Installing and Getting Started

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

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

Rational®
the e-development company™

IMPORTANT NOTICE

COPYRIGHT NOTICE

Copyright © 2000 Rational Software Corporation. All rights reserved.

THIS DOCUMENT IS PROTECTED BY COPYRIGHT AND CONTAINS INFORMATION PROPRIETARY TO RATIONAL. ANY COPYING, ADAPTATION, DISTRIBUTION, OR PUBLIC DISPLAY OF THIS DOCUMENT WITHOUT THE EXPRESS WRITTEN CONSENT OF RATIONAL IS STRICTLY PROHIBITED. THE RECEIPT OR POSSESSION OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHTS TO REPRODUCE OR DISTRIBUTE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING THAT IT MAY DESCRIBE, IN WHOLE OR IN PART, WITHOUT THE SPECIFIC WRITTEN CONSENT OF RATIONAL.

U.S. GOVERNMENT RIGHTS NOTICE

U.S. GOVERNMENT RIGHTS. Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Rational License Agreement and 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.

TRADEMARK NOTICE

Rational, the Rational logo, Purify, PureCoverage, Quantify, and ClearQuest, are trademarks or registered trademarks of Rational Software Corporation in the United States and in other countries.

All other names are used for identification purposes only and are trademarks or registered trademarks of their respective companies.

U.S. PATENT NOTICE

U.S. Registered Patent Nos. 5,193,180 and 5,335,344 and 5,535,329. Licensed under Sun Microsystems Inc.'s U.S. Pat. No. 5,404,499. Other U.S. and foreign patents pending.

Part Number: 800-023699-000

Printed in the U.S.A.

Contents

Preface

Other resources 7
Contacting Rational technical publications 8
Contacting Rational technical support 8

1 Installing the products

What you need before starting 9
Installing the products: rs_install 13
  Answers to questions about rs_install 14
Installing the products: Post-installation 15
Maintaining the rational.opt options file 19
  Modifying the list of user IDs 20
Removing a previous product release 20
Requesting and installing the permanent license key 21
  Requesting your permanent license key 21
  Entering a permanent license key after initial installation 21
Supplemental notes: Creating an installation directory manually . . 22
Supplemental notes: Mounting the CD-ROM 23
Supplemental notes: Ejecting the CD-ROM 25
Supplemental notes: Using rs_install commands 25
Supplemental notes: Using the FLEXlm License Manager 26
  The Rational license file 26
  Verifying that FLEXlm is working 27
  Using FLEXlm commands 27
  Learning more about FLEXlm 28

2 Using Purify

Finding errors in Hello World 30
Instrumenting a program 31
Compiling and linking in separate stages 31
Running the instrumented program 32
Seeing all your errors at a glance 33
Finding and correcting errors 34
Understanding the cause of the error 35
Correcting the ABR error 36
Finding leaked memory 37
Correcting the MLK error 38
Looking at the heap analysis 39
Comparing program runs 40
Suppressing Purify messages 41
Saving Purify output to a view file 42
Saving a run to a view file from the Viewer 42
Opening a view file 42
Using your debugger with Purify 43
Using Purify with PureCoverage 43
Purify API functions 44
Build-time options 45
Conversion characters for filenames 45
Run-time options 46
Purify messages 47
How Purify finds memory-access errors 48
How Purify checks statically allocated memory 50

3 Using PureCoverage

Finding untested areas of Hello World 52
Instrumenting a program 53
Running the instrumented program 54
Displaying coverage data 55
Expanding the file-level detail 56

Examining function-level detail 57
Examining the annotated source 58
Improving Hello World’s test coverage 59
Using report scripts 61
Build-time options 62
Run-time options 62
Analysis-time options 63
Analysis-time mode options 63

4 Using Quantify
How Quantify works 66
Building and running an instrumented program 67
Interpreting the program summary 68
Using Quantify’s data analysis windows .69
The Function List window 70
Sorting the function list 70
Restricting functions 71
The Call Graph window 72
Using the pop-up menu 73
Expanding and collapsing descendants .73
The Function Detail window 74
Changing the scale and precision of data 75
Saving function detail data 75
The Annotated Source window 76
Changing annotations 77
Saving performance data on exit 77
Comparing program runs with qxdiff 78
Build-time options 79
qv run-time options 79
Run-time options 80
API functions 81

Index 83

Preface

This Getting Started guide is designed to help you get up and running quickly with Rational® Purify®, PureCoverage®, and Quantify®. It includes information about:

- Installing the products
- Using Purify to pinpoint run-time errors and memory leaks everywhere in your application code
- Using PureCoverage to prevent untested application code from reaching end users
- Using Quantify to improve the performance of your applications by finding and eliminating bottlenecks

Purify, PureCoverage, and Quantify—the essential tools for delivering high-performance UNIX applications—use patented Object Code Insertion (OCI) technology to instrument your program, inserting instructions into the program's object code. This enables you to check your entire program, including third-party code and shared libraries, even when you don't have the source code.

Note: Starting to use Purify, PureCoverage, and Quantify is as easy as adding the product name (`purify`, `purecov`, or `quantify`) to the front of your link command line. For example:

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

Other resources

- `README.licensing` in the product directory contains the latest licensing information.

- Online Help is available for each application through the Help menu. To get Help on a specific item in a window, select **Help > On Context**.
- For complete product information, see the Purify, PureCoverage, and Quantify user's guides.
- For information about Rational Software and Rational Software products, go to `http://www.rational.com`.

Contacting Rational technical publications

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

Contacting Rational technical support

You can contact Rational technical support by e-mail at `support@rational.com`.

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

1 Installing the products

This chapter tells you how to use the `rs_install` program to install Rational Purify, PureCoverage, and Quantify. It also explains how to perform installation-related tasks outside of `rs_install`, if any are necessary. The chapter also contains information about post-installation tasks (such as uninstalling) and administering the GLOBEtrotter FLEXlm® Software License Manager that is included with your Rational Software product.

What you need before starting

You will need all the information in the following table to install your Rational product.

Data	Notes	Your Entry
The full pathname to the installation location (referred to in this chapter as `Rational`).	This is the directory where you install all Rational Software products. You must have **15 megabytes** of free disk space for each installation of Purify, Quantify, and PureCoverage. The directory must be accessible from every machine on which you plan to run the Rational products—both the machines on which users *instrument* their applications, and the machines on which users *run* their applications. It must be the same for each machine, so you cannot use a local automount path like `/tmp_mnt/rational`. If `Rational` does not already exist, the installation program will create it when you enter the full pathname. If you are installing on a read-only file system, or if you want to create this directory manually, see “Supplemental notes: Creating an installation directory manually” on page 22. This section also shows you the structure of the directory after installation.	

Data	Notes	Your Entry
Rational account number.	Source: your Rational license key certificate.	
Contact information for the person you want to receive license keys from Rational Software.	Name and email address are required.	
Contact information for the person who will be responsible for renewing the license.	If different from the previous entry.	
Host name or IP address of the host machine on which the license server is to run.	If this machine, the license server host, is different from the installation machine, you must have remote shell access from the installation machine to the license server host. In addition, the installation directory must be accessible from the license server host.	
License server port number.	This is the port at which the license server listens for license requests. Default is 27000. You can use any port number that is not already in use. The /etc/services file on the license host lists all ports in use by most commonly used services, but other ports may be in use on your system as well. FLEXlm reserves ports 27000–27004 for its use; these ports are ordinarily available unless a different FLEXlm server on the license host is using them. The `rs_install` program checks to make sure that the license server port number does not conflict with entries in the /etc/services file on the license server host, or with NIS services.	

Data	Notes	Your Entry
License key type.	Source: your Rational license key certificate or email from Rational Software **p = permanent**. Enter “p” if you already have your permanent license key, or if you have email access from the machine you are using for the installation. The installation procedure allows you to request your permanent key by email, and you should receive it by return email within a few minutes. For additional information, see “Requesting and installing the permanent license key” on page 21. **s = startup.** Rational provides a startup license key to get you up and running as soon as you receive your Rational product, but you don’t have to use it if you have a permanent key or are willing to wait until you receive a permanent key. If you use the startup key, you will have to enter a permanent key later. **e = evaluation**. Evaluation license keys are valid for a limited time. You will have to enter a permanent key when the evaluation key expires to ensure continued use of your Rational product. **t = term license agreement (TLA)**. TLA keys allow the use of the rational product for a specific period of time. **Note:** To enter the permanent license key after you’ve been using a startup or evaluation license, see “Entering a permanent license key after initial installation” on page 21.	
License quantity.	Source: your Rational license key certificate. Enter “0” for “uncounted” if you have an evaluation license.	
Expiration date.	If you nave a permanent license, enter `permanent`. Source for other license types: your Rational license key certificate or email from Rational Software. If you have a startup or evaluation license, enter the date in the dd-mmm-yyyy format. (The field is not case sensitive.)	

Data	Notes	Your Entry
Note: If you are installing a permanent license, you must supply user IDs for each individual who will be using the product. **You must include the user ID you are using to perform the installation**; otherwise the post-installation step (which runs a simple test case to verify the installation) will fail. User IDs are recorded in the FLEXlm options file, `rational.opt`. For information about the options file, see “Maintaining the rational.opt options file” on page 19. To input User IDs, you need the data in **A** or **B** below, or the data in **A** supplemented with the data in **B**.		
A. Path of the PureLA directory containing the file `users.purela` (available only if you licensed an earlier version of the product using PureLA License Advisor).	If you are currently running the product under a PureLA license, you have the option of importing the IDs from the PureLA database instead of entering them manually. The PureLA directory is located in the same parent directory as the previous product installation, which you can find with the command `<product> -printhomedir`. You can modify the list of imported user IDs, either while you’re running `rs_install` or afterwards. If the number of user IDs is not the same as the number of licenses you bought, `rs_install` will help you correct the list.	
B. User IDs (all IDs; or some or none, in combination with an option to generate dummy names).	You can enter all user IDs. The number of IDs you enter must match the number of licenses you purchased. You can enter some user IDs, and then enter `-n` to populate the rest of the options file with dummy names as placeholders that you can replace later. Or you can just enter `-n` to enter nothing but dummy names, and update the options file later.	
License file (`<license server host name>.dat`), including full pathname.	The `rs_install` program will suggest a default. If you want to use an existing license file, enter its name, including full pathname, instead of the default. The `rs_install` program makes a backup of the existing license file before it processes the file. For information, see “The Rational license file” on page 26.	
License keys for each product you are installing OR full pathname of the license file you received by email from Rational Software.	If you do not already have license keys, the installation program will help you request them from Rational Software. If you request them by email, you can expect to receive your license file by return email within minutes. If you do not have the license keys or a license file, you can enter this data later. See “Entering a permanent license key after initial installation” on page 21.	

Installing the products: rs_install

For information about specific product and operating system versions, see the `README` file in the `DeveloperTools.<version>` directory. For information about `rs_install`, run `rs_help`, which opens a Help file in an Adobe Acrobat viewer.

To install the products:

1 Make the product available for installation.

If you are installing the product from the Rational Software product CD-ROM and need instructions, see "Supplemental notes: Mounting the CD-ROM" on page 23.

2 Run the `rs_install` program

The `rs_install` program is a complete installer that guides you through the following processes:

- Setting up the license server
- Installing product licenses
- Installing the selected product
- Performing the post-installation tasks

To run the `rs_install` program, go to the directory where you mounted the CD-ROM. (You should not be `root` when you run `rs_install`.) For example:

```
# exit
% cd /cdrom
% ./rs_install
```

The `rs_install` program prompts you through the installation, providing detailed instructions along with default settings. The defaults appear in brackets, for example [2]. To accept the default, press **Enter**.

Note: After you install your license key, the `rs_install` program reminds you that you must configure your server to automatically

restart the license server when it reboots. The `rs_install` program gives you instructions for doing this.

3 When installation is complete, go to “Installing the products: Post-installation” on page 15 and perform any necessary post-installation procedure.

Answers to questions about rs_install

Below are the answers to some common questions about the `rs_install` program.

- **Can I rerun parts of the installation?** Yes. The `rs_install` program provides commands that enable you to rerun specific sections of the installation as needed. See “Supplemental notes: Using rs_install commands” on page 25.

- **Do I have to reenter my license server information each time I install a product?** No. You only need to enter this information once. The `rs_install` program saves the information you enter about yourself and about the machine to be used as the license server for your Rational Software product licenses in two text files: an `rs_install.defaults` file that contains information about you and your license server, and a file such as `rs_install.DeveloperTools.5.2` that records product-specific information. The `rs_install` program reports the location of these files when you quit the program. The next time you run `rs_install`, the program uses the saved configuration information.

- **Do I need to install all my licenses on one server?** No. You are not required to use all of your allowed licenses for a single license server. You might want to install a product at another site and configure a license server at that site to serve the remaining licenses in your Rational Software account.

- **Which type of product license key should I install?** If you already have your permanent license key, you can install it right away. You can also request a permanent license key by email

during installation. Otherwise, select the startup or evaluation license to get started using the product.

Note: To ensure uninterrupted use of your Rational Software product, you should install your permanent license key as soon as possible. You can request your permanent license key directly from the `rs_install` program. See "Requesting your permanent license key" on page 21.

- **Can I import existing users IDs from an earlier installation of the product?** Yes. If you installed the product previously under FLEXlm, the user IDs are imported automatically when you run `rs_install`. If you installed the product under PureLA License Administrator, `rs_install` asks you if you want to import the existing `users.purela` file, and also permits you to edit the imported user IDs. You can also edit the user IDs after installation; see "Maintaining the rational.opt options file" on page 19.

Installing the products: Post-installation

The post-installation tasks depend on the individual products. The `rs_install` program performs these tasks for you if possible, or tells you how to perform them from outside of the installation program. Post-installation tasks can include:

- Installing on a read-only file system
- Making the manual pages available
- Making the products available to all users

Note: You can rerun the post-installation at any time. See "Supplemental notes: Using rs_install commands" on page 25.

Installing on a read-only file system

Purify, PureCoverage, and Quantify work by creating and monitoring special instrumented versions of object files and libraries. They must be able to write these instrumented

files to a cache directory, which by default is `Rational/releases/<producthome>/cache`.

For this reason, if you install any of the products on a file system that is mounted read-only by client machines, you must create symbolic links to a writable file system. The `rs_install` program guides you through the process of selecting a shared directory that is mounted read/write on client machines and linking the `cache` directory to this publicly writable directory.

If there is no writable shared directory mounted on client machines, have all users make a `cache` subdirectory in their home directory and set the product's `-cache-dir` option to this directory. For example:

```
% mkdir $HOME/cache
% echo $PUREOPTIONS
```

If the `PUREOPTIONS` environment variable is already set, have users specify the `-cache-dir` option:

```
csh % setenv PUREOPTIONS "-cache-dir=$HOME/cache $PUREOPTIONS"

sh, ksh $ PUREOPTIONS="-cache-dir=$HOME/cache $PUREOPTIONS"; \
         export PUREOPTIONS
```

If the `PUREOPTIONS` environment variable is *not* set, have users specify:

```
csh % setenv PUREOPTIONS "-cache-dir=$HOME/cache"

sh, ksh $ PUREOPTIONS="-cache-dir=$HOME/cache"; export \
         PUREOPTIONS
```

Have all users add this same specification to their local or central `.cshrc` file, or its equivalent.

Making the manual pages available

The `rs_install` program installs the product manual pages in `Rational/releases/<producthome>/man`. To make them available, do one of the following:

- Set your `MANPATH` environment variable to include `Rational/releases/<producthome>/man`.

- Copy the manual pages for the product into your `man` directory. If necessary, log in as `root` to do this.

Making the products available to all users

Note: Users must be listed in the `rational.opt` file in order to use Purify, PureCoverage, and Quantify; to add users to the options file, see “Maintaining the rational.opt options file” on page 19.

To make the products available to all users listed in `rational.opt`, add the full `Rational/releases/<producthome>` pathname to each user’s `PATH` environment variable, or specify the full pathname in makefiles.

As an alternative to modifying your `PATH` environment variable, you can, create a symbolic link to `<producthome>/<product>` from a directory such as `/usr/local/bin`. Make sure this is a symbolic link, not a copy or a hard link. Create symbolic links for each product you install, as in the following examples:

- For Purify:

```
% rm /usr/local/bin/purify
% ln -s Rational/releases/\
     <producthome>/purify /usr/local/bin
```

- For PureCoverage:

```
% rm /usr/local/bin/purecov
% ln -s Rational/releases/\
     <producthome>/purecov /usr/local/bin
```

  For PureCoverage, you also need to create symbolic links to the `pc_*` script files:

```
% rm -i /usr/local/bin/pc_*
% ln -s Rational/releases/\
     <purecovhome>/scripts/pc_* /usr/local/bin
```

  For more information on the `pc_*` scripts, see the *PureCoverage User’s Guide*.

- For Quantify:

```
% rm /usr/local/bin/quantify
% ln -s Rational/releases/\
     <producthome>/quantify /usr/local/bin
```

For Quantify, you also need to create symbolic links to the `qv` program and to the `qx` script files:

```
% rm /usr/local/bin/qv
% rm -i /usr/local/bin/qx*
% ln -s Rational/releases/\
     <quantifyhome>/qv /usr/local/bin
% ln -s Rational/releases/\
     <quantifyhome>/qx* /usr/local/bin
```

For more information on the `qv` program and on the `qx` scripts, see the *Quantify User's Guide*.

HPUX

- Create symbolic links for debugger scripts on HP-UX:

On HP-UX, Purify, PureCoverage, and Quantify include three scripts that enable you to start instrumented programs under a debugger. You need to create symbolic links to these scripts. For example, for Purify:

```
% rm /usr/local/bin/purify_dde
% rm /usr/local/bin/purify_xdb
% rm /usr/local/bin/purify_softdebug

% ln -s <purifyhome>/purify_dde /usr/local/bin
% ln -s <purifyhome>/purify_xdb /usr/local/bin
% ln -s <purifyhome>/purify_softdebug /usr/local/bin
```

For PureCoverage and Quantify, create the same symbolic links, substituting `purecov` or `quantify` for `purify`.

The installation is now complete. To add names to the options file, see "Maintaining the rational.opt options file" on page 19. To remove previous versions of the products, see "Removing a previous product release" on page 20.

Maintaining the rational.opt options file

Purify, PureCoverage, and Quantify use *named-user* licensing. This means that the user IDs of all users who are authorized to run Purify, PureCoverage, and Quantify must be listed in the `rational.opt` options file. The number of users IDs in the file must match the number of licenses you have installed.

Users who are identified in the file can use all features of the product, including instrumenting applications, running instrumented applications, and viewing saved data files in the product's user interface. A user can run as many concurrent sessions as desired on a single host machine; this consumes a single license. The same user can run the product on additional host machines, but consumes another license for each additional machine.

The options file is created when you run the `rs_install` program. By default, this file is `Rational/config/rational.opt`, but you can choose any location you like. You can also relocate the file yourself after installation, provided that you edit the license file `DAEMON` line to specify the new path:

```
DAEMON rational /etc/rational /mydir/rational.opt
```

During installation, `rs_install` asks you to supply user IDs, one for each license you purchased. You don't have to enter all user IDs during installation; `rs_install` will generate dummy names to bring the total up to the number of licenses you purchased. Your entries—real names, automatically generated dummy names, or both—are recorded in the options file.

The user IDs are recorded in the options file in `GROUP` directives. An `INCLUDE` directive follows each `GROUP` directive. specifying one product that the users in the group are authorized to use:

```
GROUP <group name> <user1> <user2> . . . <usern>
INCLUDE <product>:KEY=<license key> GROUP <group name>
```

For example, in the following, alice, tom, and harry can use Purify, but only alice and harry can use Quantify:

```
GROUP DevTools1 alice tom harry
INCLUDE purify:KEY=123456778982 GROUP DevTools1
GROUP DevTools2 alice harry
INCLUDE quantify:KEY=12345778982 GROUP DevTools2
```

Modifying the list of user IDs

You can add, change, or delete user IDs by running the `options_setup` script. You can also add, change, or delete user IDs in the options file using any text editor.

The number of users listed for each product must always match the number of licenses that you purchased. The license server must be restarted before the changes can take effect; the `options_setup` script restarts the license server for you.

For additional information about the options file, refer to your FLEXlm user's manual.

Note: If you modify the options file while the license vendor daemon is running, you must restart the license server.

Removing a previous product release

Note: Only the installer of the product can uninstall it.

After you install the latest version of Purify, PureCoverage, or Quantify, and after all users have switched to the new version, you can remove the old release to reclaim disk space.

To remove a previous release of Purify, PureCoverage, or Quantify, go to the `Rational` directory and run the `uninstall` script:

```
% cd Rational
% config/uninstall
```

Running the `uninstall` script with no command-line arguments causes it to display the list of products in the `releases` directory. The script prompts you for the product you want to remove.

Requesting and installing the permanent license key

When you purchase Purify, PureCoverage, or Quantify, you purchase a specific number of licenses for each product. Rational Software issues you a license key for the product that corresponds to the type and number of licenses you purchased. You need this license key to use the software.

Purify, PureCoverage, and Quantify come with a startup license that you can use to get started using the product. You then request and install a permanent license key to ensure continued use of the product. The startup license key and other licensing information is available from the Startup License Key Certificate included in the product packaging.

Purify, PureCoverage, and Quantify use the FLEXlm Software License Manager from GLOBEtrotter Software, Inc. to manage product licenses. For more information on FLEXlm, see "Supplemental notes: Using the FLEXlm Software License Manager" on page 26.

Requesting your permanent license key

If you have email access to the Internet from the machine where you are running `rs_install`, you can automatically send email to Rational Software to request your permanent license key.

Rational uses your account number, product selection, and the number of licenses you select to issue your license keys. You can also save this information to a file which you can print and use to telephone, fax, or mail Rational Software to request your license keys. The `rs_install` program provides the Rational Software telephone and fax numbers and mailing address.

Entering a permanent license key after initial installation

To enter your permanent license key after you have installed your Rational Software product and exited the `rs_install` program:

1. Go to the `Rational/releases/DeveloperTools.<version>` directory and run the `license_setup` program. For instructions, see "Supplemental notes: Using rs_install commands" on page 25.

2. For the license type, select `Permanent`.

   Note: The program tells you how to update your license server machine so that it restarts the license server when it reboots. You need root permission to perform the update.

Supplemental notes: Creating an installation directory manually

You need a publicly readable directory for the installation of Purify, PureCoverage, and Quantify. If one does not already exist, you can create it when you run `rs_install`. You can also create it manually before you start `rs_install`.

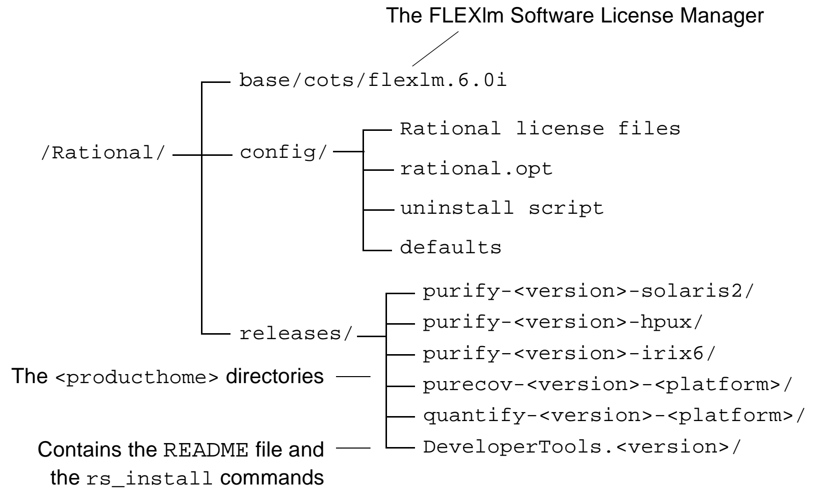
1. Log into a UNIX workstation that provides access to the CD-ROM drive and that mounts the file system(s) into which you want to load the products.

2. Create a `Rational` directory. For example:

   ```
   % mkdir /opt/Rational
   ```

   The `Rational` directory must be visible on all machines that are to run this product. The NFS name for `Rational` must be the same on all machines. (If you are installing the product for your use only, you can install it in your home directory.)

After the installation, the `Rational` directory is structured like this:



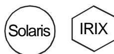
Note: Purify, PureCoverage, and Quantify must be able to write instrumented files to a `cache` subdirectory of the `<producthome>` directory. If you install on a read-only file system, you must create symbolic links to a writable file system. See "Installing on a read-only file system" on page 15.

Supplemental notes: Mounting the CD-ROM

The following instructions refer to specific operating systems. To determine your operating system, type:

```
% uname -a
```

Note: Before you begin, make sure you know the device name of your CD-ROM drive. If you do not know the device name, consult your system administrator.



On Solaris and IRIX systems with Volume Management, load the CD-ROM and then go to step 5. (On these systems, the CD-ROM automatically mounts on the `/cdrom` directory. To determine whether you have Volume Management, check to see if the Solaris

vold daemon or the IRIX mediad daemon is running on your system.)

To mount the CD-ROM:

1 Load the CD-ROM into the drive.

2 Log in as root:

```
% su root
```

3 If you do not already have one, create a cdrom directory to be the mount point for the CD-ROM drive:

```
# mkdir /cdrom
```

4 Mount the CD-ROM:

On Solaris systems without Volume Management:

```
# /etc/mount -r -F hsfs <cdrom-device-name> /cdrom
```

If your HP-UX system is configured to mount the CD-ROM at /cdrom:

```
# /etc/mount /cdrom
```

If your HP-UX system is not configured to mount the CD-ROM at /cdrom, use one of the following commands:

On HP-UX 9.x:

```
# /etc/mount -r -t cdfs <cdrom-device-name> /cdrom
```

On HP-UX 10.x and later:

```
# /etc/mount -r -F cdfs <cdrom-device-name> /cdrom
```

On IRIX 6.x:

```
# /etc/mount -r -t iso9660 <cdrom-device-name> /CDROM
```

5 To verify that the CD-ROM is mounted, use the ls command to list the files:

```
# ls -R /cdrom
```

Supplemental notes: Ejecting the CD-ROM

After you complete the installation, eject the CD-ROM.

On Solaris with Volume Management, type:

```
% eject cdrom
```

On Solaris without Volume Management, type:

```
% su root
# umount /cdrom
# eject cdrom
# exit
```

On HP-UX, type:

```
% su root
# umount /cdrom
# exit
```

Press the eject button on the CD-ROM drive.

On IRIX, type:

```
% eject /CDROM
```

Supplemental notes: Using rs_install commands

The `rs_install` program includes four commands that you can use to rerun specific sections of the `rs_install` program without actually reinstalling any products: `license_setup`, `license_check`, `post_install`, and `options_setup`.

To use these commands, go to the `DeveloperTools.<version>` directory. For example:

```
% cd Rational/releases/DeveloperTools.<version>
% ./license_setup
```

- Use the `license_setup` command to rerun the license setup phase of the installation. Use `license_setup` to add your permanent license keys and whenever you want to change your licensing information.

- Use the `license_check` command to check your license server and the license file to make sure your license information is correct.

- Use the `post_install` command to rerun the post-installation phase of the installation. For more information, see “Installing the products: Post-installation” on page 15.

- Use `options_setup` to modify the list of users allowed to use the Rational Software product. For more information, see “Modifying the list of user IDs” on page 20.

Supplemental notes: Using the FLEXlm Software License Manager

The FLEXlm Software License Manager monitors license access, simultaneous usage, idle time, and so on. It includes the following components:

- A vendor daemon named `rational` that dispenses Purify, PureCoverage, and Quantify licenses. The `rational` daemon is used for all licensed Rational Software products. If you have products from other vendors that also use FLEXlm, they will include their own vendor daemons.

- A license manager daemon named `lmgrd` that is used by all licensed products from all vendors that use FLEXlm. The `lmgrd` daemon does not process requests on its own, but forwards requests to the appropriate vendor daemon.

- A Rational license file that specifies your license servers, vendor daemons, and product licenses.

The Rational license file

The Rational license file is a text file that is automatically created when you run the `rs_install` or `license_setup` programs.

The file for startup licenses is:

```
Rational/config/Temporary.dat
```

The file for permanent licenses is:

```
Rational/config/server-name.dat
```

Note: For best results, use the Rational license file only for Rational Software product licenses.

The `rs_install` program saves the license path to `<producthome>/.lm_license_file`. This is the path that Purify, PureCoverage, and Quantify use to locate the license file. You can override the location in `.lm_license_file` by setting the `LM_LICENSE_FILE` environment variable. The full path searched is equivalent to `$LM_LICENSE_FILE:‘cat.lm_license_file‘`.

Verifying that FLEXlm is working

To verify that your FLEXlm License Manager is operational and that the daemons are running, type the following commands on your license server:

```
% ps axww | grep -v grep | egrep "lmgrd|rational"
```

or

```
% ps -e | grep -v grep | egrep "lmgrd|rational"
```

The output should include lines similar to the following (your pathnames will vary):

```
538 ?? S 0:03.50 /rational/base/cots/flexlm.6.0i/platform/lmgrd
             -c /rational/config/servername.dat
             -l /rational/config/servername.log
539 ?? I 0:00.90 rational -T servername 6.0 3 -c ...
```

Using FLEXlm commands

The FLEXlm License Manager supports the following commands for system administration:

Use this command	To
`lmdiag`	Diagnose problems when you cannot check out a license

Use this command	To
`lmdown`	Shut down the license and vendor daemons
`lmhostid`	Report the license manager host ID of a workstation
`lmreread`	Reread the license file and start new vendor daemons
`lmstat`	Report status on daemons and feature usage
`exinstal`	Report on licenses in the license file you specify on the command line

Learning more about FLEXlm

For more information about the FLEXlm Software License Manager, see the *FLEXlm End User Manual* that is included on your Rational Software CD-ROM.

The *FLEXlm End User Manual*, along with answers to frequently asked questions about FLEXlm, is also available at `http://www.globetrotter.com/manual.htm`.

2 Using Purify

Purify is the most comprehensive run-time error detection tool available. It checks all the code in your program, including any application, system, and third-party libraries. Purify works with complex software applications, including multi-threaded and multi-process applications.

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

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

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

This chapter introduces the basic concepts involved in using Purify. For complete information, see the *Purify User's Guide*.

Finding errors in Hello World

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

Before you begin:

1 Create a new working directory. Go to the new directory and copy the `hello_world.c` program and related files from the `<purifyhome>/example` directory. For example:

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

2 Examine the code in `hello_world.c`.

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

```
 1  /*
 2   * Copyright (c) 1992-1997 Rational Software Corp.
     ...
 9   * This is a test program used in Purifying Hello World.
10   */
11
12  #include <stdio.h>
13  #include <malloc.h>
14
15  static char *helloWorld = "Hello, World";
16
17  main()
18  {
19     char *mystr = malloc(strlen(helloWorld));
20
21     strncpy(mystr, helloWorld, 12);
22     printf("%s\n", mystr);
23  }
```

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

Instrumenting a program

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

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

output — `Hello, World`

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

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

On IRIX, you can add `purify` in front of the compile/link command line, or you can Purify the executable:

```
% purify a.out
```

Note: On IRIX, Purify caches Dynamic Shared Objects (DSOs), not object files. Ignore all references to linkers and link-line options in this book. These do not apply to Purify on IRIX.

Compiling and linking in separate stages

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

On the compile line, use:

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

On the link line, use:

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

Running the instrumented program

Run the instrumented Hello World program:

HPUX

```
% a.out
```

IRIX

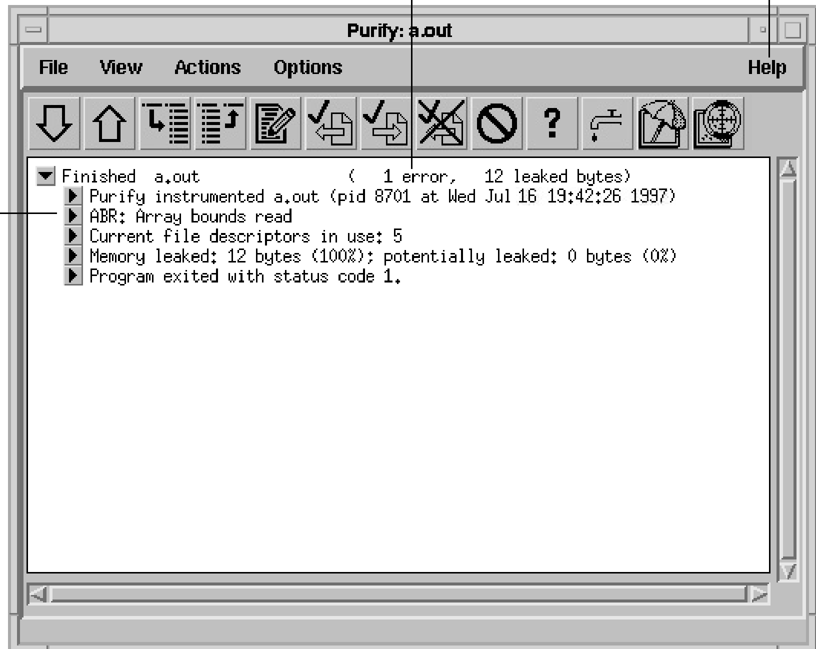
On IRIX, if you use `purify` on the executable instead of on the compile/link line, type:

```
% a.out.pure
```

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

Purify displays the number of access errors and leaked bytes detected

Click for a list of Purify error messages



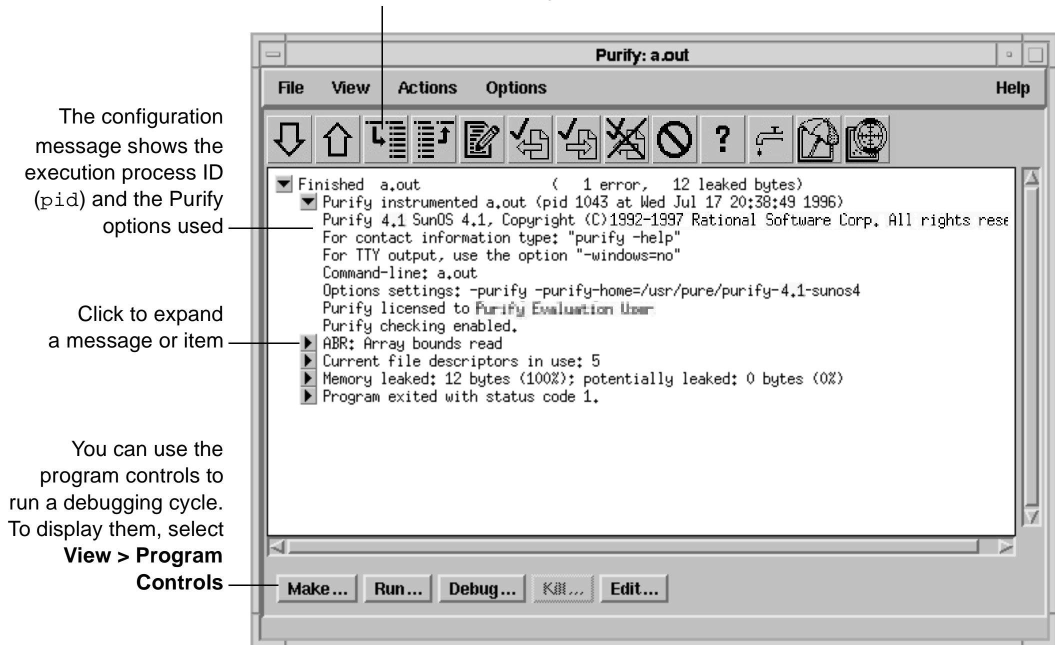
The Purify Viewer displays messages about the program, including errors such as this ABR error

For a description of a message, right click the message, then select **Explain message** from the pop-up menu

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

Seeing all your errors at a glance

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



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

Finding and correcting errors

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

```
Purify: a.out
File   View   Actions   Options                                    Help

▼ Finished  a.out              (   1 error,   12 leaked bytes)
  ▶ Purify instrumented a.out (pid 8934 at Wed Jul 16 19:42:26 1997)
  ▼ ABR: Array bounds read
    This is occurring while in:
          _doprnt         [libc.so.1.9]
          printf          [libc.so.1.9]
        ▼ main            [hello_world.c:22]
            #include <stdio.h>
            #include <malloc.h>

            static char *helloWorld = "Hello, World";

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

               strncpy(mystr, helloWorld, 12);
          ⇨    printf("%s\n", mystr);
            }
          start           [crt0.o]
    Reading 1 byte from 0x4423c in the heap.
    Address 0x4423c is 1 byte past end of a malloc'd block at 0x44230 of 12 bytes
    This block was allocated from:
          malloc          [rtlib.o]
        ▶ main            [hello_world.c:19]
          start           [crt0.o]
  ▶ Current file descriptors in use: 5
  ▶ Memory leaked: 12 bytes (100%); potentially leaked: 0 bytes (0%)
  ▶ Program exited with status code 1.
```

Click to expand the ABR message

The function call chain indicates an error occurring in `_doprnt` called by `printf`, in turn called on line 22 of `main`

The exact location of the error

The details of the access error

The allocation call chain shows that the memory block is allocated in the function `main` on line 19

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

On IRIX, system libraries retain their source file and line number information; therefore, the ▶ can appear next to a system library function whose source file is not available. When you click the ▶

for such a line, Purify prompts you for the location of the source file. Enter the location of the file if you know it, and then click `OK` to expand the line.

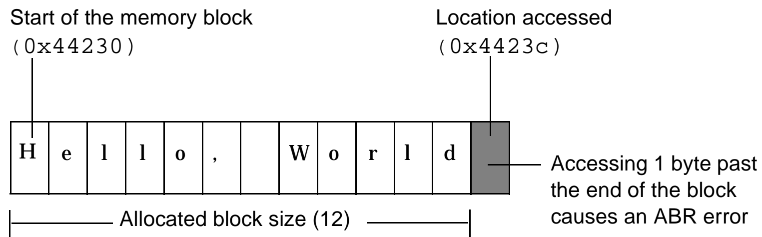
Understanding the cause of the error

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

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

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

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

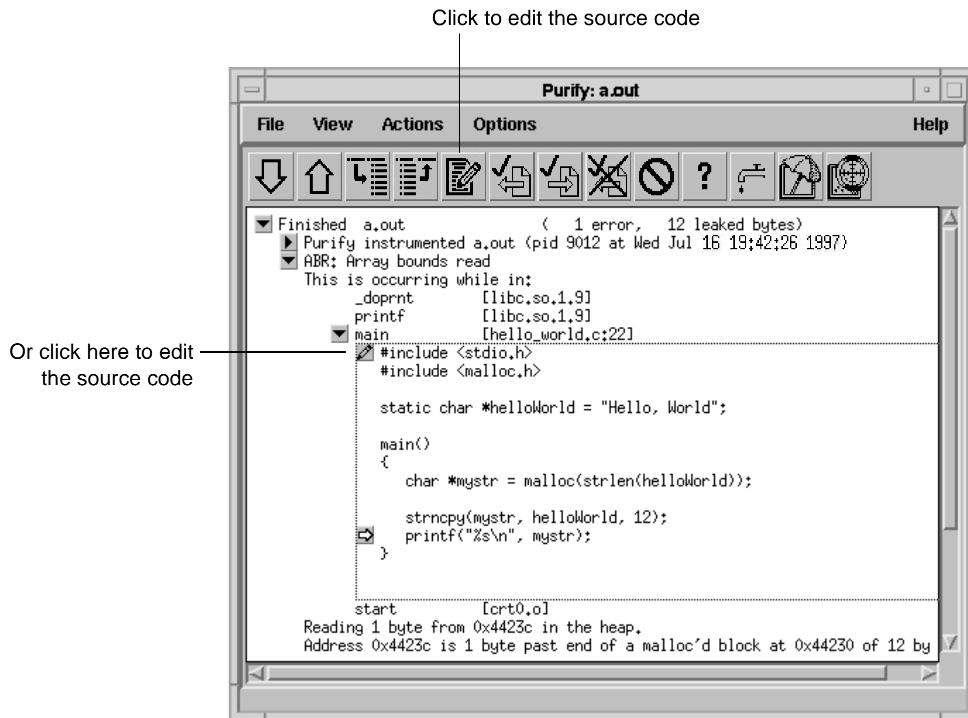


For more information, see “How Purify finds memory-access errors” on page 48.

Correcting the ABR error

To correct this ABR error:

1 Click the Edit tool to open an editor.



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

2 Change lines 19 and 21 as follows:

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

Finding leaked memory

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

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

1 Expand the memory-leaked summary for Hello World.

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

2 Expand the MLK message.

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

The memory-leaked summary reports 12 bytes of leaked memory

The call chain shows how the leaked memory was allocated

Memory analysis by category

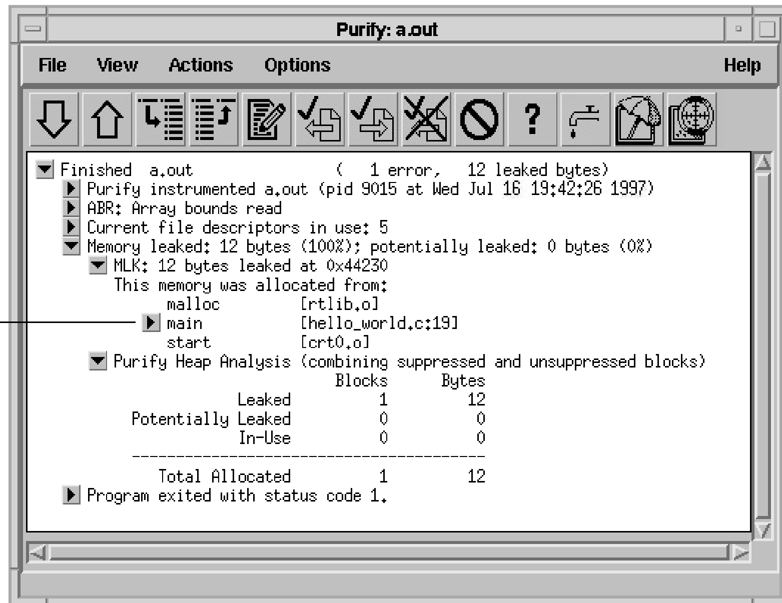
```
Purify: a.out
File   View   Actions   Options                          Help

Finished  a.out            (  1 error,   12 leaked bytes)
  Purify instrumented a.out (pid 9015 at Wed Jul 16 19:42:26 1997)
  ABR: Array bounds read
  Current file descriptors in use: 5
  Memory leaked: 12 bytes (100%); potentially leaked: 0 bytes (0%)
    MLK: 12 bytes leaked at 0x44230
     This memory was allocated from:
           malloc        [rtlib.o]
           main          [hello_world.c:19]
           start         [crt0.o]
    Purify Heap Analysis (combining suppressed and unsuppressed blocks)
                                   Blocks      Bytes
                     Leaked           1          12
          Potentially Leaked          0           0
                     In-Use           0           0
          ----------------------------------------
             Total Allocated          1          12
  Program exited with status code 1.
```

Correcting the MLK error

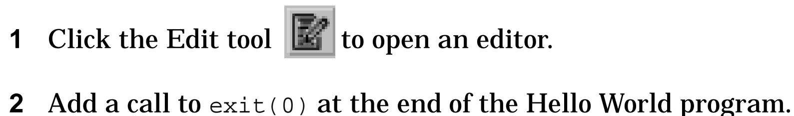
It is not immediately obvious why this memory was leaked. If you look closer, however, you can see that this program does not have an `exit` statement at the end. Because of this omission, the `main` function returns rather than calls `exit`, thereby making `mystr`—the only reference to the allocated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

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



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

To correct this MLK error:

1 Click the Edit tool  to open an editor.

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

Looking at the heap analysis

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

- Leaked memory
- Potentially leaked memory
- Memory in use

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

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

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

```
Purify: a.out
File   View   Actions   Options                                  Help

Finished  a.out              (   1 error,   12 leaked bytes)
  Purify instrumented a.out (pid 9015 at Wed Jul 16 19:42:26 1997)
  ABR: Array bounds read
  Current file descriptors in use: 5
  Memory leaked: 12 bytes (100%); potentially leaked: 0 bytes (0%)
    MLK: 12 bytes leaked at 0x44230
      This memory was allocated from:
            malloc          [rtlib.o]
            main            [hello_world.c:19]
            start           [crt0.o]
    Purify Heap Analysis (combining suppressed and unsuppressed blocks)
                                  Blocks      Bytes
                       Leaked          1         12
           Potentially Leaked          0          0
                       In-Use          0          0
           ----------------------------------------
              Total Allocated          1         12
  Program exited with status code 1.
```

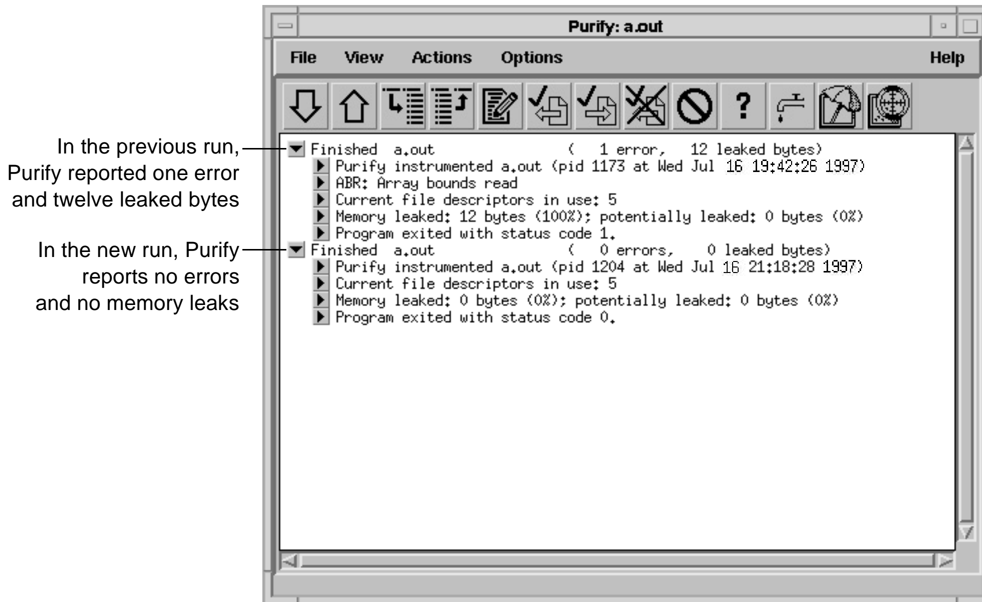
The exit status message provides information about:

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

Comparing program runs

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

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



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

Suppressing Purify messages

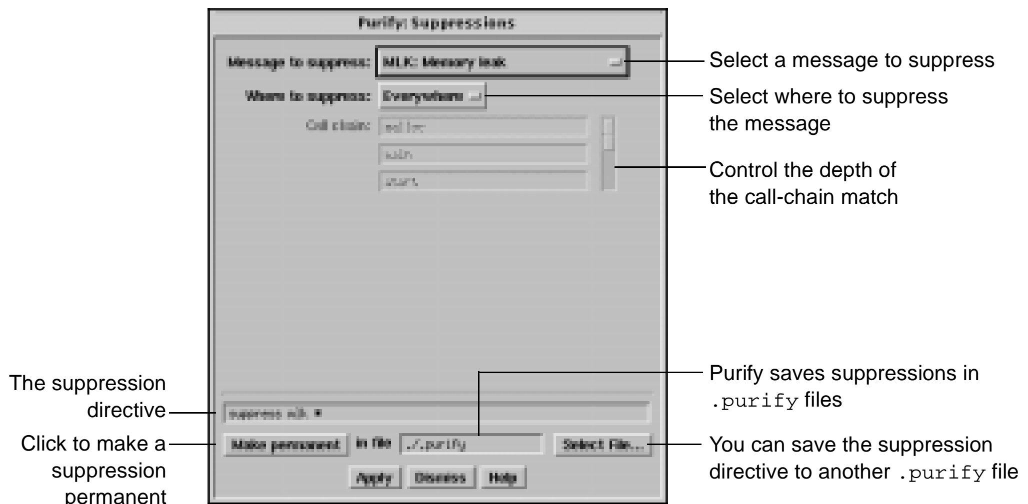
A large program can generate hundreds of error messages. To quickly focus on the most critical ones, you can suppress the less critical messages based on their type and source. For example, you might want to hide all informational messages, or hide all messages that originate in a specific file.

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

1 Select the message you want to suppress.

2 Select **Options** > **Suppressions**.

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



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

Saving Purify output to a view file

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

Saving a run to a view file from the Viewer

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

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

Opening a view file

To open a view file from the Viewer:

1. Select **File > Open**.
2. Select the view file you want to open.

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

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

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

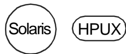
   This opens the `<filename>.pv` view file in a new Viewer.

Using your debugger with Purify

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

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

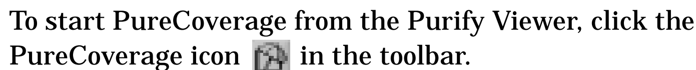
Using Purify with PureCoverage



Purify is designed to work closely with PureCoverage, Rational Software's run-time test coverage tool. PureCoverage identifies the parts of your program that have not yet been tested so you can tell whether you're exercising your program sufficiently for Purify to find all the memory errors in your code.

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

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

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

For more information, see Chapter 3, Using PureCoverage.

Purify API functions

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

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

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

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

Build-time options

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

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

Commonly used build-time options	Default
-always-use-cache-dir	**no**
Forces all instrumented object files to be written to the global cache directory	
-cache-dir	**<purifyhome>/cache**
Specifies the global directory where Purify caches instrumented object files	
-collector	none
Specifies the collect program to handle static constructors (for use with gcc, g++)	
-ignore-runtime-environment	**no**
Prevents the run-time Purify environment from overriding the option values used in building the program	
-linker	system-dependent
Sets the alternative linker to build the executables instead of the system default	
-print-home-dir	
Prints the name of the directory where Purify is installed, then exits	

Conversion characters for filenames

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

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

Run-time options

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

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

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

† *Can use the conversion characters listed on page 45.*

Purify messages

Purify reports the following messages:

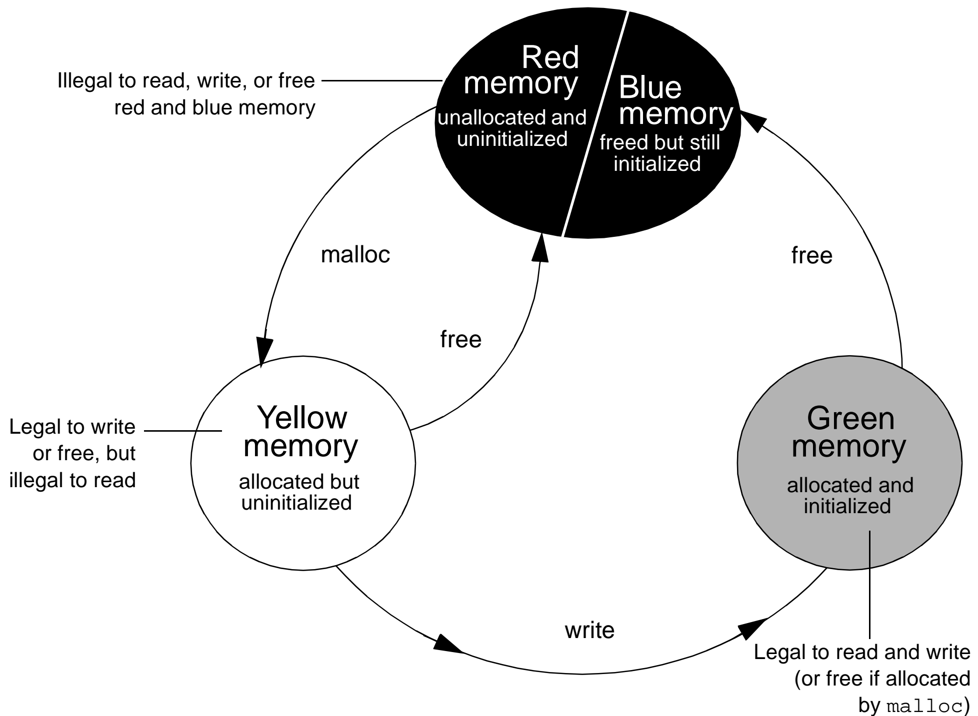
Message	Description	Severity*	Message	Description	Severity*
ABR	Array Bounds Read	W	**NPR**	Null Pointer Read	F
ABW	Array Bounds Write	C	**NPW**	Null Pointer Write	F
BRK	Misuse of Brk or Sbrk	C	**PAR**	Bad Parameter	W
BSR	Beyond Stack Read	W	**PLK**	Potential Leak	W
BSW	Beyond Stack Write	W	**SBR**	Stack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

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

How Purify finds memory-access errors

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

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



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

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

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

- Yellow: Memory returned by `malloc` or `new` is yellow. This memory has been allocated, so the program owns it, but it is uninitialized. You can write yellow memory, or free it if it is allocated by `malloc`, but it is illegal to read it because it is uninitialized. Purify sets stack frames to yellow on function entry.
- Green: When you write to yellow memory, Purify labels it green. This means that the memory is allocated and initialized. It is legal to read or write green memory, or free it if it was allocated by `malloc` or `new`. Purify initializes the data and `bss` sections of memory to green.
- Blue: When you free memory after it is initialized and used, Purify labels it blue. This means that the memory is initialized, but is no longer valid for access. It is illegal to read, write, or free blue memory.

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

How Purify checks statically allocated memory

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

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

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

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

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

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

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

3 Using PureCoverage

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.

Using PureCoverage, you can:

- 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 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

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

This chapter introduces the basic concepts involved in using PureCoverage. For complete information, see the *PureCoverage User's Guide*.

Finding untested areas of Hello World

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

Before you begin:

1. Create a new working directory. Go to the new directory, and copy the `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);
}
```

Instrumenting a program

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

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

```
% a.out
```

output —

```
Hello, World
```

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

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

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

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

```
PureCoverage 4.4 Solaris 2, Copyright 1994-1999 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 the instrumented program

Run the instrumented Hello World program:

```
% a.out
```

PureCoverage displays the following:

Name of the instrumented executable

You can use this command to display technical support contact information

Start-up banner

```
****  PureCoverage instrumented a.out (pid 3466 at Wed Feb 3 10:32:40 1999)
* PureCoverage 4.4 Solaris 2, Copyright 1994-1999 Rational Software Corp.
  * All rights reserved.
  * For contact information type: "purecov -help"
  * Command-line: a.out
  * Options settings: -purecov \
    -purecov-home=/usr/pure/purecov-4.4-solaris2
  * PureCoverage licensed to Rational Software Corp.
  * Coverage counting enabled.
Hello, World

****  PureCoverage instrumented a.out (pid 3466)  ****
  * Saving coverage data to /usr/home/pat/example/a.out.pcv.
```

Normal program output

PureCoverage saves coverage data to a `.pcv` file

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

Displaying coverage data

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

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

This displays the PureCoverage Viewer.

These columns show statistics for function usage

This column shows the number of adjusted lines

These columns show statistics for line usage

Summary information for the entire program

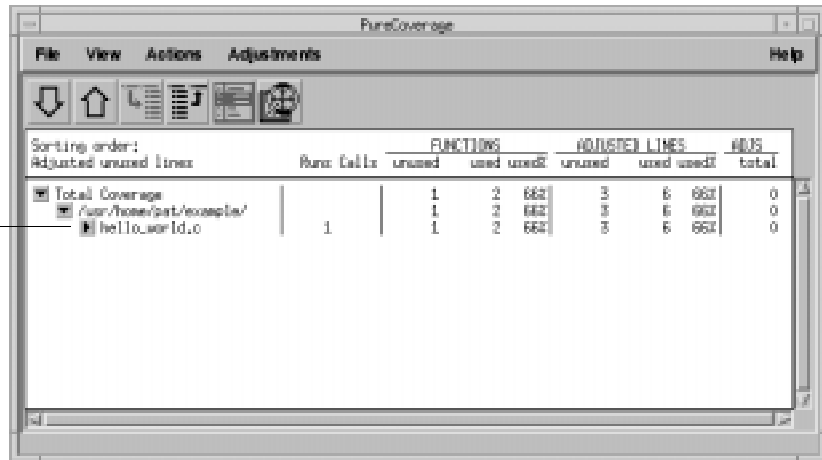
Information for the source directory

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

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

Expanding the file-level detail

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



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

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

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

Examining function-level detail

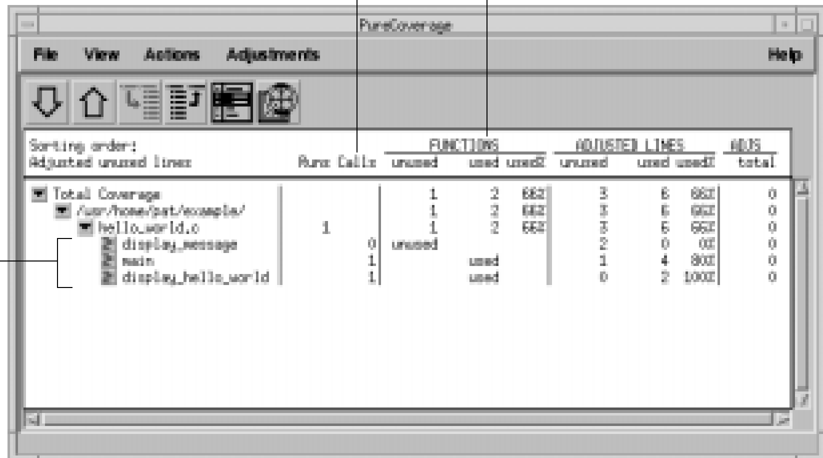
Expand the `hello_world.c` line to show function-level information.

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

The Calls column shows how many times the program called each function

The `FUNCTIONS` columns tell at a glance whether each function was used or unused

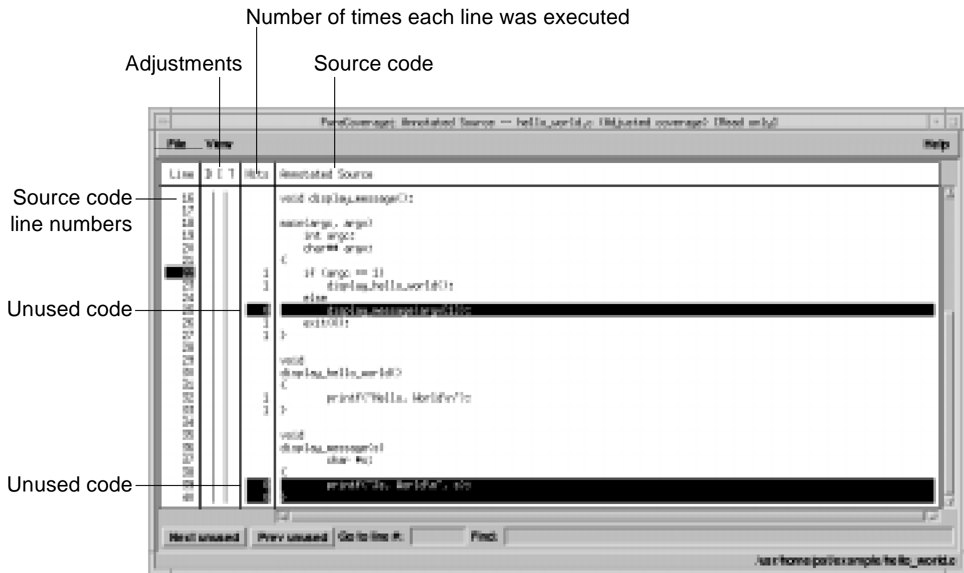
Function-level information includes the number of times the program called each function



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

Examining the annotated source

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



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

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

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

Improving Hello World's test coverage

To improve the test coverage for Hello World:

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

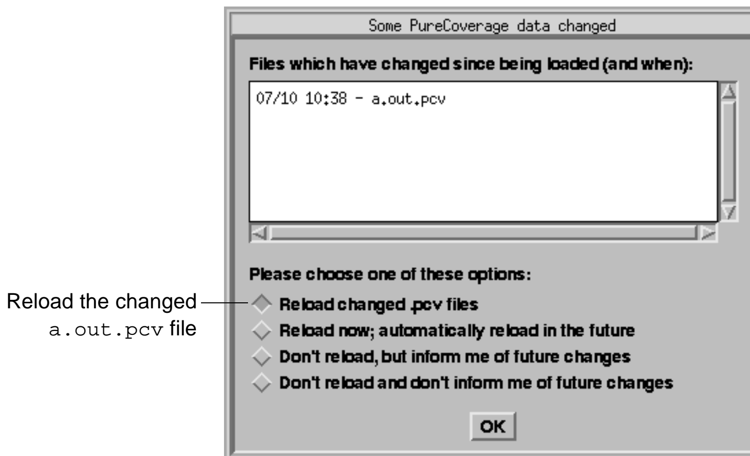
```
% a.out Goodbye
```

PureCoverage displays the following:

```
****  PureCoverage instrumented a.out (pid 17331 at Wed Feb 3 10:38:07 1999)
PureCoverage 4.4 Solaris 2, Copyright (C) 1994-1999 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-4.4-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 PureCoverage displays a dialog confirming that coverage data has changed for this run. 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.

Function and line coverage is now 100%

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

and the function `display_message` are now shown as used

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

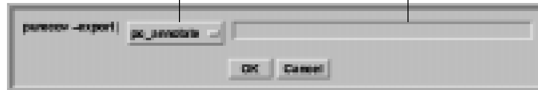
3 Select **File > Exit**.

Using report scripts

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

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

Select a script from the selection list   Type arguments



You can also run report scripts from the command line.

Report scripts

pc_annotate — Produces an annotated source text file

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

pc_below — Reports low coverage

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

pc_build_diff — Compares PureCoverage data from two builds of an application

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

pc_covdiff — Annotates the output of `diff` for modified source code — **Note**: Cannot run from Viewer

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

pc_diff — Lists files for which coverage has changed

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

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

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

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

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

pc_ssheet — Produces a summary in spreadsheet format

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

pc_summary — Produces an overall summary in table format

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

Build-time options

You can specify 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	**no**
Forces all PureCoverage instrumented object files to be written to the global cache directory	
-auto-mount-prefix	**/tmp_mnt**
Removes the prefix used by file system auto-mounters	
-cache-dir	**<purecovhome>/cache**
Specifies the global directory where PureCoverage caches instrumented object files	
-collector	none
Specifies the collect program to handle static constructors (for use with gcc, g++)	
-ignore-run-time-environment	**no**
Prevents the run-time PureCoverage environment from overriding the option values used in building the program	
-linker	system-dependent
Specifies a linker other than the system default for building the executables	

Run-time options

You can specify 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**	**%v.pcv**
Specifies an alternate file for writing coverage count data in binary format	
-follow-child-processes	**no**
Controls whether PureCoverage is enabled in forked child processes	
† **-log-file**	**stderr**
Specifies a log file for PureCoverage run-time messages	
-program-name	**argv[0]**
Specifies the full pathname of the PureCoverage instrumented program	
† **-user-path**	none
Specifies a list of directories to search for source code	

† Can use the conversion characters listed on page 45.

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

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`) 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`) in binary format	
-view	**-force-merge, -user-path**
Opens the PureCoverage Viewer for analysis of one or more coverage data files (`.pcv`)	

4 Using Quantify

Your application's run-time performance—its speed—is one of its most visible and critical characteristics. Developing high-performance software that meets the expectations of customers is not an easy task. Complex interactions between your code, third-party libraries, the operating system, hardware, networks, and other processes make identifying the causes of slow performance difficult.

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

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

Unlike sampling-based profilers, Quantify's reports do not include any overhead. The numbers you see represent the time your program would take without Quantify. Quantify instruments *all* the code in your program, including system and third-party libraries, shared libraries, and statically linked modules.

This chapter introduces the basic concepts involved in using Quantify. For complete information, see the *Quantify User's Guide*.

How Quantify works

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

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

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

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

Building and running an instrumented program

To instrument your program, add `quantify` to the front of the link command line. For example:

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

```
Quantify 4.4 Solaris 2, Copyright 1993-1999 Rational Software Corp.
Instrumenting: hello_world.o Linking
```

Run the instrumented program normally:

```
% hello_world
```

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

```
**** Quantify instrumented hello_world (pid 20352 at Sat 5 08:41:27
1999)
Quantify 4.4 Solaris 2, Copyright 1993-1999 Rational Software Corp.
    * For contact information type: "quantify -help"
    * Quantify licensed to Quantify Evaluation User
    * Quantify instruction counting enabled.
```

Program output

```
Hello, World.
```

Data transmission

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

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

Interpreting the program summary

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

Time Quantify expects the original program to take

Time spent executing program functions (compute-bound)

Time spent waiting for system calls to complete

Time spent loading dynamic libraries

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

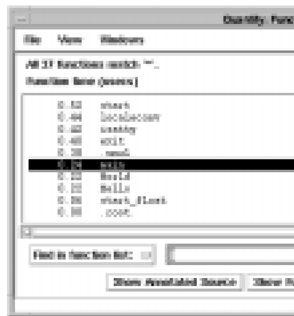
```
Quantify: Resource Statistics for hello_world (pid 20352)
 *                                              cycles       secs
 * Total counted time:                        16148821      0.323 (100.0%)
 *       Time in your code:                       2721      0.000 (  0.0%)
 *       Time in system calls:                  843950      0.017 (  5.2%)
 *       Dynamic library loading:             15302150      0.306 ( 94.8%)
 *
 *
 * Note: Data collected assuming a sparcstation_lx with clock rate of 50 MHz.
 * Note: These times exclude Quantify overhead and possible memory effects.
 *
 * Elapsed data collection time:       0.336 secs
 *
 * Note: This measurement includes Quantify overhead.
```

Using Quantify's data analysis windows

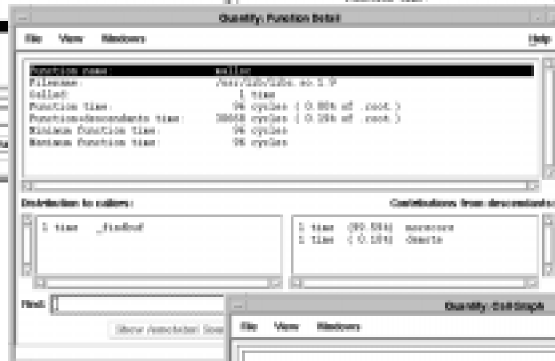
After transmitting the last dataset, Quantify displays the Control Panel. From here, you can display Quantify's data analysis windows and begin analyzing your program's performance.

CONTROL PANEL

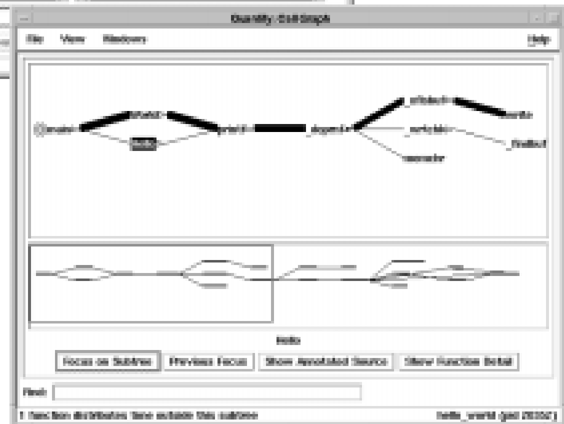
ANNOTATED SOURCE



FUNCTION LIST



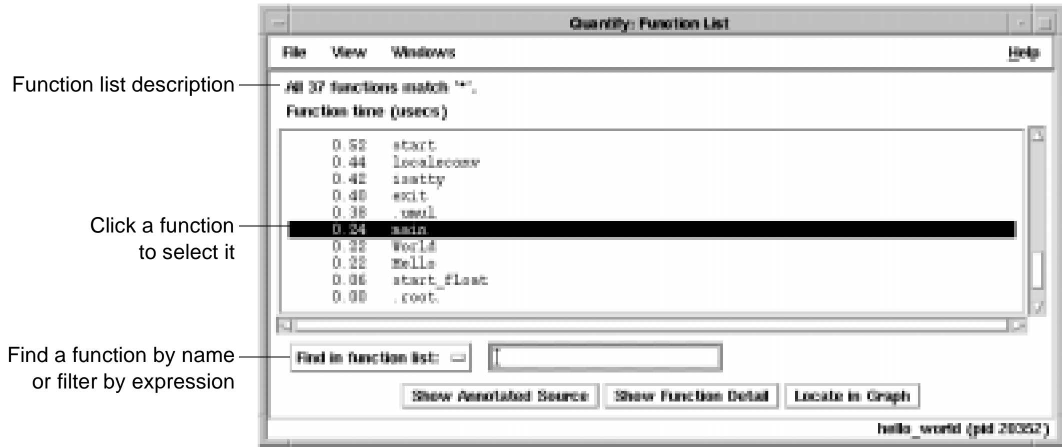
FUNCTION DETAIL



CALL GRAPH

The Function List window

The Function List window shows the functions that your program executed. By default, it displays the top 20 most expensive functions in your program, sorted by their *function time*. This is the amount of time a function spent performing computations (compute-bound) or waiting for system calls to complete.



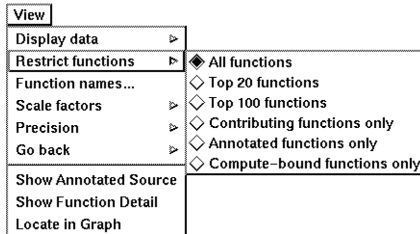
Sorting the function list

To sort the function list based on the various data Quantify collects, select **View > Display data.**

View	
Display data ▹	◆ Function time
Restrict functions ▹	◇ Function+descendants time
Function names...	◇ Descendants time
Scale factors ▹	◇ System call time
Precision ▹	◇ Register window trap time
Go back ▹	◇ Number of function calls
	◇ Number of callers
Show Annotated Source	◇ Number of descendants
Show Function Detail	◇ Number of system calls
Locate in Graph	◇ Number of register window traps

Restricting functions

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



You can restrict the list to the top 20 or top 100 functions in the list, to the functions that have annotated source, to functions that are compute-bound (make no system calls), or to functions that contribute non-zero time for a recorded data type.

The Call Graph window

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

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

Thicker lines mean more expensive paths

Click and drag anywhere in the call graph to move to a new location

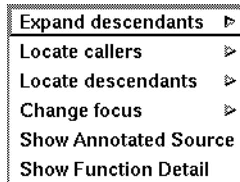
Or click and drag the Viewport to move to a new location

The selected function

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

Using the pop-up menu

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

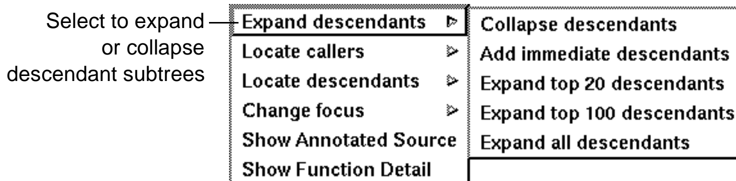


You can use the pop-up menu to:

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

Expanding and collapsing descendants

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

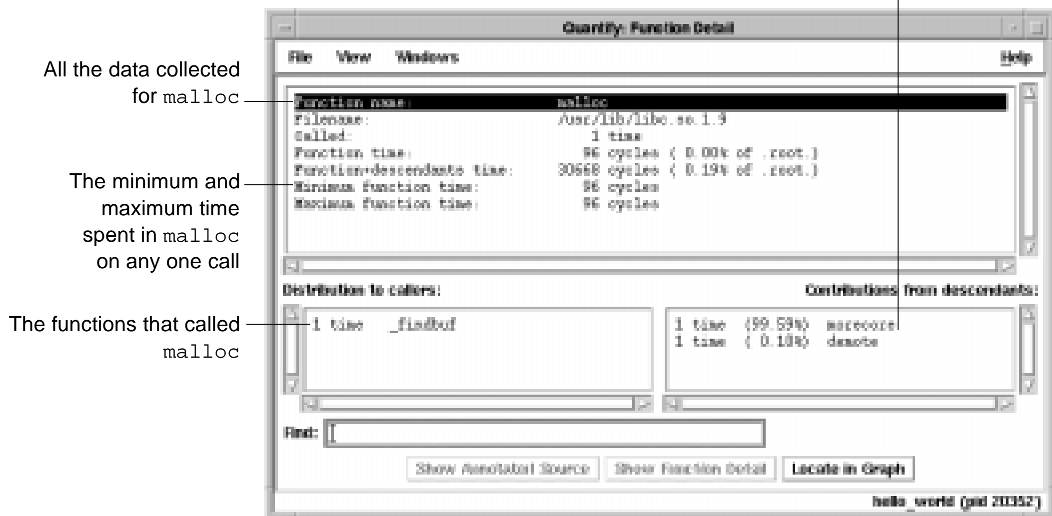


After expanding or collapsing subtrees, you can select **View > Redo layout** to remove any gaps that your changes create in the call graph.

The Function Detail window

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

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

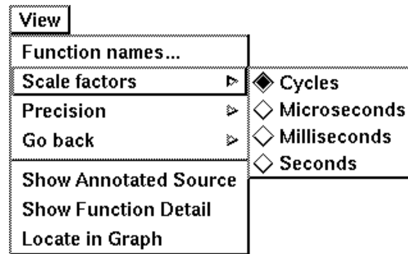


Double-click a caller or descendant function to display the detail for that function.

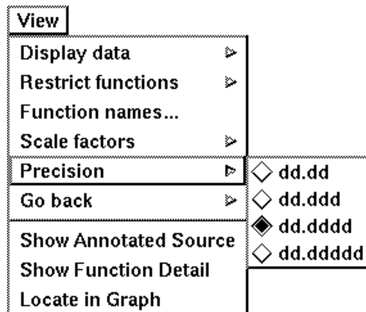
The function time and the function+descendants time are shown as a percentage of the total accumulated time for the entire run. These percentages help you understand how this function's computation contributed to the overall time of the run. These times correspond to the thickness of the lines in the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

Changing the scale and precision of data

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



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



Saving function detail data

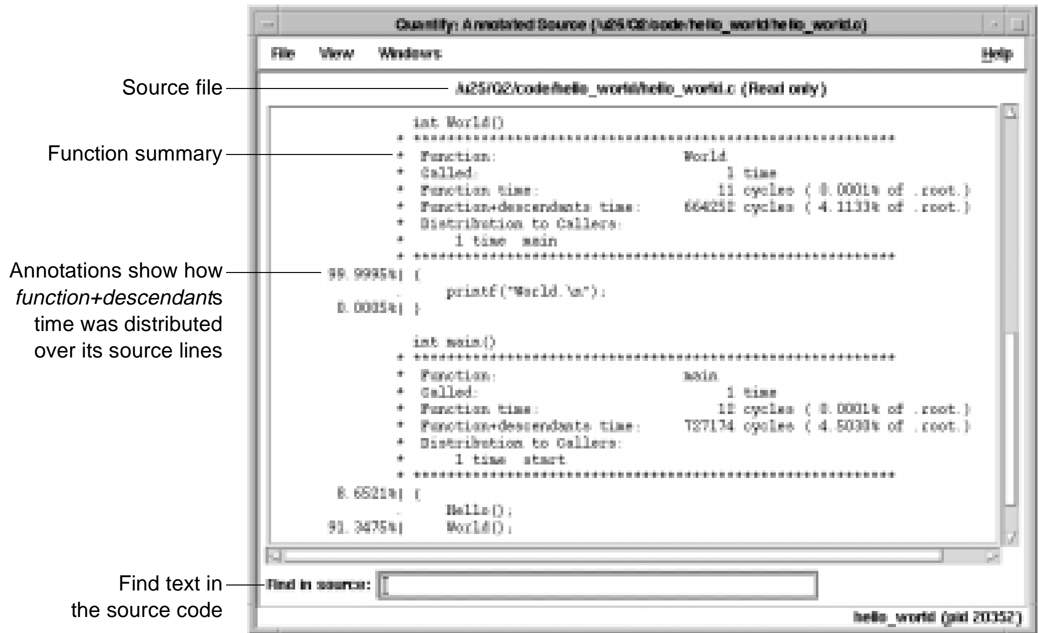
To save the current function detail display to a file, select **File > Save current function detail as**.

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

The Annotated Source window

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

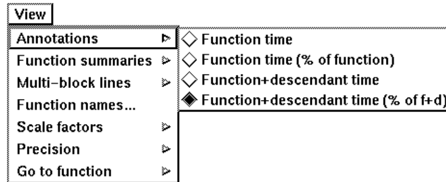
Note: The Annotated Source window is available only for files that you compile using the -g debugging option.



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

Changing annotations

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



Saving performance data on exit

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

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

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

Comparing program runs with qxdiff

The `qxdiff` script compares two export data files from runs of an instrumented program and reports any changes in performance. To use the `qxdiff` script:

1. Save baseline performance data to an export file. Select **File > Export Data As** in any data analysis window.
2. Change the program and run Quantify on it again.
3. Select **File > Export Data As** to export the performance data for the new run.
4. Use the `qxdiff` script to compare the two export data files. For example:

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

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

Below is the output from this example.

qxdiff lists the functions that have changed . . .

and summarizes the differences for the entire run

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

Build-time options

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

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

Commonly used build-time options	Default
-always-use-cache-dir Specifies whether instrumented files are written to the global cache directory	**no**
-cache-dir Specifies the global cache directory	**<quantifyhome>/cache**
-collection-granularity Specifies the level of collection granularity	**line**
-collector Specifies the collect program to handle static constructors in C++ code	none
-ignore-runtime-environment Prevents the run-time Quantify environment from overriding option values used in building the program	**no**
-linker Specifies an alternative linker to use instead of the system linker	system-dependent
-use-machine Specifies the build-time analysis of instruction times according to a particular machine	system-dependent

qv run-time options

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

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

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

Run-time options

Specify run-time options on the link line or by using the QUANTIFYOPTIONS environment variable. For example:

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

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

API functions

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

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

† `<action>` is one of: `start`, `stop`, `is`. For example: `quantify_stop_recording_system_call`

Index

Symbols

%V, %v, %p 45

A

ABR, array bounds read error
  correcting 36
  in Hello World 34
access errors, how Purify finds 48
account number, Rational Software 10
-add-annotation 79
adjusted lines 55
-always-use-cache-dir 79
analysis-time options 63
Annotated Source window
  PureCoverage 58
  Quantify 76, 77
a.out.pcv 54
API functions
  Purify 44
  Quantify 81
appending function detail 75
-avoid-recording-system-calls 80

B

blue memory color 49
building programs, *see* instrumenting a program
build-time options
  PureCoverage 62
  Purify 45
  Quantify 79

C

cache subdirectory
  creating in home directory 16
  location of 23
-cache-dir 16, 79
caching dynamic shared objects on IRIX 31
caching options
  PureCoverage 62
  Purify 45
  Quantify 79
Call Graph window, Quantify 72, 73
Calls column, PureCoverage 57
CD-ROM
  ejecting 25
  mounting 23
changing annotations, Quantify 77
characters, conversion 45
code, *see* source code
collapsing subtrees 73
-collection-granularity 79
-collector 79
color, *see* memory color
comparing program runs
  with PureCoverage 59
  with Purify 40
  with Quantify qxdiff script 78
compiling and linking 31
compute-bound
  functions 70, 71
  time 68
configuration message 33
controls, Purify program 33
conversion characters for filenames 45
coverage data
  file level 56
  function level 57
  in PureCoverage Viewer 55
cycles
  counted by Quantify 66
  scale factor 75

D

daemons, and licensing 26

data
  comparing export files 78
  saving Quantify data 77
debugger(s)
  JIT debugging 43
  scripts on HP-UX 18
  using with Purify 43
debugging option, *see* -g debugging option
defaults file 14
deleting product releases 20
directories
  cache 16
  installation 9, 22
  PureLA 12
  Rational 22–23
disk space requirements 9
DSO caching on IRIX 31
dynamic library, timing 68
dynamic shared objects caching 31

E

editing source code 36, 38
ejecting CD-ROM 25
e-mail, requesting licenses by 21
environment variables
  LM_LICENSE_FILE 27
  MANPATH 16
  PATH 17
  PURECOVOPTIONS 62
  PUREOPTIONS 16
  PURIFYOPTIONS 46
  QUANTIFYOPTIONS 80
evaluation license 11
executable, Purify'ing on IRIX 31
expanding subtrees 73
expiration date, licenses 11
exporting Quantify data 77

F

file(s)
  a.out.pcv 54
  installing product 23
  Purify view 42
  Rational license 26
  rational.opt 19
  rs_install.defaults 14
  users.purela 12
filename conversion characters 45
filesystems, installing on read-only 15
FLEXlm
  commands 27
  End User Manual 28
  GLOBEtrotter Web site 28
  License Manager 26
Function Detail window 74
  saving data 75
  scale and precision of data 75
Function List window
  finding top contributors 70
  restricting functions 71
function+descendants time 74
functions
  compute-bound 71
  coverage detail 57
  restricting display in Quantify 71
  sorting in Quantify 70
  See also API functions
Functions columns, PureCoverage 57

G

-g debugging option
  and PureCoverage 53
  and Purify 34
  and Quantify 67, 76
GLOBEtrotter Web site 28
graph, *see* Call Graph window
green memory color 49

H

heap analysis, Purify 39
Hello World example
  PureCoverage 52
  Purify 30
help, technical 7
hiding
  functions in Quantify 71
  messages in Purify 41
HP-UX debugger scripts 18

I

-ignore-runtime-environment 79
installation
  directory 9, 22
  evaluation license 11

installation (continued)
- on read-only filesystems 15
- permanent license 11, 21
- requirements 9
- rs_install commands 25
- startup license 11, 15
- term license agreement 11
- user input 9

instrumenting a program
- description of 7
- with PureCoverage 53
- with Purify 31
- with Quantify 67

integration, Purify and PureCoverage 43

IRIX
- compile/link command 31
- DSO caching 31
- running a Purify instrumented program 32

J

just-in-time debugging 43

K

keys, license 12, 14

L

leaks, *see* memory leaks

library
- system and PureCoverage 57
- time loading dynamic 68

license daemon, lmgrd 26

license file 12, 26

license keys 12, 14

License Manager, FLEXlm 26

license server 14
- port number 10
- requirements 10

license(s)
- checking 26
- evaluation 11
- expiration date 11
- key types 11
- permanent 11, 21
- quantity 11
- setting up 25
- startup 11, 15, 21
- term license agreement 11
- user IDs 12, 19–20

license_check command 26

license_setup command 25

line numbers
- -g option 31, 34
- on IRIX 34

-linker 79

links, symbolic 17

LM_LICENSE_FILE environment variable 27

lmgrd license daemon 26

local variable names, displaying 31

M

machine cycles 66

MANPATH environment variable 16

manual pages 16

-measure-timed-calls 80

memory access errors
- example 34
- how Purify finds 48

memory color 48

memory in use message 39

memory leaks 44
- definition 39
- heap analysis 39
- message 37
- new leaks button 37
- potential 39
- purify_new_leaks 44

menu, Quantify pop-up 73

messages
- Purify 47
- suppressing Purify 41

MLK, memory leak 38
- example 37

mounting CD-ROM 23

N

new memory leaks, Purify 37

O

Object Code Insertion (OCI) 66

operating system, identifying 23

options
- PureCoverage analysis-time 63

options (continued)
- PureCoverage build-time 62
- PureCoverage run-time 62
- Purify build-time 45
- Purify run-time 46
- Quantify build-time 79
- Quantify run-time 80
- qv run-time 79

options (by name)
- -add-annotation 79
- -always-use-cache-dir 79
- -avoid-recording-system-calls 80
- -cache-dir 79
- -collection-granularity 79
- -collector 79
- -ignore-runtime-environment 79
- -linker 79
- -measure-timed-calls 80
- -print-annotations 79
- -record-child-process-data 80
- -record-system-calls 80
- -report-excluded-time 80
- -run-at-exit 80
- -run-at-save 80
- -save-data-on-signals 80
- -save-thread-data 80
- -use-machine 79
- -view 55
- -windows 79, 80
- -write-export-file 79, 80
- -write-summary-file 79, 80

options file 19
options_setup command 26
overhead, Quantify 68

P

PATH environment variable 17
performance data 67
- saving 77

permanent licenses
- defined 11
- installing manually 21
- requesting 21

pop-up menu, Quantify 73
port number, license server 10
post_install command 26
post-installation 15
potential memory leak 39
-print-annotations 79
product license keys 14
producthome directory 22
products, removing 20
program controls, Purify 33
program runs, comparing
- Quantify qxdiff script 78
- with PureCoverage 59
- with Purify 40

program summary, Quantify 68
programs, running instrumented
- PureCoverage 54
- Purify 32
- Quantify 67

PureCoverage
- benefits 51
- symbolic links for 17
- using with Purify 43
- Viewer 55

PURECOVOPTIONS environment variable 62
PureLA directory 12
PUREOPTIONS environment variable 16
Purify
- API functions 44
- instrumenting a program 31
- messages 47
- Viewer 32

PURIFYOPTIONS environment variable 46

Q

Quantify
- API functions 81
- build-time options 79
- Call Graph window 72, 73
- overhead 68
- repeatability of timing 66
- run-time options 80
- symbolic links for 18

QUANTIFYOPTIONS environment variable 80
qv 67
qv script files 18
qx script files 18
qxdiff script 78

R

Rational account number 10
rational daemon 26

rational.opt options file 19
README file location 13
read-only filesystems 15
-record-child-process-data 80
-record-system-calls 80
red memory color 49
Redo layout, Quantify 73
removing previous releases 20
report(s)
  program summary 68
  PureCoverage scripts 61
-report-excluded-time 80
restricting functions in Quantify 71
rs_install
  commands 25
  program 9, 13
  user input 9
rs_install.defaults file 14
-run-at-exit 80
-run-at-save 80
running an instrumented program
  PureCoverage 54
  Purify 32
  Quantify 67
runs
  column, PureCoverage 56
  comparing with PureCoverage 59
  comparing with Purify 40
  comparing with Quantify 78
run-time options
  PureCoverage 62
  Purify 46
  Quantify 80
  qv 79

S

-save-data-on-signals 80
-save-thread-data 80
saving
  function detail data 75
  Purify run 42
  Quantify data 77
scale factors 75
scripts
  HP-UX debugger 18
  PureCoverage 17
  PureCoverage report 61
  Quantify 18
  qxdiff 78
server, license 13, 14
server-name.dat file 27
sorting function list 70
source code
  annotated in PureCoverage 58
  annotated in Quantify 76
  displaying filenames 34
  editing from Viewer 36, 38
  line numbers, Purify 34
  number of lines displayed 36
startup license 11, 15, 21
statically allocated memory 50
subtrees, Quantify 73
summary, Quantify program 68
support, technical 7
suppressing Purify messages 41
symbolic links 17
  for HP-UX debugger scripts 18
  for PureCoverage 17
  for Purify 17
  for Quantify 18
system call timing 66
system libraries and
    PureCoverage 57

T

technical support 7
Temporary.dat file 26
term license agreement (TLA) 11
time
  compute-bound 68
  function+descendants 74
  in code 68
  loading dynamic libraries 68
  to collect the data 68
TLA 11
Total Coverage row,
    PureCoverage 55

U

uname command 23
uninstall command 20
-use-machine 79
user IDs, for licensing 12, 19–20

V

variable, *see* environment variable
-view 55
view file, Purify 42

Viewer 55
  PureCoverage 55
  Purify 32
viewport, call graph 72

W

-windows 79, 80
windows
  PureCoverage viewer 55
  Purify viewer 32
  Quantify data analysis 69
World Wide Web sites
  GLOBEtrotter 28
-write-export-file 79, 80
-write-summary-file 79, 80

Y

yellow memory color 49