 5-4

G

-g compiler option 2-13
incompatible with -O 2-14
graphical display 5-1–5-12

H

-handle-signals 2-17, 7-13
hardware, supported by PureCoverage ix
harnesses, test
discarding data for 3-7
using PureCoverage with 3-2
header file, PureCoverage 8-1
Hello World example
accessing 2-2
code 2-2
example directory 2-6
file information line 2-6
hello_world.c 2-2
improving test coverage 2-9
program output 2-4
system library coverage 2-7
hello_world.c 2-2
Help menu 5-8
hiding the toolbar 5-2
hierarchy of options 7-4
Highlight menu items (Annotated Source View menu) 5-11
highlighting
adjustment columns (Annotated Source) 4-12
changing colors for adjustments 4-10

I

-ignore-signals 2-17, 7-13
improving test coverage, Hello World example 2-9
individual test runs, separating data from 3-3
informational options 7-2, 7-19
in-line function annotation C-7
inspected adjustment type
defined 4-5
effect on Viewer display 4-8
installation ix
instrumentation
defined 2-12
for excluded libraries A-2
instrumenting a program 2-1
partial 4-2
instrumented programs, running 2-4
interactive marking of adjustments 4-10

K

keyboard accelerators 5-2

L

li, export format directive B-8
libraries
composite coverage A-3
coverage 4-1
excluding from coverage A-1
instrumentation of excluded libraries A-2
licensing ix
line adjustments
display 4-9
marking interactively 4-11
marking manually 4-5
line usage statistics 2-5
line, export format keyword B-8
line-level coverage data 2-8
linker
default 7-9
options 7-9
linking with PureCoverage vii
separately from compiling 2-3
link-line command 7-5
lint 4-14
local variable declaration annotation C-4
-log-file 7-3, 7-11

low coverage mail report 6-3
low coverage report 6-3

M

mailing reports, automatically 6-3
Make toolbar visible (Viewer View menu) 5-7
makefile example 2-11
marking adjustments
  interactively 4-10
  manually 4-5
maximum coverage counts 2-18
menus
  Annotated Source window 5-10–5-11
  Script Output window 3-11
  Viewer 5-6–5-9
-merge 7-7, 7-18, A-3
merging files
  forcing 3-6
  with -view option 3-4
merging options 7-15
minimum instrumentation A-2
mode options
  listed 7-2
  using 7-7
modifying makefiles 2-11
multi-line statements
  annotations C-2
  marking adjustments
    manually 4-12
multiple processes, covering 2-15
multiple program runs, combining data 3-4
multi-threaded applications 2-17

N

names
  cache files 2-12
  source filenames for excluding coverage 4-4
  specifying output file names 3-3
  unqualified file names 4-3
navigating (Annotated Source window) 5-11
nightly builds, using PureCoverage in 3-1

O

-O compiler option 2-14
Object Code Insertion (OCI) 2-12
object files, handling by PureCoverage 2-12
On Context (Help menu) 5-8
On license (Help menu) 5-9
On version (Help menu) 5-9
online Help ix, 5-8
Open (Viewer File menu) 5-6
options
  analysis-time 7-15
  build-time 7-4, 7-7
  compatibility with
    modes 7-17–7-18
  default values 7-2
  environment variable 7-4
  for handling coverage data 2-9
  for Purify and PureCoverage 7-5
  informational 7-19
  link-line 7-5
  listing built-in options 7-6
  mode 7-16
  processing 7-4
  reference tables 7-1
  run-time 7-4
  setting site-wide 7-4
  syntax 7-2
options (by name)
  -always-use-cache-dir 2-12, 7-7
  -apply-adjustments 7-15
  -auto-mount-prefix 7-10
  -cache-dir 7-5, 7-7
  -collector 7-9
  -counts-file 2-16, 3-3, 7-12
  -export 7-7, 7-17
  -extract 7-17
    in makefile 4-8
    running twice on one file 4-7
  -follow-child-processes 2-16, 2-17, 7-12
  -forbidden-directories 7-8
  -force-merge 3-6, 7-15, A-3
  -force-rebuild 4-4
  -handle-signals 2-17, 7-13
  -ignore-signals 2-17, 7-13
  -linker 7-9
  -log-file 2-4, 7-3, 7-11
  -merge 7-7, 7-18
  -print-home-dir 7-19

options (by name), *continued*
-program-name 7-11
-run-at-exit 3-7, 7-14
-user-path 2-14, 7-10
-version 7-19
-view 2-5, 7-7, 7-18
output (.pcv) files
defined 2-5
merging 3-4
separating 3-3
specifying 3-3
output from instrumented programs 2-4
overhead, for PureCoverage A-4
overriding the run-time environment 7-6

P

parent process counts, separating from child process counts 2-16
partial instrumentation 4-2
pathnames, canonical 4-3
pc_annotate 6-8
pc_below 6-3
pc_build_diff 6-6
pc_covdiff 6-9
pc_diff 6-5
pc_email 6-3
pc_select 6-11
pc_ssheet 6-4
pc_summary 6-2
.pcv files
defined 2-5
merging 3-4
separating 3-3
specifying 3-3
Perl
included with PureCoverage 6-2
sample custom report 6-16
precedence of options 7-4
printf function, excluded 2-7
-print-home-dir 7-19
-program-name 7-11
pure_remove_old_files 2-12, 2-13, 7-7
purecov_clear_data 8-3
purecov_disable_save 2-17, 8-3
purecov_enable_save 8-4
purecov_is_running 8-4
purecov_save_data 8-3
purecov_set_filename 2-16, 2-17, 8-3
purecov_stubs.a 8-1
purecov_what_options 7-6
.purecov.adjust 4-7
PureCoverage
and ClearDDTS 3-12
and PureLink 3-12
and Purify 3-11, 7-5
and Quantify 3-11
API 8-1–8-4
benefits provided 1-1
cache files 2-12
discarding data 3-5
ease of use 1-2
files created by 2-12
functional description 2-12
header file 8-1
installation ix
key features 1-3
modifying makefiles 2-11, 3-2
option processing 7-4
options 7-1–7-19
overhead A-4
release notes ix
report scripts 6-1–6-17
starting vii
support questions x
tutorial 2-1–2-11
using with nightly builds 3-1
using with test harnesses 3-2
PureCoverage overview (Help menu) 5-9
PureCoverage Viewer
See Viewer
purecov.h 8-1
purecovhome viii
PURECOVOPTIONS 7-4, 7-6
.purecov.Xdefaults 5-8
changing adjustment colors 5-11
changing comment style 4-11
PureLink, using with PureCoverage 3-12
PUREOPTIONS 7-4, 7-5, 7-6
Purify, using with PureCoverage 3-11, 7-5, A-4

Q

Quantify, separate instrumentation 3-11

R

Rational Software Corporation, contacting x
README file ix, 5-9
redirecting output 2-4
Refresh display (Viewer View menu) 5-7
release notes, displaying ix
Reload all (Viewer File menu) 5-6
reloading changed data 2-9
removing
  adjustments 4-13
  coverage data 4-3
report scripts 6-1–6-17
  directory location 6-2
  entering arguments 3-10
  running 3-9–3-11
  writing 6-15
report scripts (by name)
  pc_annotate 6-8
  pc_below 6-3
  pc_build_diff 6-6
  pc_covdiff 6-9
  pc_diff 6-5
  pc_email 6-3
  pc_select 6-11
  pc_ssheet 6-4
  pc_summary 6-2
reports
  annotated differences 6-9
  annotated source 6-8
  automating distribution 3-1
  build differences summary 6-6
  coverage summary 6-2
  custom 6-15
  differences 6-5
  low coverage 6-3
  low coverage mail 6-3
  selected tests 6-11
  spreadsheet 6-4
  writing scripts 6-15
Run script (Viewer File menu) 5-6
-run-at-exit 3-7, 7-14
running instrumented programs 2-4
running report scripts 3-9–3-11
Runs column (Viewer) 5-3
  different counts for each file 2-7
runs, multiple, combining data 3-4
run-time environment, ignoring 7-6
run-time options 7-1, 7-10–7-14

S

saturating counters 2-18
Save (Annotated Source File menu) 5-10
Save as (Viewer File menu) 5-6
Save source and annotations as (Annotated Source File menu) 5-10
saving files with adjustments 4-12
Script dialog 3-10
Script Output window 3-10
scripts
  See report scripts
searching, Annotated Source window 5-12
Select columns (Viewer View menu) 5-7
Select sorting order (Viewer View menu) 5-7
selected tests report 6-11
selecting items (Viewer) 5-1
Set display style for names (Viewer View menu) 5-7
setting site-wide options 7-4
shared library coverage 4-1
Show adjusted annotated source (Annotated Source View menu) 5-10
Show line counts (Annotated Source View menu) 5-11
Show line numbers (Annotated Source View menu) 5-11
signal handling 2-17, 7-13
Snapshot (Annotated Source File menu) 5-10
software, supported by PureCoverage ix
sorting order (Viewer) 4-11, 5-7
source file, finding path to 2-14
source filenames, for excluding coverage 4-4
source files, extracting adjustments 4-7
spreadsheet report 6-4
Start ClearDDTS (Viewer File menu) 5-7
starting PureCoverage vii
statistics
  See coverage data

strings, finding in the Annotated Source window 5-12
stubs library 8-1
support, technical x
supported hardware and software ix
switch statement annotation C-5
symbolic links, not used in exclude directives 4-3
syntax, PureCoverage options 7-2
system library coverage 4-2

T

targets, in makefiles 2-11
technical support, contacting x
test harnesses
  discarding data from 3-7
  using PureCoverage with 3-2
test programs, modified to discard data 3-8
test script, example 3-7
tested adjustment type
  defined 4-5
  effect on Viewer display 4-8
tests
  discarding data for failed tests 3-6–3-8
  selecting a subset 6-11
third-party library coverage 4-1
timestamps and merging data 3-6
to, export format directive B-4
toolbar 5-2
Total Coverage row 2-5
total, export format keyword B-4
tutorial 2-1–2-11

U

unnecessary adjustments 4-12
unqualified file names, for excluding files and libraries 4-3
Update ~/.purecov.Xdefaults (Viewer View menu) 5-8
updating adjustments 4-8
-user-path 2-15, 7-10

V

variant annotations C-1–C-8
-version 7-19
Vertical column separators (Viewer View menu) 5-7
vfork 2-17
-view 2-5, 7-18
View menu
  Annotated Source window 5-10
  Viewer 5-7
Viewer 5-1–5-9
  adjusted display 4-8
  Adjusted Lines columns 5-4
  Adjustments menu 5-8
  Calls column 2-7
  default settings 5-8
  displaying 2-5
  effect of adjustment types 4-8
  File menu 5-6
  file-level detail 2-6
  Function columns 5-4
  function-level detail 2-7
  Functions columns 2-7
  optional columns 5-4
  Runs column 2-7
  toolbar 5-2
  View menu 5-7
  window components 5-1

W

web site, Rational Software Corporation x (front matter)
wildcards
  Annotated Source window 5-12
  exclude directives 4-3
windows
  See Annotated Source window, Script Output window, Viewer
World Wide Web site, Rational Software Corporation x (front matter)
Write export file (Viewer File menu) 5-6
Write .pcv file (Viewer File menu) 5-6
Write view settings to (Viewer View menu) 5-8

X

X defaults 5-8