Rational the e-development company™

UNIX

Installing and Getting Started

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

PART NUMBER: 800-024983-000

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

Rational®
the e-development company™

COPYRIGHT NOTICE

U.S. GOVERNMENT RIGHTS NOTICE

TRADEMARK NOTICE

U.S. PATENT NOTICE

Printed in the U.S.A.

Contents

Preface

What's in this guide?

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

- Installing the products.

  Note: PurifyPlus is a Rational product that includes Purify, PureCoverage, and Quantify, and provides a unified procedure for installing all three applications on your system at the same time.

- 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.

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
```

Audience

Read this guide if you are responsible for installing Rational PurifyPlus, Purify, PureCoverage, or Quantify, or if you need an introduction to the use of Purify, PureCoverage, or Quantify.

Other resources

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

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

  Note: You can also view the help systems independently of the products. Open the following in your Netscape browser:

  - `‘purify -printhomedir‘/UI/html/punix.htm`
  - `‘purecov -printhomedir‘/UI/html/pcu.htm`
  - `‘quantify -printhomedir‘/UI/html/qunix.htm`

- 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.

Installing the products 1

The basic steps

Note: Rational® PurifyPlus is a Rational product that includes Rational Purify®, PureCoverage®, and Quantify®, and provides a unified procedure for installing all three applications on your system at the same time.

To install PurifyPlus, Purify, PureCoverage, or Quantify:

1. If you have purchased a permanent license or a term license agreement (TLA) from Rational Software, obtain a .upd file with license information from the AccountLink page on Rational Software's website.

   If you are installing the product with a startup or evaluation key, skip this step.

   Note: A permanent key allows the use of the product without time limits. A TLA key allows the use of the product for a specific period of time. A startup license key gets you up and running as soon as you receive your Rational product, but you will have to enter a permanent or TLA key later to continue using the product. An evaluation license key is valid for a limited time, but again you will have to enter a permanent or TLA key later to ensure continued use of your Rational product.

2. Install the product on your system using the `rs_install` program.

This chapter tells you how to gather the information you need to perform these steps, as well as instructions to get you started using AccountLink and the `rs_install` program. It 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.

Information you need for AccountLink

In addition to contact information, you will need to provide the following to obtain a .upd file with licensing information:

Data for AccountLink	**Notes**	**Your Entry**
Your Rational account number	Source: your Rational license key certificate.	
License Type	Permanent and TLA licenses for PurifyPlus can be either Floating licenses or Named User licenses. Permanent and TLA licenses for Purify, PureCoverage, and Quantify are always Named User licenses. For further information, see *Maintaining the rational.opt options file* on page 11.	
Rational Product Line		DEVELOPER TOOLS
Product Name	Source: your Rational license key certificate.	PurifyPlus for UNIX, Purify for UNIX, PureCoverage for UNIX, or Quantify for UNIX
Quantity	Source: your Rational license key certificate. This is the number of named-user licenses that you have purchased.	
Host Name and hostid	This is the name and hostid of the machine that you intend to use as your license server host. If 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. If you do not know the hostid of your license server host, download the tool `gethostinfo.sh` from AccountLink, and run it on your system.	

Obtaining a .upd file using AccountLink

Access AccountLink at http//:www.rational.com/accountlink.

When you supply the required information to AccountLink, you will receive by return email a file named `license_for_<server name>.upd`. Save this file as a text file in a location that is accessible from the installation machine.

If you are installing a permanent or TLA license, you need this file before you proceed with the next step, running `rs_install`.

Information you need for rs_install

You need all the information in the following table to install your Rational product, unless you have previously set up a FLEXlm license for the product. If you have set up a FLEXlm license, all the data you need is already available to the `rs_install` program.

Data for rs_install	Notes	Your Entry
The full pathname to the installation location (referred to in this chapter as `Rational`).	This is the directory for installing all Rational Software products. You must have **20 megabytes** of free disk space for each installation of Purify, Quantify, and PureCoverage. You must have **60 megabytes** for a complete installation of PurifyPlus. 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* on page 14. This section also shows you the structure of the directory after installation.	
License key type.	Source: your Rational license key certificate. **Note:** To enter a permanent or TLA license key after you've been using a startup or evaluation license, see *Entering a permanent or TLA license key after initial installation* on page 14.	permanent, TLA, startup, or evaluation

Data for rs_install	Notes	Your Entry
Host name or IP address of the host machine on which the license server is to run (the “license server host”).	This data is required only if you are setting up a permanent or TLA license and cannot import the `license_for_<server name>.upd` file. If the license server host is different from the installation machine, you should have remote shell access from the installation machine to the license server host. If you do you have remote shell access, the `rs_install` program provides instructions for how to proceed. In addition, the installation directory must be accessible from the license server host.	
License server port number.	This data is required only if you are setting up a permanent or TLA license and cannot import the `license_for_<server name>.upd` file. 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 27000–27009 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.	
License quantity.	This data is required only if you are setting up a permanent or TLA license and cannot import the `license_for_<server name>.upd` file. Source: your Rational license key certificate.	
Expiration date.	This data is required only if you are setting up an evaluation or startup license. Source: 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 for rs_install	**Notes**	**Your Entry**
Note: If you are installing a permanent or TLA Named User license, you must supply user names for each individual who will be using the product. **You must include your user name in order to perform the post-installation selt-test successfully.** User names are recorded in the FLEXlm options file, `rational.opt`. For information about the options file, see *Maintaining the rational.opt options file* on page 11. You do not need this information if you are installing a Floating license for PurifyPlus. To input User names, 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 user names 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 names, either while you're running `rs_install` or afterwards. If the number of user names is not the same as the number of licenses you bought, `rs_install` will help you correct the list.	
B. user names (all names; or some or none, in combination with an option to generate dummy names).	You can enter all user names. The number of names you enter must match the number of licenses you purchased. You can enter some user names, 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.	
Name for the Rational license file (`<server name>.dat`), including full pathname.	This name is required only if you are setting up a permanent or TLA license. The `rs_install` program, which creates this file when it runs, will suggest `<server name>.dat` as a default. If you want to use an existing Rational license .dat 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 with the new data. For information, see *The Rational license file* on page 18.	

Installing the products using 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 a Netscape browser.

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 *Mounting the CD-ROM* on page 15.

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 and .html documentation.

     Note: Your users can get online help only if you install the html documentation.

   - 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 *After you install: Post-installation tasks* on page 8 for any necessary post-installation procedures.

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 *Using rs_install commands* on page 17.
- **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 ro TLA license key, you can install it right away. You can also request a permanent license key at www.rational.com/accountlink. 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 or TLA license key as soon as possible.
- **Can I import existing user names from an earlier installation of the product installed with Named User licensing?** Yes. If you installed the product previously under FLEXlm, the user names 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 names. You can also edit the user names after installation; see *Maintaining the rational.opt options file* on page 11.

- **How do I proceed if I already have other Rational product licenses installed on my server?** You must add the new licenses to your current Rational product license file. To do this, specify the current Rational license file as the license file name instead of using the default.
- **How do I get updates for the rs_install program and for the Rational products?** You can get updates from within the `rs_install` program, though you must be running the program on a machine that has network access. The `rs_install` program's Licensing Options screen lets you select an item to download the latest version of `rs_install` (in which case `rs_install` replaces itself and restarts using the new version) or get product updates.

After you install: Post-installation tasks

For Rational PurifyPlus

The single post-installation task for Rational PurifyPlus is to notify all users of the product that they must source either `purifyplus_setup.csh` or `purifyplus_setup.sh`, depending on the shell they are using. Sourcing these files makes the product license available to the users and updates their command search path and man page path.

These files can be found in the Rational directory.

For Rational Purify, PureCoverage, and Quantify

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 *Using rs_install commands* on page 17.

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: If you are using Named User licensing, 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 11.

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 online help system.

- 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 online help system.

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 11. To remove previous versions of the products, see *Removing a previous product release* on page 13.

Maintaining the rational.opt options file

Named User licensing is always used with Purify, PureCoverage, and Quantify, and is available for use with PurifyPlus. Under Named User licensing, the user names of all users who are authorized to run Purify, PureCoverage, and Quantify must be listed in the rational.opt options file. The number of user names 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. You can 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 names, one for each license you purchased. You don't have to enter all user names 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 names 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=456778982 GROUP DevTools1
GROUP DevTools2 alice harry
INCLUDE quantify:KEY=12345778654 GROUP DevTools2
```

The KEY is the license key that you receive from Rational Software in the .dat license file.

Modifying the list of user names

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

You can add, change, or delete user names by running the `options_setup` script. You can also add, change, or delete user names 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.

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 or TLA 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 a permanent or TLA license key from Rational Software at www.rational.com/accountlink and install it to ensure continued use of the product. The startup license key and other licensing information is available from the 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 *Using the FLEXlm Software License Manager* on page 18.

Requesting your permanent or TLA license key

To request a permanent or TLA license key, go to www.rational.com/accountlink and follow the instructions provided there.

Entering a permanent or TLA license key after initial installation

To enter your permanent or TLA 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 *Using rs_install commands* on page 17.
2. For the licensing option, select the option for setting up a permanent license.

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

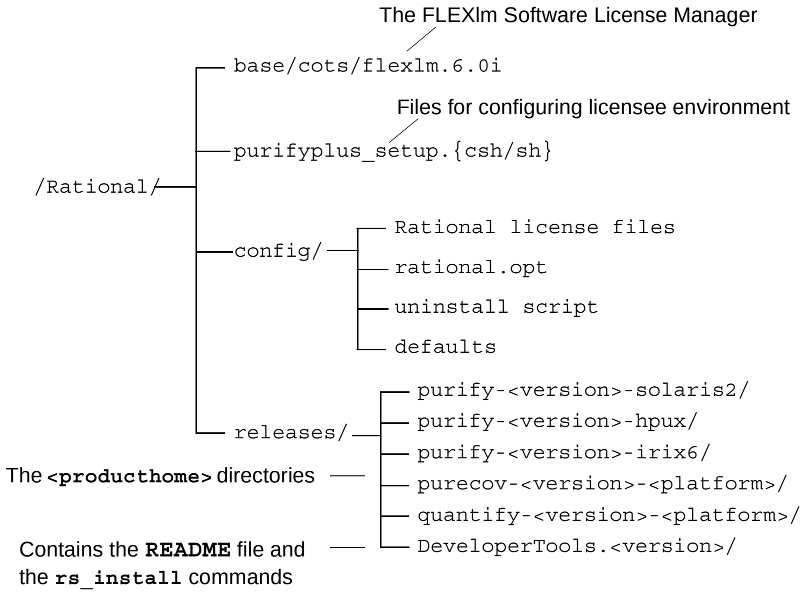
You need a publicly readable directory for the installation of Purify, PureCoverage, and Quantify. If one does not already exist, you can create it when you run `rs_install`. You can also create it manually before you start `rs_install`.

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:



The FLEXlm Software License Manager
base/cots/flexlm.6.0i
Files for configuring licensee environment
purifyplus_setup.{csh/sh}
/Rational/
config/
Rational license files
rational.opt
uninstall script
defaults
releases/
purify-<version>-solaris2/
purify-<version>-hpux/
purify-<version>-irix6/
purecov-<version>-<platform>/
quantify-<version>-<platform>/
DeveloperTools.<version>/
The <producthome> directories
Contains the README file and the rs_install commands

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 9.

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.

Solaris IRIX

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:

   Solaris

   On Solaris systems without Volume Management:

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

   HPUX

   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 the following command:

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

   IRIX

   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
   ```

Ejecting the CD-ROM

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

Solaris On Solaris with Volume Management, type:

```
% eject cdrom
```

On Solaris without Volume Management, type:

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

HPUX On HP-UX, type:

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

Press the eject button on the CD-ROM drive.

IRIX On IRIX, type:

```
% eject /CDROM
```

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 import your permanent or TLA 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 *After you install: Post-installation tasks* on page 8.

- 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 names* on page 12.

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 and evaluation licenses is:

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

The default file for permanent or TLA licenses is:

```
Rational/config/<license 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
`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.

Using Rational Purify

2

Rational Purify: What it does

Rational® 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 online help system.

Finding errors in Hello World

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

Before you begin:

1. If Purify has been installed on your system as a component of Rational PurifyPlus, the Rational directory contains the files `purifyplus_setup.csh` and `purifyplus_setup.sh`. Source the file that is appropriate to your shell to license Purify for your use.
2. 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* .
   ```

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

```
/*
 * Copyright (c) 1992-1997 Rational Software Corp.
 ...
 * This is a test program used in Purifying Hello World
 */

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

static char *helloWorld = "Hello, World";

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

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

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

Instrumenting a program

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

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

output

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
```

IRIX

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

```
% purify a.out
```

IRIX

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:

Solaris HPUX

```
% a.out
```

IRIX

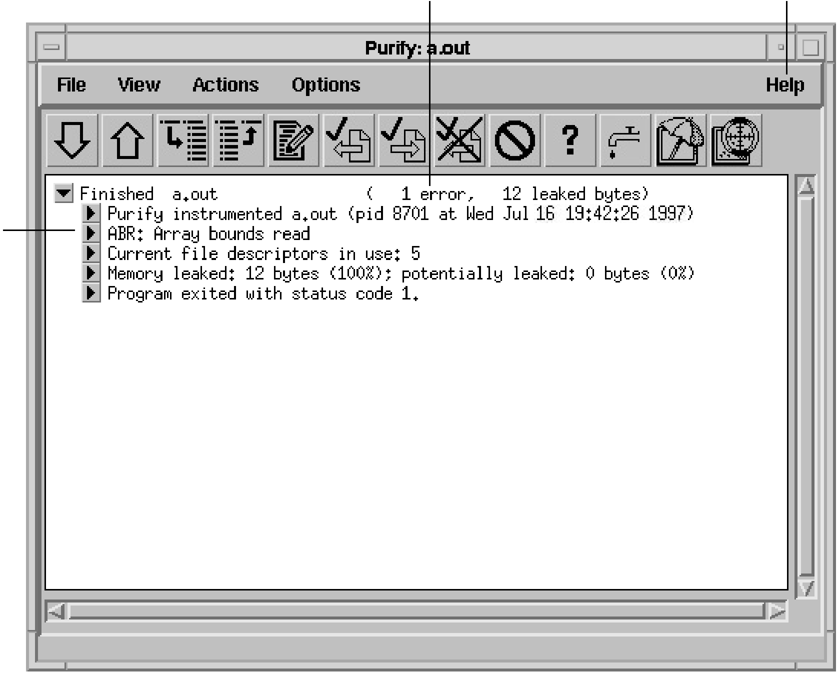
On IRIX, if you use `purify` on the executable instead of on the compile/link line, type:

```
% 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



```
Finished  a.out              (   1 error,   12 leaked bytes)
  Purify instrumented a.out (pid 8701 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%)
  Program exited with status code 1.
```

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

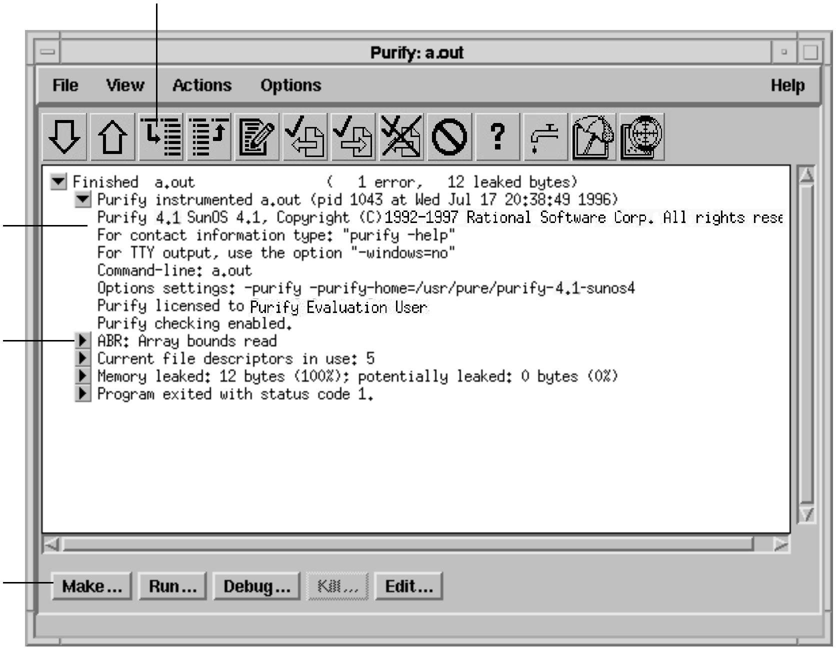
For a description of a message, right click the message, then select **Explain message** from the pop-up menu

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.

Select one or more messages in the Viewer, then click to expand the messages



```
▼ Finished  a.out             (   1 error,   12 leaked bytes)
  ▼ Purify instrumented a.out (pid 1043 at Wed Jul 17 20:38:49 1996)
    Purify 4.1 SunOS 4.1, Copyright (C)1992-1997 Rational Software Corp. All rights rese
    For contact information type: "purify -help"
    For TTY output, use the option "-windows=no"
    Command-line: a.out
    Options settings: -purify -purify-home=/usr/pure/purify-4.1-sunos4
    Purify licensed to Purify Evaluation User
    Purify checking enabled.
  ▶ ABR: Array bounds read
  ▶ Current file descriptors in use: 5
  ▶ Memory leaked: 12 bytes (100%); potentially leaked: 0 bytes (0%)
  ▶ Program exited with status code 1.
```

The configuration message shows the execution process ID (`pid`) and the Purify options used

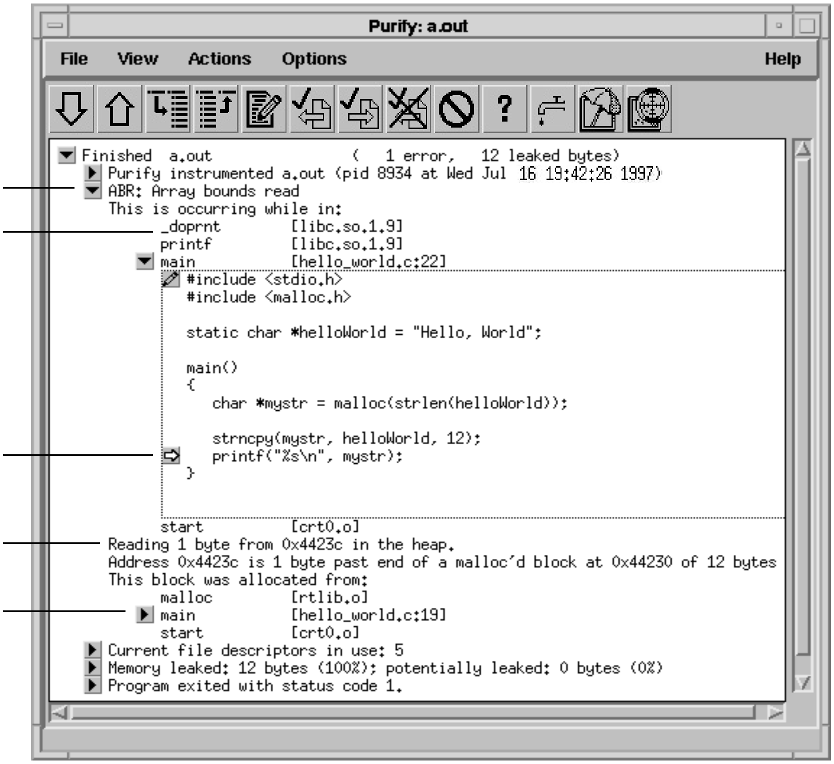
Click to expand a message or item

You can use the program controls to run a debugging cycle. To display them, select **View > Program Controls**

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.



```
▼ 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.

IRIX

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.

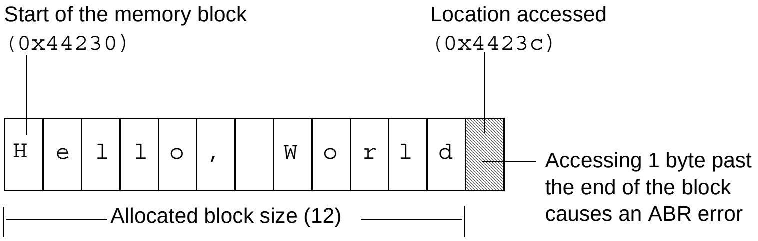
Understanding the cause of the error

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

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

Purify reports that the ABR error occurs here

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.


Start of the memory block
(0x44230)
Location accessed
(0x4423c)
H e l l o ,   W o r l d
Accessing 1 byte past the end of the block causes an ABR error
Allocated block size (12)

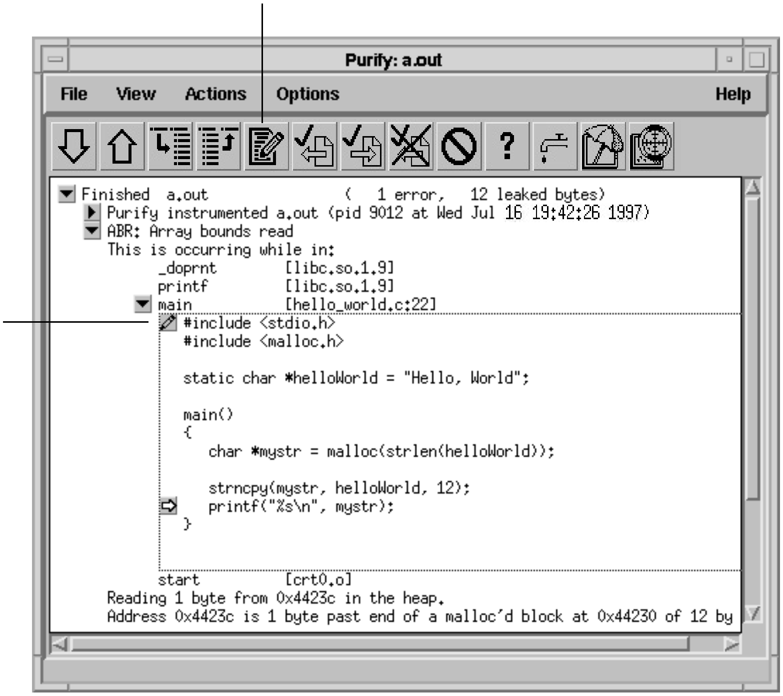
For more information, see *How Purify finds memory-access errors* on page 40.

Correcting the ABR error

To correct this ABR error:

1 Click the Edit tool to open an editor.

Click to edit the source code



```
Finished  a.out            (   1 error,   12 leaked bytes)
  Purify instrumented a.out (pid 9012 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 by
```

Or click here to edit the source code

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

2 Change lines 19 and 21 as follows:

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

Finding leaked memory

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

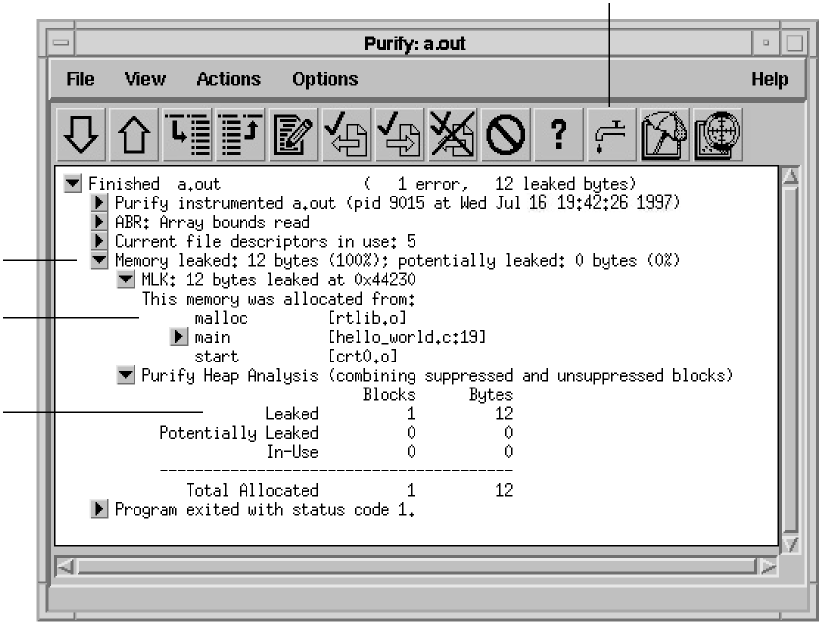
Note: When you run longer-running instrumented programs, you can click the New Leaks tool to generate a new leaks summary while the program is running.

1 Expand the memory-leaked summary for Hello World.

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

2 Expand the MLK message.

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



```
▼ 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.
```

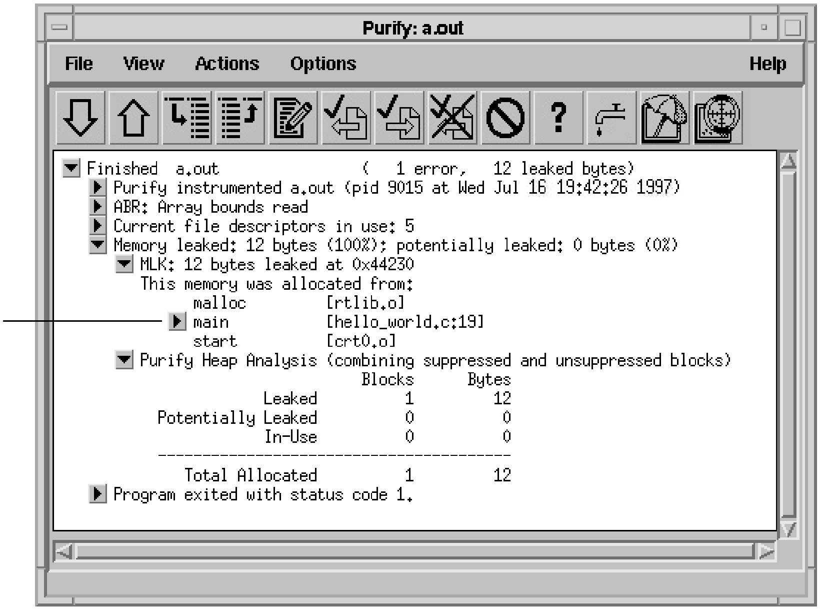
The memory-leaked summary reports 12 bytes of leaked memory

The call chain shows how the leaked memory was allocated

Memory analysis by category

Correcting the MLK error

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



```
▼ 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.
```

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:

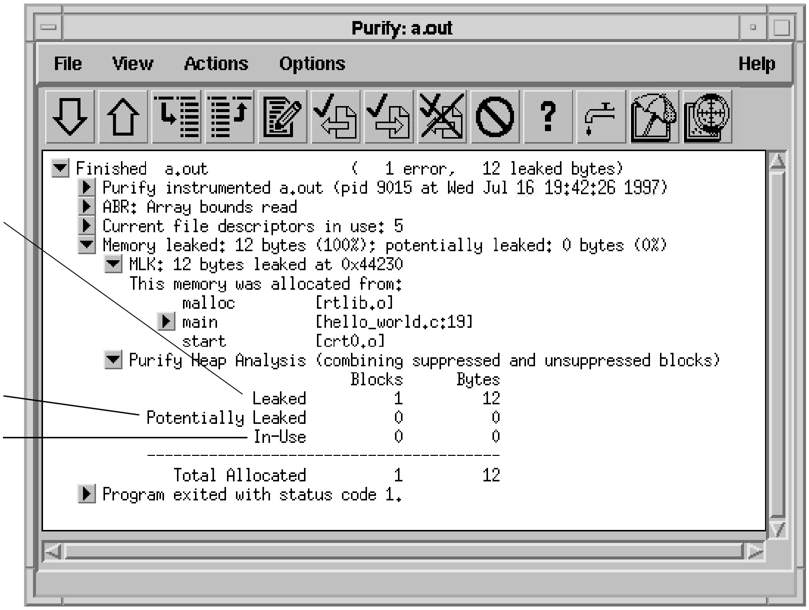
1 Click the Edit tool to open an editor.

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



```
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.
```

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

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

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

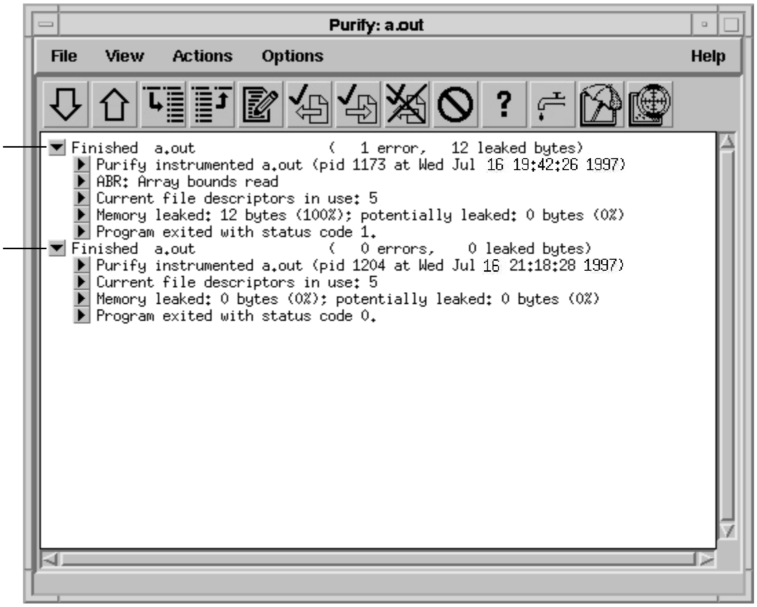
The exit status message provides information about:

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

Comparing program runs

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

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



```
▼ Finished  a.out              (   1 error,   12 leaked bytes)
   ▶ Purify instrumented a.out (pid 1173 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%)
   ▶ Program exited with status code 1.
▼ Finished  a.out              (   0 errors,    0 leaked bytes)
   ▶ Purify instrumented a.out (pid 1204 at Wed Jul 16 21:18:28 1997)
   ▶ Current file descriptors in use: 5
   ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)
   ▶ Program exited with status code 0.
```

In the previous run, Purify reported one error and twelve leaked bytes

In the new run, Purify reports no errors and no memory leaks

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

Suppressing Purify messages

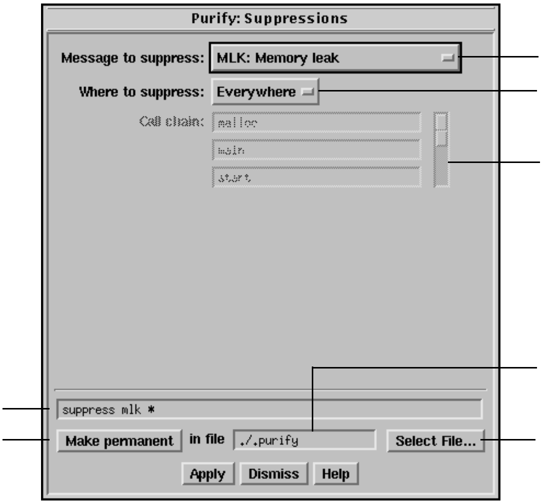
A large program can generate hundreds of error messages. To quickly focus on the most critical ones, you can suppress the less critical messages based on their type and source. For example, you might want to hide all informational messages, or hide all messages that originate in a specific file.

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.



Purify: Suppressions

Message to suppress: MLK: Memory leak

Where to suppress: Everywhere

Call chain: malloc, main, start

```
suppress mlk *
```

in file ./.purify

Select a message to suppress

Select where to suppress the message

Control the depth of the call-chain match

Purify saves suppressions in `.purify` files

You can save the suppression directive to another `.purify` file

The suppression directive

Click to make a suppression permanent

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

Solaris HPUX

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 *Using Rational PureCoverage* on page 43.

Purify API functions

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

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

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

Commonly used functions	**Description**
`int   purify_describe (char *addr)`	Prints specific details about memory
`int   purify_is_running (void)`	Returns `"TRUE"` if the program is instrumented

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

Build-time options

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

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

Commonly used build-time options	Default
`-always-use-cache-dir` Forces all instrumented object files to be written to the global cache directory	`no`
`-cache-dir` Specifies the global directory where Purify caches instrumented object files	`<purifyhome>/cache`
`-collector` Specifies the collect program to handle static constructors (for use with `gcc`, `g++`)	none
`-ignore-runtime-environment` Prevents the run-time Purify environment from overriding the option values used in building the program	`no`

Commonly used build-time options	Default
-linker Sets the alternative linker to build the executables instead of the system default	system-dependent
-print-home-dir Prints the name of the directory where Purify is installed, then exits	

Conversion characters for filenames

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

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

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

Commonly used run-time options	Default
-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 37.*

Purify messages

Purify reports the following messages:

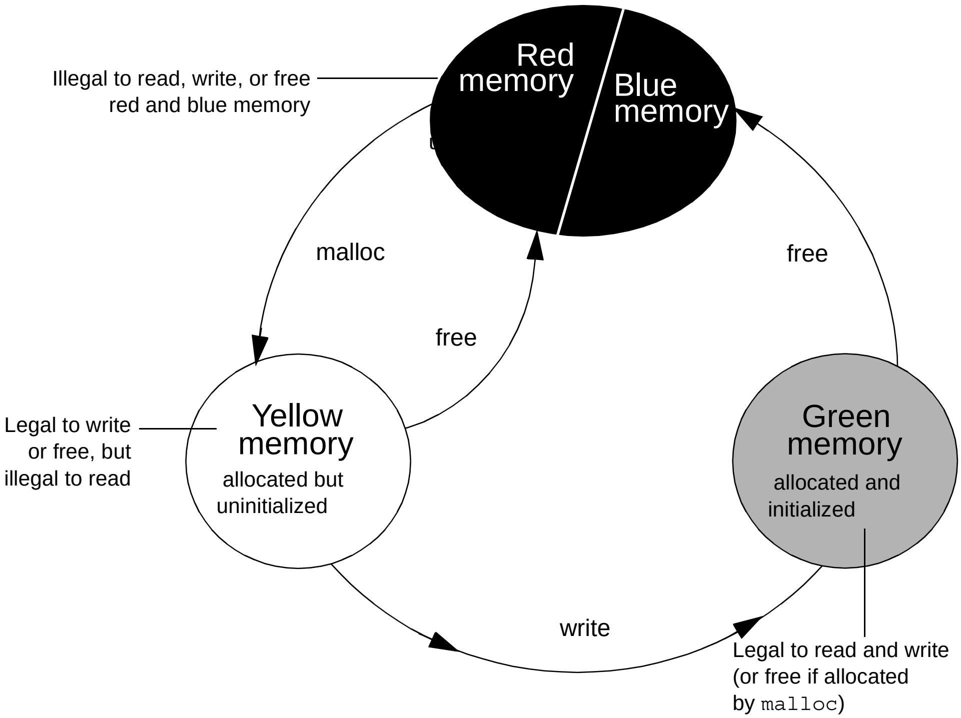
Message	Description	Severity*	Message	Description	Severity*
ABR	Array Bounds Read	W	**NPR**	Null Pointer Read	F
ABW	Array Bounds Write	C	**NPW**	Null Pointer Write	F
BRK	Misuse of Brk or Sbrk	C	**PAR**	Bad Parameter	W
BSR	Beyond Stack Read	W	**PLK**	Potential Leak	W
BSW	Beyond Stack Write	W	**PMR**	Partial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

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

How Purify finds memory-access errors

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

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



Illegal to read, write, or free red and blue memory
Red memory
Blue memory
malloc
free
free
Legal to write or free, but illegal to read
Yellow memory
allocated but uninitialized
Green memory
allocated and initialized
write
Legal to read and write (or free if allocated by malloc)

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

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

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

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

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

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

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

How Purify checks statically allocated memory

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

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

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

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

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

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

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

Using Rational PureCoverage 3

Rational PureCoverage: What it does

During the development process, software changes daily, sometimes hourly. Unfortunately, test suites do not always keep pace. Rational® 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 online help system.

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. If PureCoverage has been installed on your system as a component of Rational PurifyPlus, the Rational directory contains the files purifyplus_setup.csh and purifyplus_setup.sh. Source the file that is appropriate to your shell to license PureCoverage for your use.
2. 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* .
   ```

3. 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
```

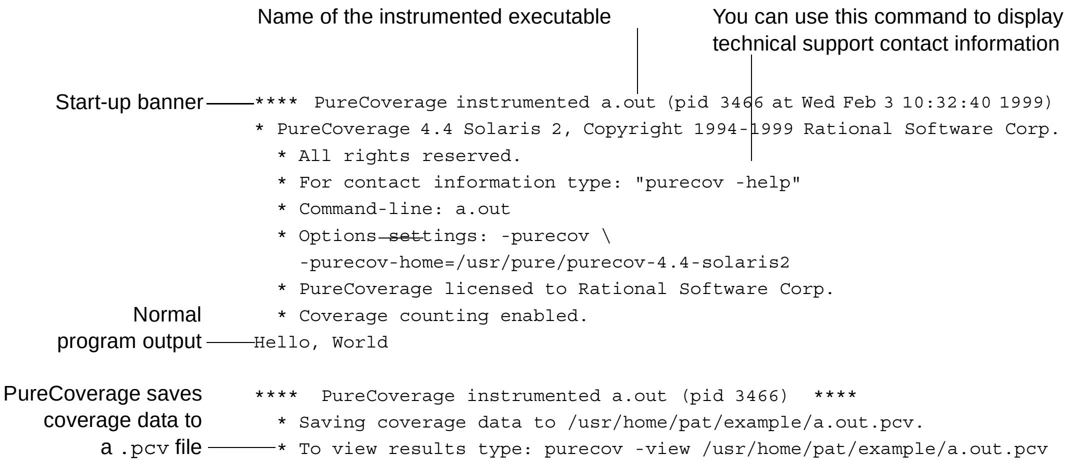
Note: When you compile and link in separate stages, add `purecov` only to the link line.

Running the instrumented program

Run the instrumented Hello World program:

```
% a.out
```

PureCoverage displays the following:



```
**** 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 ~~set~~tings: -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.
  * To view results type: purecov -view /usr/home/pat/example/a.out.pcv
```

Name of the instrumented executable
You can use this command to display technical support contact information
Start-up banner
Normal program output
PureCoverage saves coverage data to a .pcv file

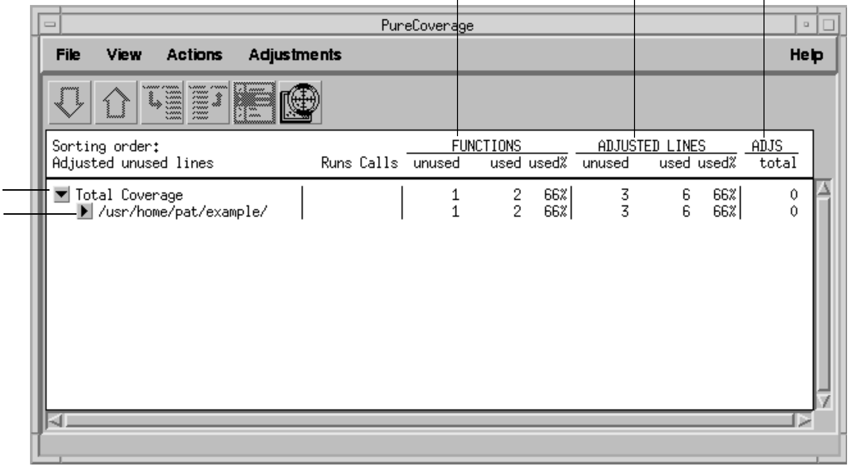
The a.out program produces its normal output, just as if it were not instrumented. When the program completes execution, PureCoverage writes coverage information for the session to the file a.out.pcv. Each time the program runs, PureCoverage updates this file with additional coverage data.

Displaying coverage data

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

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

This displays the PureCoverage Viewer.



```
Sorting order:                               FUNCTIONS             ADJUSTED LINES        ADJS
Adjusted unused lines          Runs Calls  unused    used used%  unused    used used%   total
Total Coverage                                  1       2   66%       3       6   66%       0
/usr/home/pat/example/                          1       2   66%       3       6   66%       0
```

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

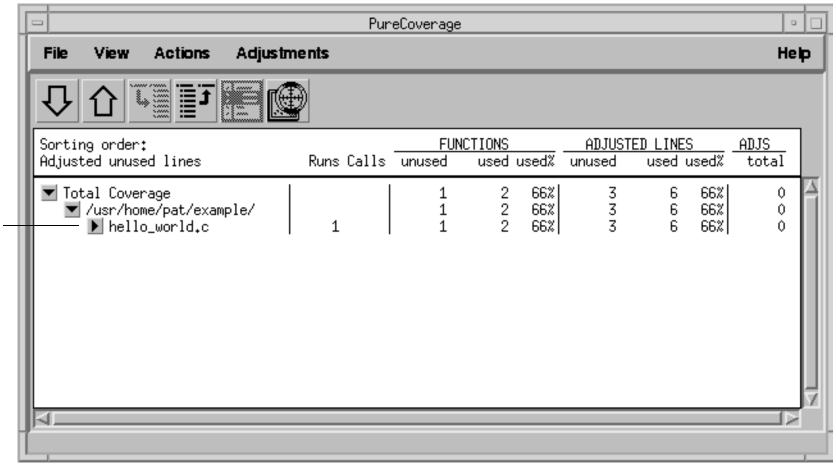
Information for the source directory

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.



Sorting order: Adjusted unused lines	Runs	Calls	FUNCTIONS unused	 used	 used%	ADJUSTED LINES unused	 used	 used%	ADJS total
▼ Total Coverage			1	2	66%	3	6	66%	0
▼ /usr/home/pat/example/			1	2	66%	3	6	66%	0
▶ hello_world.c	1		1	2	66%	3	6	66%	0

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.

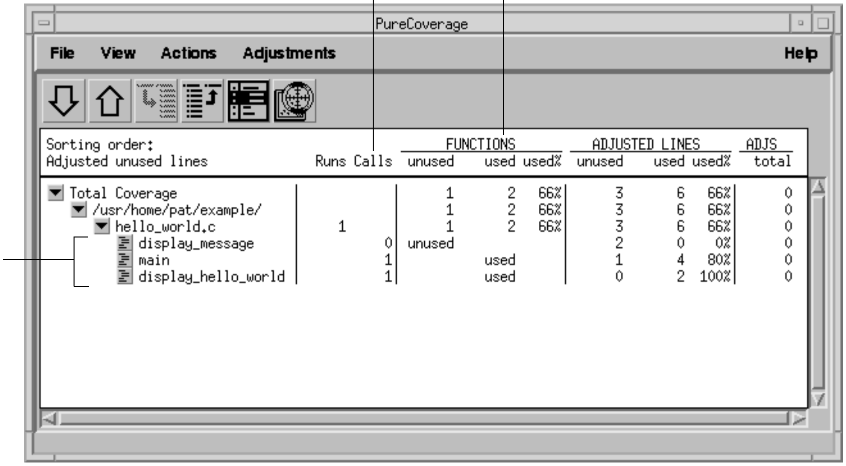
Examining function-level detail

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



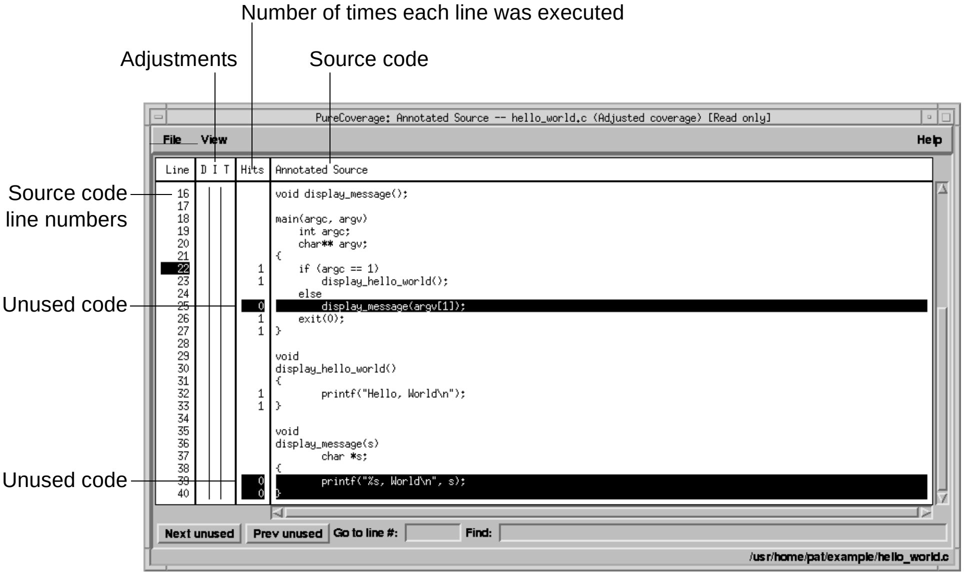
Sorting order: Adjusted unused lines	Runs	Calls	FUNCTIONS unused	 used	 used%	ADJUSTED LINES unused	 used	 used%	ADJS total
Total Coverage			1	2	66%	3	6	66%	0
/usr/home/pat/example/			1	2	66%	3	6	66%	0
hello_world.c	1		1	2	66%	3	6	66%	0
display_message		0	unused			2	0	0%	0
main		1		used		1	4	80%	0
display_hello_world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

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.


Number of times each line was executed
Adjustments
Source code
Source code line numbers
Unused code
Unused code
PureCoverage: Annotated Source -- hello_world.c (Adjusted coverage) [Read only]
File
View
Help
Next unused
Prev unused
Go to line #:
Find:
/usr/home/pat/example/hello_world.c

```
Line  D I T  Hits  Annotated Source
 16                void display_message();
 17
 18                main(argc, argv)
 19                    int argc;
 20                    char** argv;
 21                {
 22             1      if (argc == 1)
 23             1          display_hello_world();
 24                    else
 25             0          display_message(argv[1]);
 26             1      exit(0);
 27             1  }
 28
 29                void
 30                display_hello_world()
 31                {
 32             1          printf("Hello, World\n");
 33             1  }
 34
 35                void
 36                display_message(s)
 37                        char *s;
 38                {
 39             0          printf("%s, World\n", s);
 40             0  }
```

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:

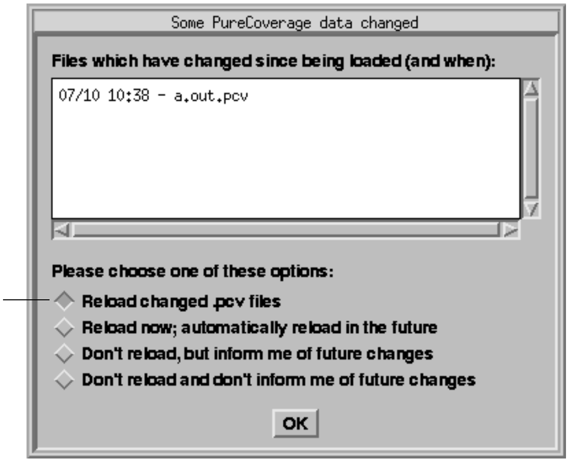
```
% a.out Goodbye
```

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**.



Some PureCoverage data changed

Files which have changed since being loaded (and when):

07/10 10:38 - a.out.pcv

Please choose one of these options:

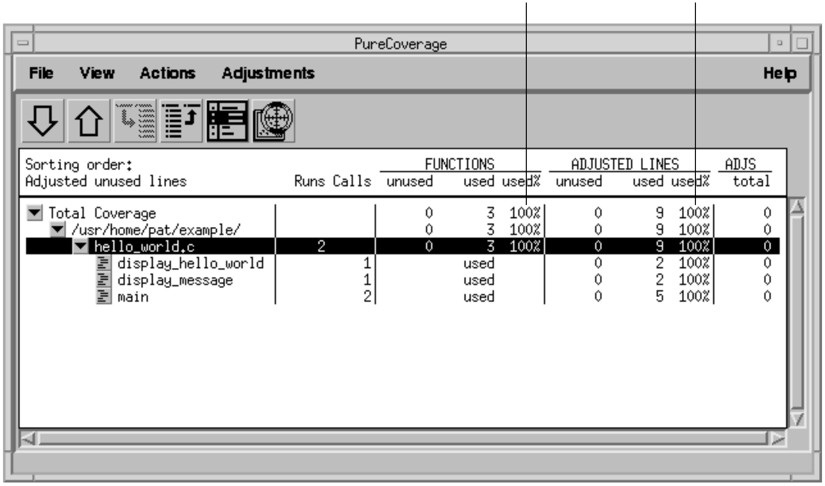
◇ Reload changed .pcv files
◇ Reload now; automatically reload in the future
◇ Don't reload, but inform me of future changes
◇ Don't reload and don't inform me of future changes

Reload the changed `a.out.pcv` file

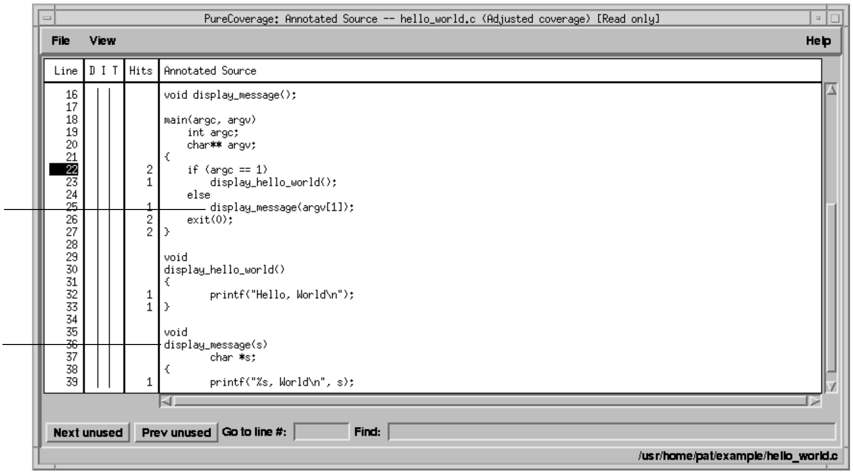
Note: This dialog appears only if the PureCoverage Viewer is open when you run the program.

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

Function and line coverage is now 100%



```
Sorting order:                                     FUNCTIONS          ADJUSTED LINES      ADJS
Adjusted unused lines          Runs Calls   unused   used used%   unused   used used%    total
▼ Total Coverage                                 0      3  100%        0      9  100%        0
  ▼ /usr/home/pat/example/                       0      3  100%        0      9  100%        0
    ▼ hello_world.c               2              0      3  100%        0      9  100%        0
        display_hello_world             1            used               0      2  100%        0
        display_message                 1            used               0      2  100%        0
        main                            2            used               0      5  100%        0
```



PureCoverage: Annotated Source -- hello_world.c (Adjusted coverage) [Read only]
File View Help
Next unused
Prev unused
Go to line #:
Find:
/usr/home/pat/example/hello_world.c

```
Line  D I T  Hits  Annotated Source
  16                void display_message();
  17
  18                main(argc, argv)
  19                    int argc;
  20                    char** argv;
  21                {
  22            2       if (argc == 1)
  23            1           display_hello_world();
  24                    else
  25            1           display_message(argv[1]);
  26            2       exit(0);
  27            2   }
  28
  29                void
  30                display_hello_world()
  31                {
  32            1           printf("Hello, World\n");
  33            1   }
  34
  35                void
  36                display_message(s)
  37                        char *s;
  38                {
  39            1           printf("%s, World\n", s);
```

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**.

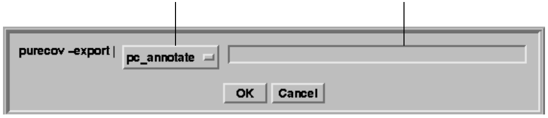
Using report scripts

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...
```

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

Run-time options

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

† *Can use the conversion characters listed on page 37.*

Analysis-time options

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

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

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

Analysis-time mode options

Command-line syntax:

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

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

Using Rational Quantify 4

Rational Quantify: What it does

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.

Rational® 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 online help system.

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

Note: If Quantify has been installed on your system as a component of Rational PurifyPlus, the Rational directory contains the files `purifyplus_setup.csh` and `purifyplus_setup.sh`. Source the file that is appropriate to your shell to license Quantify for your use.

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.
Hello, World.

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

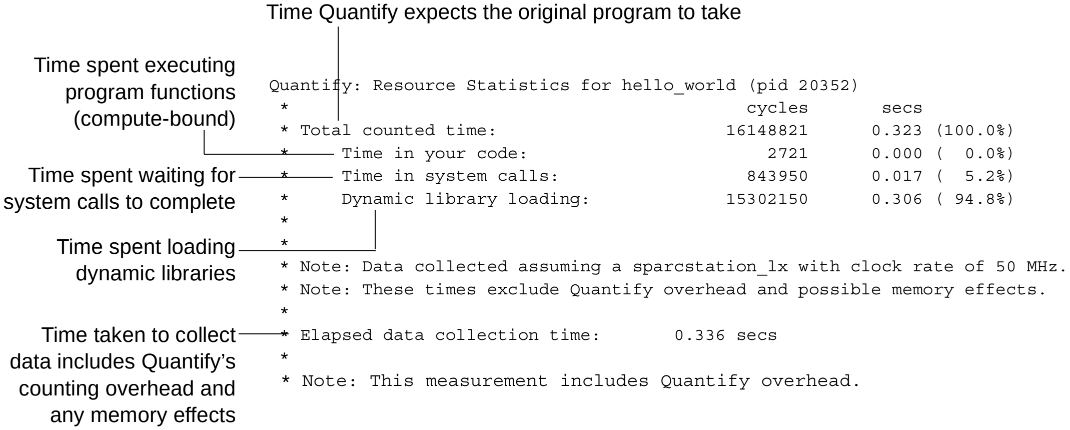
Program output

Data transmission

*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.

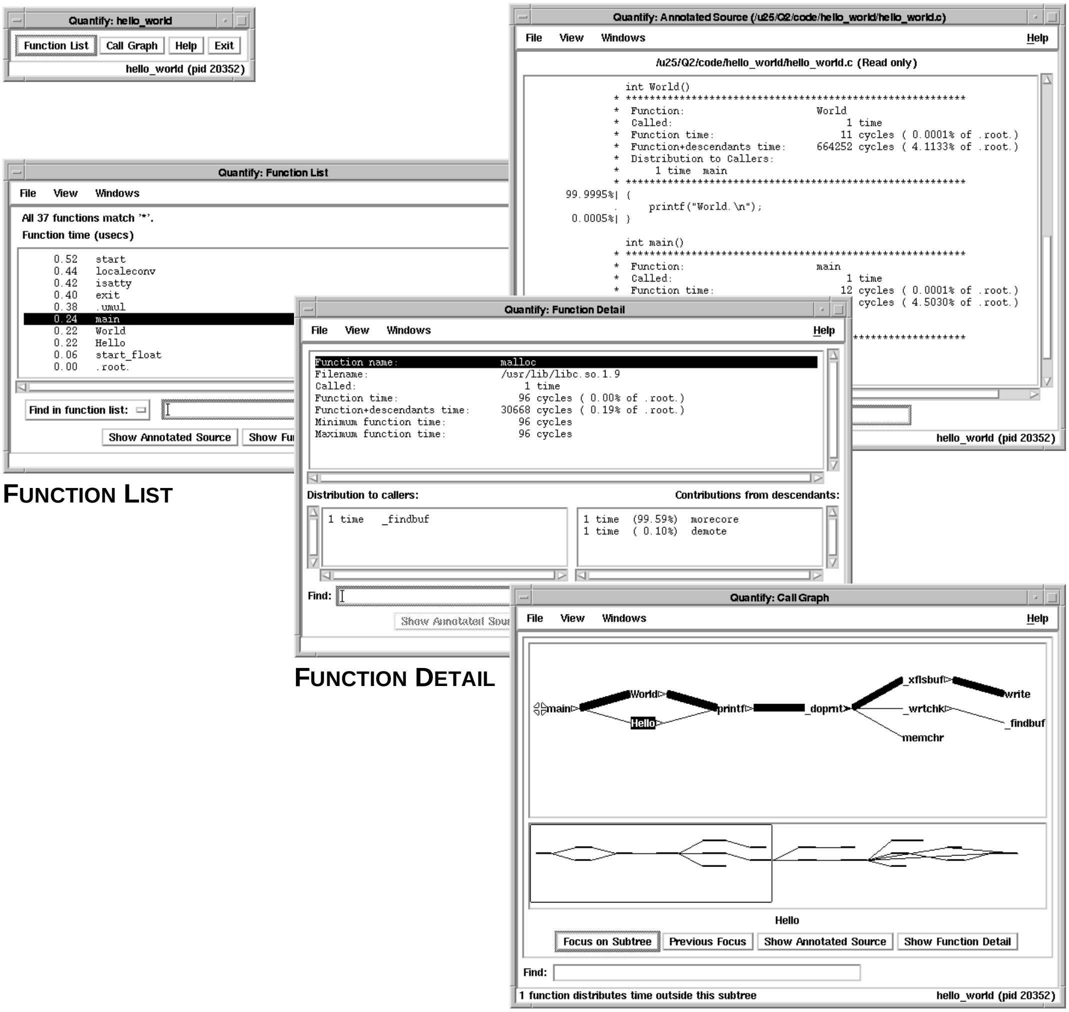


```
Quantify: Resource Statistics for hello_world (pid 20352)
 *                                            cycles       secs
 * Total counted time:                      16148821      0.323 (100.0%)
 *      Time in your code:                      2721      0.000 (  0.0%)
 *      Time in system calls:                 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.
```

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

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

hello_world (pid 20352)

ANNOTATED SOURCE

Quantify: Annotated Source (/u25/Q2/code/hello_world/hello_world.c)
File
View
Windows
Help

/u25/Q2/code/hello_world/hello_world.c (Read only)

```
              int World()
            * ***********************************************************
            *  Function:                         World
            *  Called:                               1 time
            *  Function time:                       11 cycles ( 0.0001% of .root.)
            *  Function+descendants time:       664252 cycles ( 4.1133% of .root.)
            *  Distribution to Callers:
            *      1 time  main
            * ***********************************************************
   99.9995%| {
           .     printf("World.\n");
    0.0005%| }

              int main()
            * ***********************************************************
            *  Function:                         main
            *  Called:                               1 time
            *  Function time:                       12 cycles ( 0.0001% of .root.)
                                                       cycles ( 4.5030% of .root.)

                                                ********************
```

hello_world (pid 20352)

All 37 functions match '*'.
Function time (usecs)

```
0.52   start
0.44   localeconv
0.42   isatty
0.40   exit
0.38   .umul
0.24   main
0.22   World
0.22   Hello
0.06   start_float
0.00   .root.
```

FUNCTION LIST

```
Function name:                malloc
Filename:                     /usr/lib/libc.so.1.9
Called:                          1 time
Function time:                  96 cycles ( 0.00% of .root.)
Function+descendants time:   30668 cycles ( 0.19% of .root.)
Minimum function time:          96 cycles
Maximum function time:          96 cycles
```

Distribution to callers:

```
1 time   _findbuf
```

Contributions from descendants:

```
1 time  (99.59%)  morecore
1 time  ( 0.10%)  demote
```

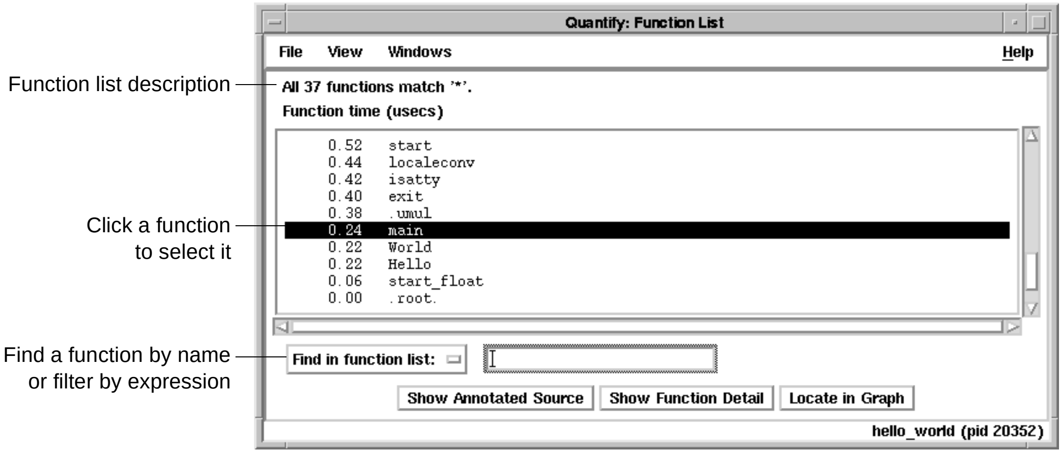
FUNCTION DETAIL

1 function distributes time outside this subtree   hello_world (pid 20352)

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.



Function list description

Click a function to select it

Find a function by name or filter by expression

All 37 functions match '*'.

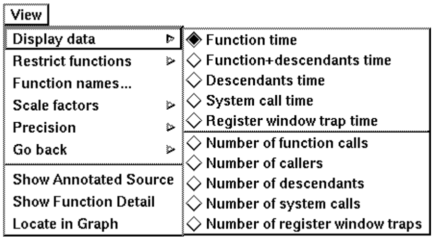
Function time (usecs)

```
      0.52   start
      0.44   localeconv
      0.42   isatty
      0.40   exit
      0.38   .umul
      0.24   main
      0.22   World
      0.22   Hello
      0.06   start_float
      0.00   .root.
```

hello_world (pid 20352)

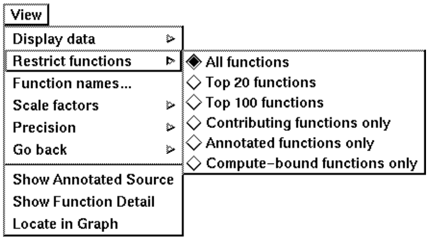
Sorting the function list

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



Restricting functions

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



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.

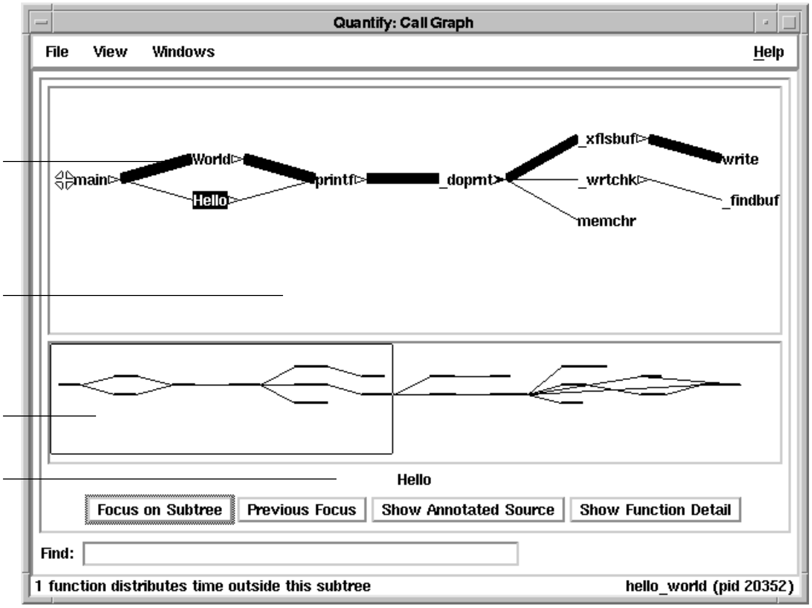
The call graph helps you understand the calling structure of your program and the major call paths that contributed to the total time of the run. Using the call graph, you can quickly discover the sources of bottlenecks.

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



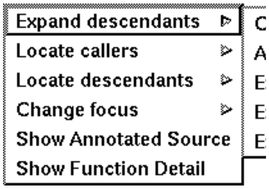
1 function distributes time outside this subtree

hello_world (pid 20352)

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

Using the pop-up menu

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

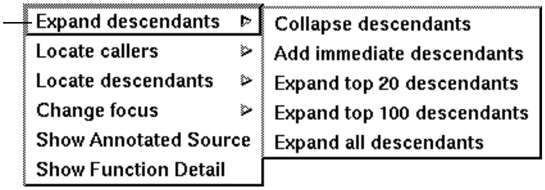


You can use the pop-up menu to:

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

Expanding and collapsing descendants

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



Select to expand or collapse descendant subtrees

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

The Function Detail window

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

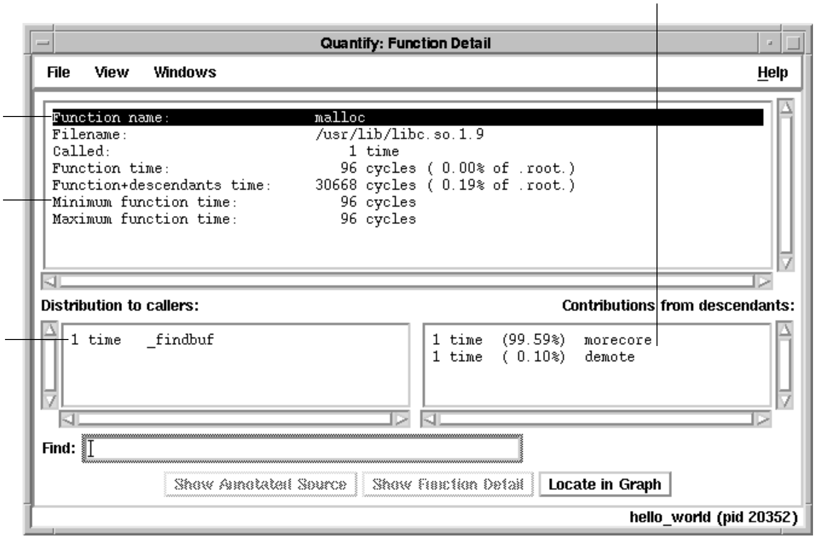
For each function, Quantify reports both the time spent in the function's own code (its *function* time) and the time spent in all the functions that it called (its *descendants* time). Quantify distributes this accumulated *function+descendants* time to the function's immediate caller.

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

All the data collected for `malloc`

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

The functions that called `malloc`



```
Function name:                 malloc
Filename:                      /usr/lib/libc.so.1.9
Called:                           1 time
Function time:                   96 cycles ( 0.00% of .root.)
Function+descendants time:    30668 cycles ( 0.19% of .root.)
Minimum function time:           96 cycles
Maximum function time:           96 cycles
```

Distribution to callers:

```
1 time   _findbuf
```

Contributions from descendants:

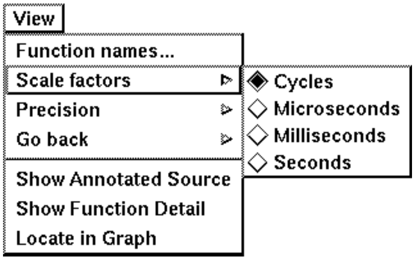
```
1 time   (99.59%)  morecore
1 time   ( 0.10%)  demote
```

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

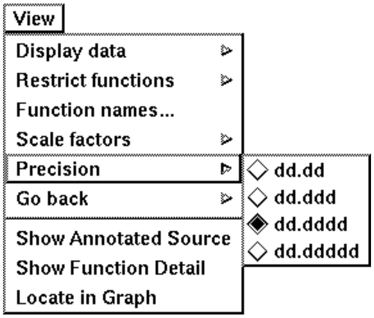
The function time and the function+descendants time are shown as a percentage of the total accumulated time for the entire run. These percentages help you understand how this function's computation contributed to the overall time of the run. These times correspond to the thickness of the lines in the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

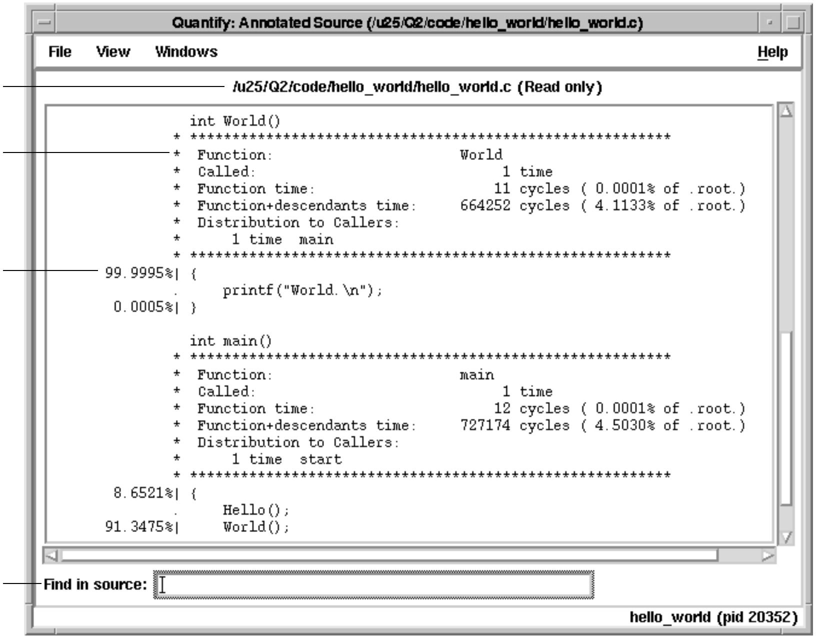
To save the current function detail display to a file, select **File > Save current function detail as**.

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.



Quantify: Annotated Source (/u25/Q2/code/hello_world/hello_world.c)
File
View
Windows
Help
Find in source:

/u25/Q2/code/hello_world/hello_world.c (Read only)

```
                int World()
              * **********************************************************
              *  Function:                         World
              *  Called:                                1 time
              *  Function time:                        11 cycles ( 0.0001% of .root.)
              *  Function+descendants time:     664252 cycles ( 4.1133% of .root.)
              *  Distribution to Callers:
              *      1 time  main
              * **********************************************************
    99.9995%| {
              .     printf("World.\n");
     0.0005%| }

                int main()
              * **********************************************************
              *  Function:                         main
              *  Called:                                1 time
              *  Function time:                        12 cycles ( 0.0001% of .root.)
              *  Function+descendants time:     727174 cycles ( 4.5030% of .root.)
              *  Distribution to Callers:
              *      1 time  start
              * **********************************************************
     8.6521%| {
              .     Hello();
    91.3475%|     World();
```

hello_world (pid 20352)

Source file

Function summary

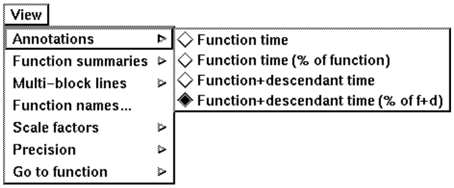
Annotations show how *function+descendants* time was distributed over its source lines

Find text in the source code

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.

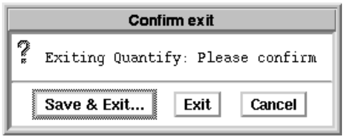
Changing annotations

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.


Confirm exit

Exiting Quantify: Please confirm

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

Commonly used build-time options	Default
-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