Purify User's Guide

Contents

2 Finding Errors in Hello World

3 Memory Access Errors

8   Setting Watchpoints

9   Custom Memory Managers

10   Purify Messages Reference

11   Using Purify Options and API Functions

12 Purify Options and API Reference

13 Common Questions

Purify Quick Reference

Index

x

Welcome to Purify

This guide documents the features and capabilities of Purify release 4.0. It can help you quickly master the basics of using Purify and move on to using Purify's more specialized features.

Getting started

Chapter 1, "Introducing Purify" provides an overview of how and when to use Purify in order to get the most out of it, including how to use Purify with other Pure Software products.

It also contains a discussion of the importance of finding various types of memory access errors and memory leaks.

Mastering the basics

Three tutorial chapters help you begin successfully using Purify:

- Chapter 2, "Finding Errors in Hello World" shows you the basics of how to use Purify and the messages it generates.
- Chapter 3, "Memory Access Errors" explains how Purify finds memory access errors and shows you how to correct them.
- Chapter 4, "Memory Leaks" describes how Purify reports memory leaks and shows you how correct them.

Learning to use special features

These chapters help you take advantage of Purify's special features:

- Chapter 5, "Analyzing File Descriptors" describes how Purify reports the file descriptors that are open when your application exits.

- Chapter 6, "Customizing Purify" explains how to customize Purify's Viewer messages and scripts, save output to log files and *view files*, and enable just-in-time debugging.
- Chapter 7, "Suppressing Purify Messages" describes how to prevent Purify messages from being displayed.
- Chapter 8, "Setting Watchpoints" describes how to use watchpoints to monitor memory reads, writes, allocations, and frees.
- Chapter 9, "Custom Memory Managers" describes Purify's support for special-purpose or custom memory allocators.

Using the reference chapters

These reference chapters provide a complete resource for your ongoing use of Purify.

- Chapter 10, "Purify Messages Reference" describes each of the messages generated by Purify.
- Chapter 11, "Using Purify Options and API Functions" explains how to specify Purify options and API functions.
- Chapter 12, "Purify Options and API Reference" provides a complete reference of all Purify options and API functions.
- Chapter 13, "Common Questions" contains answers to the most frequently asked questions about Purify.

Using online Help

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

Conventions used in this guide

- `<purifyhome>` refers to the directory where Purify is installed. To find the Purify directory on your system, type:

  ```
  % purify -printhomedir
  ```

- `Courier` font indicates source code, program names or output, file names, and commands that you enter.

- Angle brackets `< >` indicate variables.

- *Italics* introduce new terms and show emphasis.

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

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

- This icon appears next to instructions for the HP-UX operating system.

- This icon appears next to instructions for the Silicon Graphics IRIX operating system.

Displaying the release notes

The Purify README file is located in the `<purifyhome>` directory. You can open it from the Purify Viewer by selecting **Release Notes** from the Help menu. The README file contains the latest information about this release of Purify, including hardware and software supported, and notes about specific operating systems.

Installing Purify

For information about licensing and installing Purify, refer to the *Installation & Licensing Guide*, part number 300-00001-02.

Contacting technical support

If you have a technical problem and you can't find the solution in this guide, contact your nearest Pure Software Technical Support Center. See the back cover of this guide for addresses and phone numbers of Pure Software Technical Support Centers.

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

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

1

Introducing Purify

In the world of C and C++ software development, no tool exists that can prevent you from introducing memory-related bugs into your application. But there is a tool that can help you locate and resolve these bugs easily and quickly, minimizing their impact on your budget, schedule, and customers. That tool is Purify.

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

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

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

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

When to use Purify

The key to delivering quality software applications is to use Purify consistently, right from the start. As soon as your code is ready to run, you can benefit from using Purify.

Begin coding		
As you code	Resolve memory-related errors at the best and least expensive time: when the code is fresh in your mind.	
Acceptance tests	Validate code supplied by third-party vendors or other groups before you incorporate it into your business-critical code.	
Code check-in	Reduce the risk that bugs in your code might impact other code modules, causing team members to lose valuable time.	
Nightly builds	Verify that modules work together, and expose code dependencies and collisions.	
Project milestones	Uncover bugs and validate product quality during alpha, beta, and other milestones.	
Release product		

Use Purify throughout your development cycle for progressively cleaner code and a more solid product as your project advances.

Starting to use Purify

Purify is easy to use. You just add `purify` to your link line. For example:

```
% purify cc -g <myprogram>.o
```

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

```
% purify cc -g <myprogram>.c
```

Purify uses Object Code Insertion (OCI) technology to instrument a copy of your object code, inserting checking instructions before every memory operation. (On IRIX, Purify instruments the executable file, then saves the file under a new name.)

When you run the instrumented program, Purify reports run-time errors and memory leaks in the Purify Viewer.

Click here to expand one or more selected messages

Total number of memory access errors
Total amount of leaked memory

Click here to expand a message or item

Acronyms summarize the type of error

```
Purify: a.out

File   View   Actions   Options                           Help

▼ Finished  a.out              ( 583 errors,  182 leaked bytes)
  ▶ Purify instrumented a.out (pid 3198 at Wed Jun 26 17:36:17 1996)
  ▶ UMR: Uninitialized memory read (13 times)
  ▶ UMR: Uninitialized memory read (13 times)
  ▶ ABW: Array bounds write (100 times)
  ▶ ABR: Array bounds read (8 times)
  ▶ ABR: Array bounds read (108 times)
  ▶ ABR: Array bounds read (50 times)
  ▶ FMR: Free memory read (50 times)
  ▶ ABR: Array bounds read (50 times)
  ▶ ABW: Array bounds write (50 times)
  ▶ ABR: Array bounds read (4 times)
  ▶ ABR: Array bounds read (50 times)
  ▶ FNH: Freeing non heap memory (87 times)
  ▶ Current file descriptors in use: 5
  ▶ Memory leaked: 182 bytes (37.8%); potentially leaked: 0 bytes (0%)
  ▶ Program exited with status code 0.
```

Purify displays messages in outline format for easy browsing.

Purify's condensed outline format makes it easy for you to scan the messages, quickly identify critical errors, and assess the state of your program.

The number of times the error occurs

The call chain shows the function calls leading to the error, and the line where the error occurs

The exact location of the error in the program

The exact location in memory where the error occurs


```
┌──────────────────────────────────────────────────────────────────────┐
│ ▭                         Purify: a.out                          ▫  ▢ │
├──────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                            Help   │
├──────────────────────────────────────────────────────────────────────┤
│  ⬇  ⬆  ⬑  ⬐  ▨  ⊘  ?  ⌐  ⬙  ◉                                         │
├──────────────────────────────────────────────────────────────────────┤
│  ▶ UMR: Uninitialized memory read (13 times)                          │
│  ▶ UMR: Uninitialized memory read (13 times)                          │
│  ▼ ABW: Array bounds write (100 times)                                │
│    This is occurring while in:                                        │
│        strcpy          [rtlib.o]                                      │
│     ▶  putHash          [hash.c:146]                                  │
│     ▶  testPutHash      [testHash.c:84]                               │
│     ▼  main             [testHash.c:210]                              │
│        ▨   fillTestTable(testTable);                                  │
│                                                                       │
│            ht = testMakeHashTable();                                  │
│        ▨   testPutHash(ht, testTable,  0, 100, FALSE);                │
│            testGetHash(ht, testTable,  0, 100, TRUE);                 │
│            testRemHash(ht, testTable,  0,  50, TRUE);                 │
│            testRemHash(ht, testTable,  0,  50, FALSE);                │
│        start           [crt0.o]                                       │
│    Writing 2 bytes to 0x489a0 in the heap (1 byte at 0x489a1 illegal).│
│    Address 0x489a0 is at the end of a malloc'd block at 0x489a0 of 1 byte.│
│    This block was allocated from:                                     │
│        malloc          [rtlib.o]                                      │
│     ▶  putHash          [hash.c:145]                                  │
│     ▶  testPutHash      [testHash.c:84]                               │
│     ▶  main             [testHash.c:210]                              │
│        start           [crt0.o]                                       │
└──────────────────────────────────────────────────────────────────────┘
```

Purify displays exactly the information you need to quickly locate and correct errors.

To make messages easy to scan, Purify displays the first occurrence of a message, with a count of repeated identical occurrences (those with the same error type and call chain). For example, if a single error occurs repeatedly inside a loop, Purify displays an initial message and then simply updates the repeat count.

Getting the most out of Purify

Whether you spend your time deep within a debugger or with a set of test scripts, work alone or as part of a team, Purify can work the way you do.

Customizing Purify

You can customize Purify to suit your own needs. For example, you can:

- Direct Purify output to a binary file instead of displaying it in the Viewer. This is useful when you are running a nightly test suite.

- Customize Purify messages to include additional information that can help you locate errors. For example, you can increase the number of source code lines that are displayed in messages, or include instruction addresses and offsets.

- Suppress Purify messages in order to focus on critical errors. For example, you might want to suppress all messages related to third-party libraries for which you don't have source code, or hide all messages except those in your own code. Suppressing a message affects only the display of information. Purify continues to detect and report all problems, but it doesn't display the suppressed messages in the Viewer.

For more information, see Chapter 6, "Customizing Purify" and Chapter 7, "Suppressing Purify messages."

Using your debugger with Purify

Purify lets you run your Purify'd program directly under your debugger. When Purify finds an error, you can investigate it immediately without rerunning your program separately in your debugger.

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.

For more information, see "Using your debugger to set breakpoints" on page 4-7 and "Enabling JIT debugging" on page 6-11.

Calling Purify's API functions

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

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

For more information about how to use Purify API functions, see Chapter 11, "Using Purify Options and API Functions." For a complete list of Purify API functions, see Chapter 12.

Integrating Purify into makefiles and scripts

You can easily use Purify with existing makefiles, test harnesses, or scripts. For example, in a makefile:

```
hello_world: hello_world.o
      cc -g -o hello_world hello_world.o
```

Just add `purify`, or another target:

```
hello_world.pure: hello_world.o
      purify cc -g -o hello_world.pure hello_world.o
```

Using Purify with other Pure Software products

You can use Purify with other Pure Software products such as PureCoverage, PureLink, PureDDTS, and PureTestExpert.

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

Using Purify with PureCoverage

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

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
```

When you run your program, you see the Purify banner and the PureCoverage banner. Purify reports memory access errors and memory leaks as the program runs. You can examine the test coverage data after the program terminates.

For more information about the order in which Purify applies options, see "Purify option processing" on page 11-4.

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

Using PureLink with Purify and PureCoverage

You can use Purify and PureCoverage with PureLink, Pure Software's incremental linker, to save time building your programs. Specify purelink *first* on the link line. Include all options with the program to which they refer.

To use Purify with PureLink, type:

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

To use Purify with Purelink and PureCoverage, type:

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

Using Purify with PureDDTS

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

Using Purify with PureTestExpert

You can run Purify'd programs with PureTestExpert, Pure Software's test-management tool to automatically get run-time reports. This is especially useful when running regression tests.

Checking for memory errors

Memory errors, such as array-bounds errors, dangling pointers, and uninitialized memory reads, are among the most difficult to detect. The symptoms of incorrect memory use typically occur far from the cause of the error and are unpredictable, so that a program that appears to work correctly really works only by accident.

Memory access errors

When your program writes memory past the bounds of an allocated block, the memory could belong to another data structure in the program, which would become corrupted when it is overwritten. Reading from beyond the memory block might appear less critical because memory is not corrupted. However, the behavior of the program comes to depend on the values accessed, which are unpredictable. If the exact layout of memory is changed, the memory block adjacent to the block for which the reference was intended might be totally different and contain different values.

Purify inserts guard zones around statically and dynamically allocated memory to catch this type of access error. Purify reports an Array Bounds Read (ABR) or an Array Bounds Write (ABW) message at the time it detects the error.

Accessing through dangling pointers

When a dynamically allocated block of memory is freed, the memory is often reallocated to a new data structure in a different part of the program. If a program uses a dangling pointer to read values from a recently-freed memory block, and the freed memory hasn't been reallocated yet, the expected value might still be present. Although the program appears to work, it could fail if the memory allocation pattern changes and the freed memory block is reallocated earlier.

In a threaded program, the reallocation can happen in another thread, in which case the failure becomes dependent upon specific timing issues. For example, the program might fail only on a multi-processor machine where the second processor allocates the memory while the first processor is still accessing it.

Similarly, if a program uses a dangling pointer to write a value to a recently-freed memory block, the program might continue to work. However, if the memory is already being used by another data structure, the write will corrupt that other data structure. The corruption is apparent only later in the run when it's difficult to identify the cause.

Purify tracks freed memory and reports invalid memory accesses as Free Memory Read (FMR) or Free Memory Write (FMW) errors at the time the errors occur.

Uninitialized memory reads

In C and C++, local variables are allocated from memory on the stack at the time the function or block defining the variable is entered. Initially, the variables contain whatever values the stack memory last held, and are considered uninitialized. If your program attempts to use the value of such a variable without first setting it, the value is undefined. Unfortunately, the value is not random, but depends on how that memory was last used. Similarly, memory you get from `malloc` or `new` starts out uninitialized.

Purify tracks new memory blocks as they are allocated and reports any attempt to read or use a value from the block before it's initialized as an Uninitialized Memory Read (UMR) error.

Memory allocation errors

If your program incorrectly uses memory-allocation primitives, it might continue to run in spite of the error. However, in this case you risk corrupted heap data structures and failures at a later point in your program's run.

Purify intercepts all calls to memory allocation API functions such as `malloc`, `new`, `new[]`, `calloc`, `realloc` and related functions, to warn you about their incorrect use. For example, when you use an incorrect function to free memory, such as calling `free` on memory obtained from `new`, Purify generates a Freeing Mismatched Memory (FMM) message.

Memory leaks

Leaked memory is memory that is allocated but never freed, and for which no pointers are accessible. Although these blocks of memory can't be used again or freed, they still occupy address space. Because leaked memory blocks are typically scattered throughout the heap, the address space becomes fragmented. The memory leaks gradually affect the performance of the program, and can eventually cause the program to fail from lack of memory.

Purify identifies true memory leaks by searching the entire address space looking for allocated memory to which there are no pointers. This technique enables Purify to detect a few leaked blocks out of the many blocks in use. This precision is critical, as a few bytes leaked can be easily missed amid the megabytes of allocated data in use. With Purify, even short test cases can be valuable in finding memory leaks.

Errors in third-party code and libraries

When you run an instrumented program, Purify reports all problems that it detects in the third-party libraries that your program uses. Although you can't edit this code to fix the problems, there are compelling reasons to review the messages that Purify reports about the code.

The reliability and quality of your application depend on the third-party code you include in it. When Purify detects errors or warnings in third-party code, you can develop a workaround or use an alternative product from another vendor. You can also request that your vendor fix the problems (or even Purify its

product!), to help ensure that the components you build into your application meet your high standards.

It's possible that your own code is causing error reports to appear in the third-party code. Unless you check the third-party code, you can't uncover the errors in your code. For example, if your code allocates an undersized buffer and then passes the buffer into a third-party routine to receive an argument, the buffer overwrite occurs within the third-party code. Purify detects these errors.

2

Finding Errors in Hello World

This chapter uses the Hello World program provided with Purify to show you how to:

- Build and run a Purify'd program
- Analyze messages
- Correct an Array Bounds Read (ABR) error
- Understand the memory leaked summary
- Rerun a Purify'd program

The examples in this chapter are built and run on a SunOS 4.1 system. You might see some differences on Solaris 2, HP-UX, and IRIX systems.

Note: In order to open the Viewer, Purify must be able to connect to an X Window display. Set your DISPLAY environment variable before running Purify. If you are *not* running on an X display, or if Purify is unable to make a connection to the display, Purify generates text output.

Before you start

Before you start, you need to copy the Hello World program from the Purify installation directory to a new directory:

1 Create a new working directory, then go to that directory:

```
% mkdir /usr/home/chris/pwork
% cd /usr/home/chris/pwork
```

2 Copy the `hello_world.c` program from `<purifyhome>/example` to the new working directory:

```
% 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. At first glance there are no obvious errors, yet the program contains errors that you can quickly identify with Purify:

```
 1  /*
 2   * Copyright (c) 1992-1995 Pure Software.
    ...
 9   * This is a test program used in Purifying Hello World.
10   */
11
12  #include <stdio.h>
13  #include <malloc.h>
14
15  static char *helloWorld = "Hello, World";
16
17  main()
18  {
19      char *mystr = malloc(strlen(helloWorld));
20
21      strncpy(mystr, helloWorld, 12);
22      printf("%s\n", mystr);
23  }
```

Building a Purify'd program

1 Compile and link the Hello World program using the debugging option –g:

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

Note: If you compile your code without the -g option, Purify reports only function names and object file names. It does not report line numbers, source filenames, or local variable names.

2 Run the program and verify that it produces the expected output:

```
% a.out
```

output ——— `Hello, World`

`%`

3 Add `purify` in front of the compile/link command line:

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

IRIX   On IRIX, you can add `purify` in front of the compile/link command line, or 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 manual. 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 a Purify'd program

Run the instrumented Hello World program:

Sun OS4 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 opens the Purify Viewer. Notice that the Hello World program starts, runs, and exits. The error does not cause the program to stop.

Purify displays the number of access errors and leaked bytes detected

Menu bar ——

Toolbar ——

Message display ——

```
Purify: a.out

File    View    Actions    Options                              Help

▼ Finished  a.out              (   1 error,   12 leaked bytes)
  ▶ Purify instrumented a.out (pid 8701 at Wed Jul 10 19:42:26 1996)
  ▶ 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.
```
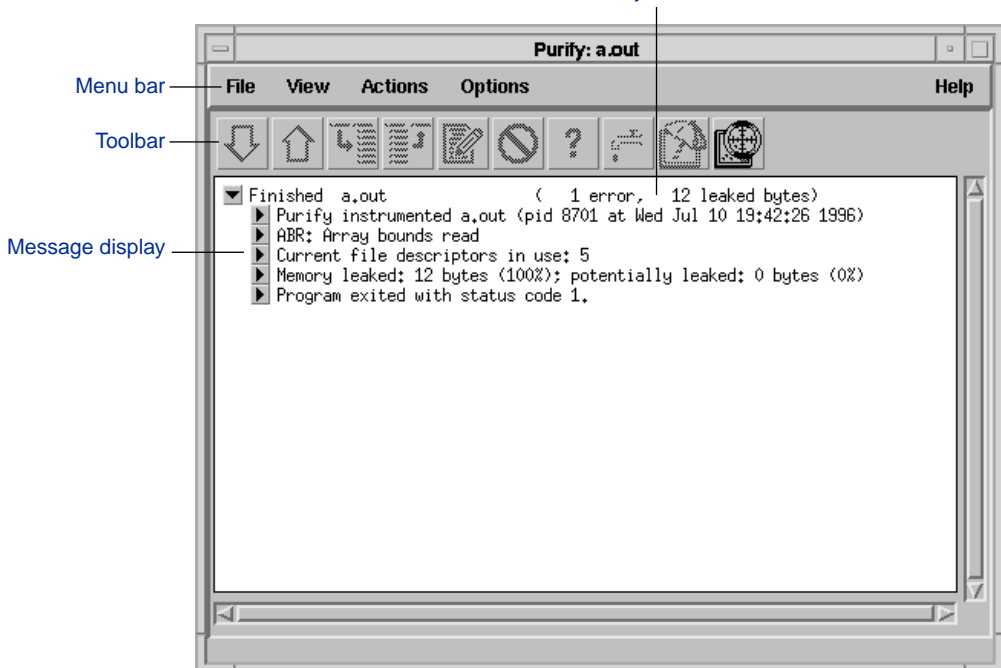
Note: The Viewer displays messages for a single executable only. It is specific to the name of the executable, the directory containing the executable, and the user id.

Using the Purify Viewer

The Viewer displays the results of the run of the Purify'd Hello World program. You can expand the outline to see additional details.
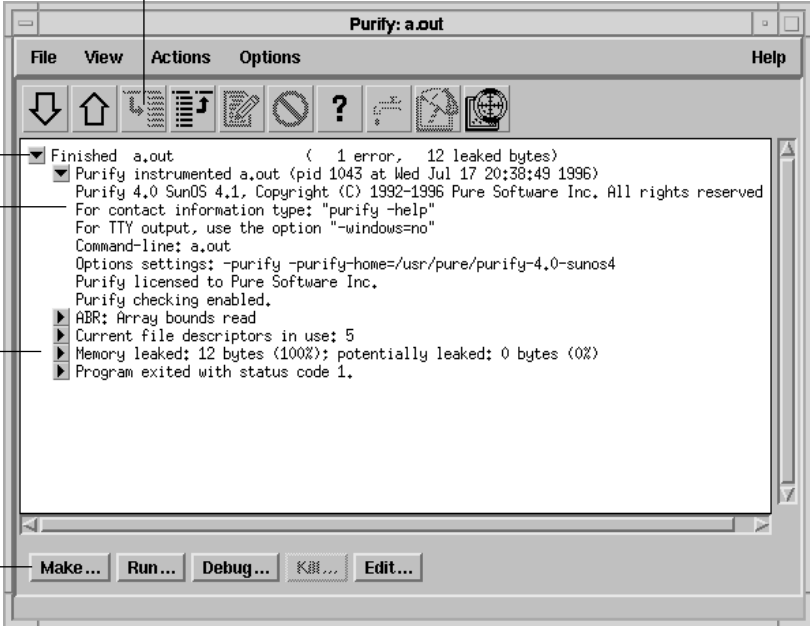
Click here to expand a message one item at a time, or all items at once

The startup banner shows the name of the program

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

Click here to expand a message or item

You can use the program controls to run a debugging cycle. To display, select **Program Controls** from the View menu
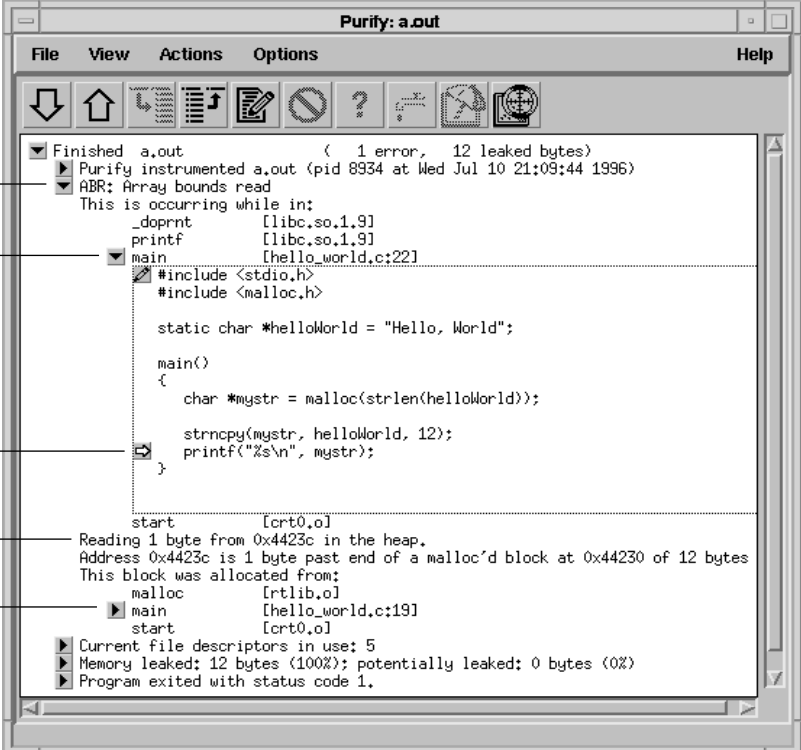
```
Purify: a.out

File    View    Actions    Options                    Help

▼ Finished  a.out              (  1 error,   12 leaked bytes)
  ▼ Purify instrumented a.out (pid 1043 at Wed Jul 17 20:38:49 1996)
    Purify 4.0 SunOS 4.1, Copyright (C) 1992-1996 Pure Software Inc. All rights reserved
    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-4.0-sunos4
    Purify licensed to Pure Software Inc.
    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.

Make...   Run...   Debug...   Kill...   Edit...
```

Note: For a list of keyboard accelerators, see the *Purify Quick Reference* at the end of this guide.

Analyzing an ABR message

Purify reports an Array Bounds Read (ABR) error in the Hello World program. This is a memory access error.

Expand the ABR message to show the details of this error.

Click to expand the ABR message —

The function call chain indicates an error occurring in _doprnt called by printf, in turn called by main (in hello_world.c)

The exact location of the error is on line 22

The details of the access error

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

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

▼ Finished  a.out                (    1 error,   12 leaked bytes)
  ▶ Purify instrumented a.out (pid 8934 at Wed Jul 10 21:09:44 1996)
▼ 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.
```

Using line numbers and source filenames

To make debugging easier, Purify indicates line number information when you build a Purify'd program using the `-g` compiler option. Purify also identifies program variables by name whenever possible.

Purify displays the ▶ button at the beginning of the function name lines when a source file and line number are available. If line number information is not available, Purify reports only function and filename information as shown with `_doprnt` and `printf`.

IRIX **Note:** On IRIX, system libraries retain their source file and line number information. Therefore, the ▶ button might appear next to a system library function whose source file is not available. When you click the ▶ button for such a line, Purify asks you for the location of the source file. Enter the location of the file if you know it, then click **OK** to expand the line.

Finding the exact location of the error

To find the exact location of the ABR error, look at the code in
`hello_world.c` again:

```
 1   /*
 2    * Copyright (c) 1992–1995 Pure Software.
     ...
 9    * This is a test program used in Purifying Hello World.
10    */
11
12   #include <stdio.h>
13   #include <malloc.h>
14
15   static char *helloWorld = "Hello, World";
16
17   main()
18   {
19       char *mystr = malloc(strlen(helloWorld));
20
21       strncpy(mystr, helloWorld, 12);
22       printf("%s\n", mystr);
23   }
```

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)
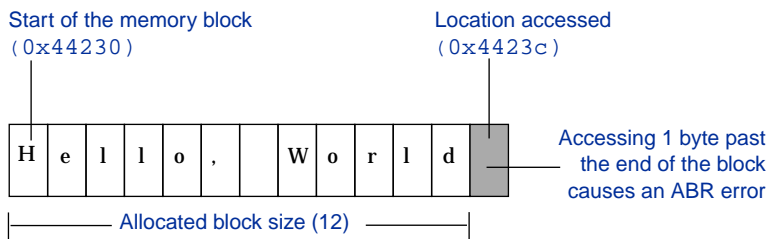
Correcting the ABR error

To correct this ABR error:

1 Click the Edit ![edit button] button to open an editor.

For information about how to change the editor used by the Viewer, see "Customizing Purify scripts" on page 6-16.

Click here to edit the source code

Or click here to edit the source code

```
┌─────────────────────────────────────────────────────────────────┐
│ □                         Purify: a.out                      □  □ │
├─────────────────────────────────────────────────────────────────┤
│   File    View    Actions    Options                      Help   │
├─────────────────────────────────────────────────────────────────┤
│  ⬇  ⬆  ⬒  ⬓  ✎  ⊘  ?  ⣿  ⬚  ⊕                                    │
├─────────────────────────────────────────────────────────────────┤
│ ▼Finished  a.out                  (  1 error,   12 leaked bytes) │
│  ▶Purify instrumented a.out (pid 9012 at Wed Jul 10 21:26:09 1996)│
│  ▼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│
├─────────────────────────────────────────────────────────────────┤
│ ◄│                                                         ▷      │
└─────────────────────────────────────────────────────────────────┘
```

Note: By default, Purify displays 7 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. See "Customizing the Purify Viewer" on page 6-15.

2 Change lines 19 and 21 as follows:

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

For more information about how to correct memory access errors, see Chapter 3, "Memory Access Errors."

Looking at the file descriptors message

Expand the file descriptors message.

Purify prints the number of file descriptors that are open and information about their origins.

Expand the file
descriptors message

Purify reserves file
descriptors 26 and
27 for its own use

```
▼ Finished  a.out                    (   1 error,   12 leaked bytes)
  ▶ Purify instrumented a.out (pid 9015 at Wed Jul 10 21:36:39 1996)
  ▶ ABR: Array bounds read
  ▼ Current file descriptors in use: 5
      FIU: file descriptor 0: <stdin>
      FIU: file descriptor 1: <stdout>
      FIU: file descriptor 2: <stderr>
      FIU: file descriptor 26: <reserved for Purify internal use>
      FIU: file descriptor 27: <reserved for Purify internal use>
  ▶ Memory leaked: 12 bytes (100%); potentially leaked: 0 bytes (0%)
  ▶ Program exited with status code 1.
```
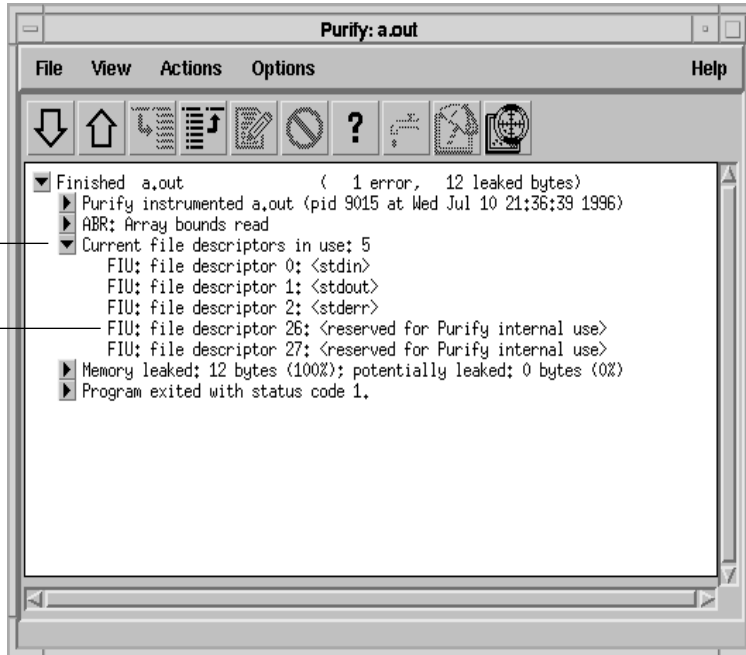
For more information about file descriptors in use, see Chapter 5, "Analyzing File Descriptors."

Understanding the memory leaked summary

When the Hello World program exits, Purify searches for memory leaks and reports all memory blocks that have been allocated but for which no pointers exist.

Note: When you run longer-running Purify'd programs, you can click the New Leaks button to generate a new leaks summary while the program is running. See "Using the new leaks button" on page 4-10.

1 Expand the memory leaked summary.

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 button to generate a new leaks summary while the program is running

The memory leaked summary reports one Memory Leak (MLK) error

The call chain shows how the leaked memory was allocated

Memory analysis by category

```
                              Purify: a.out

  File    View    Actions    Options                        Help


   ⬇    ⬆    ⏬   ☰    📝    🚫    ?    ⟐    ⬚    🌐


 ▼ Finished  a.out               (   1 error,   12 leaked bytes)
   ▶ Purify instrumented a.out (pid 9015 at Wed Jul 10 21:36:39 1996)
   ▶ 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.
```

Looking at the MLK error

It is not immediately obvious why this memory 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 function main returns rather than calls exit, thereby making mystr—the only reference to the allocated memory—go out of scope.

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

```
                         Purify: a.out

 File    View    Actions    Options                         Help


   ⬇  ⬆  ⬚  ▤  ▨  ⬚  ?  ⬚  ⬚  ⬚

 ▼ Finished  a.out                 (  1 error,   12 leaked bytes)
    ▶ Purify instrumented a.out (pid 9015 at Wed Jul 10 21:36:39 1996)
    ▶ 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.
```

If main called exit at the end, mystr would remain in scope at 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 📝 button to open an editor.

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

Looking at the heap analysis

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

- Leaked memory
- Potentially leaked memory
- Memory in use

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

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

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

```
┌─────────────────────────────────────────────────────────────────┐
│ □                        Purify: a.out                     ▫  □  │
├─────────────────────────────────────────────────────────────────┤
│   File   View   Actions   Options                          Help  │
├─────────────────────────────────────────────────────────────────┤
│  ⬇  ⬆  ⌁  ≣  ▨  ⊘  ?  ⠿  ⬡  ⬢                                   │
├─────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out                 (   1 error,   12 leaked bytes)│
│    ▶ Purify instrumented a.out (pid 9015 at Wed Jul 10 21:36:39 1996)│
│    ▶ 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.                          │
└─────────────────────────────────────────────────────────────────┘
```

Notice that this heap analysis includes information for suppressed as well as unsuppressed blocks. For information about suppressing messages, see Chapter 7, "Suppressing Purify Messages."

For information about how to find and correct memory leaks, see Chapter 4, "Memory Leaks."

Looking at the exit status summary

When the Hello World program terminates, Purify generates a summary message showing the exit status.

Expand the exit status message.

The exit status message shows information about memory usage in program code and shared libraries

```
Purify: a.out                                    □ ▢

File    View    Actions    Options                              Help

▽ Finished  a.out                    (   1 error,   12 leaked bytes)
  ▶ Purify instrumented a.out (pid 9015 at Wed Jul 10 21:36:39 1996)
  ▶ 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.
     * Basic memory usage (including Purify overhead):
          212568 code
           49656 data/bss
           12296 heap (peak use)
            2640 stack
     * Shared library memory usage (including Purify overhead):
          688128 libc_pure_p3_c0_106132226_413_U1V.so.1.9 (shared code)
           16384 libc_pure_p3_c0_106132226_413_U1V.so.1.9 (private data)
            8192 libdl.so.1.0 (shared code)
            8192 libdl.so.1.0 (private data)
            8192 libinternal_stubs.so.1.0 (shared code)
            8192 libinternal_stubs.so.1.0 (private data)
```

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.

For more information, see "Exit processing options" on page 12-10.

Rerunning a Purify'd program

To verify that you have corrected the ABR and MLK errors:

1 Recompile the program with `purify`, and run it again.

Purify directs the new run to the same Viewer:

```
┌─────────────────────────────────────────────────────────────┐
│  ▢                    Purify: a.out                    ▫ □ │
├─────────────────────────────────────────────────────────────┤
│   File    View    Actions    Options                   Help │
├─────────────────────────────────────────────────────────────┤
│  ⬇  ⬆  ⬿ ⬾  📝 ⊘  ?  ⌁  🖾 🎯                             │
├─────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out          (   1 error,   12 leaked bytes)  │
│   ▶ Purify instrumented a.out (pid 1173 at Wed Jul 17 21:15:50 1996)│
│   ▶ 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 17 21:18:28 1996)│
│   ▶ Current file descriptors in use: 5                      │
│   ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)│
│   ▶ Program exited with status code 0.                      │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

Purify reports no errors and no memory leaks

The Viewer displays messages for a single executable only. If you rename the Hello World program and run it again, Purify displays the run in a new Viewer.

2 Compare the new run to the previous run and see that Purify no longer reports the ABR or MLK errors.

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

Note: You can save the output from a run of a Purify'd program for later viewing. See "Saving Purify output to a view file" on page 6-3. For more extensive tutorials on correcting memory access errors and memory leaks, see Chapters 3 and 4.

3

Memory Access Errors

This chapter begins with a description of how Purify finds memory access errors; then uses the `testHash` program provided with Purify to show you how to find and correct four types of memory access errors:

- Reading uninitialized memory
- Reading and writing beyond the bounds of an array
- Reading and writing freed memory
- Freeing unallocated or non-heap memory

To begin the `testHash` tutorial, go to "Building the testHash example program" on page 3-6.

For a complete list of memory access options and API functions, see "Memory access options" on page 12-14 and "Memory access API" on page 12-15. For a list of memory access error messages, see Chapter 10.

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 still initialized.

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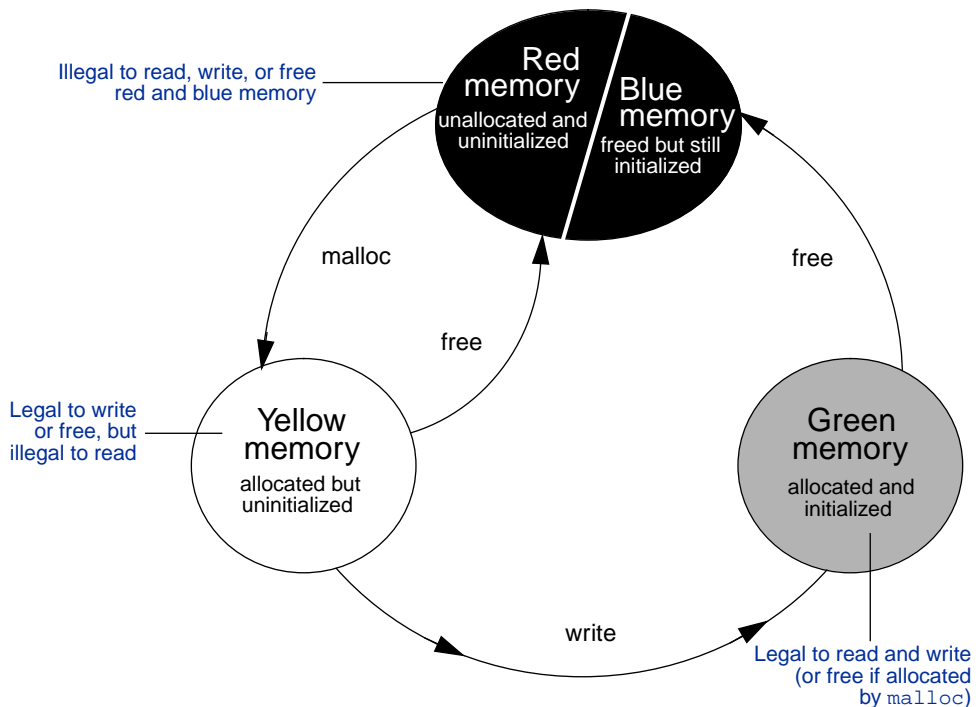
Purify labels memory states by color.

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 upon 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 ABW error at the assignment to `array[11]` because it is 4 bytes beyond the end of the array.

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

Purify inserts guard 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 guard 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. See "Static checking options" on page 12-24.

Notes and limitations

- Purify does not detect array bounds errors between individual local (stack) variables. On SunOS and Solaris, Purify inserts guard zones between stack frames, causing Stack Array Bounds Read (SBR) and Stack Array Bounds Write (SBW) errors on accesses that extend beyond all the local variables in a function. Purify detects accesses beyond the end of the stack (BSR and BSW errors) on all platforms, as well as Uninitialized Memory Reads (UMR) on all stack variables.

- Due to the flexibility of manipulating pointers in C and C++ programs, a pointer can *accidently* access a legally-allocated block of memory that is in fact beyond the block that you are attempting to access. In this case, Purify does *not* signal illegal memory access errors because the memory is properly allocated and initialized. Purify monitors memory accesses and the blocks of memory accessed, not pointer arithmetic. You can use the `-chain-length` option to adjust the size of red zones to find these types of errors. See page 12-18.

- Purify detects array bounds errors in arrays within C structures *only* when the access extends beyond the entire structure.

Building the testHash example program

Before you start, you need to build the testHash program. The testHash program is located in the `<purifyhome>/example` directory. The original uncorrected source code is in the `hash.c` file. The corrected source code is in `hash.c_afterpure`.

To correct the errors in `hash.c` as you work through this chapter, you must build all the programs included in the `example` directory.

Copy the `example` directory to your home directory, then run `make`.

```
% cp -r <purifyhome>/example ~/example
% cd ~/example
% make
cc -c -g testHash.c
cc -c -g hash.c
cc -o testHash testHash.o hash.o
purify cc -o testHash.pure testHash.o hash.o
Purify 4.0 SunOS 4.1, Copyright 1992-1996 Pure Software Inc.
Instrumenting: testHash.o hash.o Linking
%
```

Note: The examples in this chapter show the testHash program built on a SunOS 4.1 system. You might see different output and debugging information on Solaris 2, HP-UX, or IRIX. The memory address information in the examples is compiler and system dependent. Also, the source code line numbers you see as you work through this tutorial might be different from the numbers shown in this chapter.

Running the testHash program without Purify

The `testHash` program implements a hash table and includes "rigorous" testing routines.

Run the `testHash` program:

```
% testHash
Testing makeHashTable.
Testing putHash - adding from 0 to 100.
Testing getHash - getting from 0 to 100.
Testing remHash - removing from 0 to 50.
Testing remHash - removing from 0 to 50.
Testing getHash - getting from 0 to 50.
Testing getHash - getting from 50 to 100.
Testing putHash - adding from 0 to 50.
Testing putHash - adding from 50 to 100.
Testing delHashTable.
%
```

The `testHash` program shows `hash.c` as passing this test suite. However, Purify will show that it contains a number of major errors. If these routines are included in a larger program, the errors can appear as crashes in seemingly unrelated code.

Running the Purify'd testHash program

Run the instrumented version of testHash:

```
% testHash.pure
```

You can see that Purify detects many memory access errors.

Memory access errors

Purify found these memory access errors

```
  Finished  testHash.pure        ( 583 errors,  182 leaked bytes)
    Purify instrumented testHash.pure (pid 7014 at Mon Jul  8 20:24:56 1996)
    UMR: Uninitialized memory read (13 times)
    UMR: Uninitialized memory read (13 times)
    ABW: Array bounds write (100 times)
    ABR: Array bounds read (8 times)
    ABR: Array bounds read (108 times)
    ABR: Array bounds read (50 times)
    FMR: Free memory read (50 times)
    ABR: Array bounds read (50 times)
    ABW: Array bounds write (50 times)
    ABR: Array bounds read (4 times)
    ABR: Array bounds read (50 times)
    FNH: Freeing non heap memory (87 times)
    Current file descriptors in use: 5
    Memory leaked: 182 bytes (37.8%); potentially leaked: 0 bytes (0%)
    Program exited with status code 0.
```

Purify: testHash.pure

File   View   Actions   Options   Help

Debugging the testHash program

The easiest way to track down multiple errors in a program is to run the instrumented program under a debugger and set a breakpoint on the purify_stop_here function. Each time Purify detects a new error, it generates a message and hits the breakpoint in the debugger. This helps identify the error at its origin.

Alternatively, Purify can automatically attach a debugger to your program when it detects an error. See "Enabling JIT debugging" on page 6-11 for details.

Note: Unless otherwise noted, the debugging examples in this section use the dbx debugger. The banner information that appears in several of the messages is left out of these examples.

Debugging with dbx

IRIX You might notice minor differences between these examples and the implementation of dbx on IRIX.

```
% dbx testHash.pure
Reading symbolic information...
Read 2588 symbols
(dbx) stop in purify_stop_here
(2) stop in purify_stop_here
(dbx) run
Running: testHash.pure

Testing makeHashTable.
Testing putHash - adding from 0 to 100.

stopped in purify_stop_here at 0x2dd4
purify_stop_here:nop
Current function is putHash
 135  entry && strcmp (entry->key, key);
(dbx)
```

(HPUX) Debugging with xdb

On HP-UX, use the wrapper script for xdb, located in
<producthome>/purify_xdb. This configures the debugger for
Purify'd programs. (Purify provides similar scripts for dde and
softdebug.) You should also add the line:

```
z 18 sir
```

to your ~/.xdbrc file. See the README file for more details.

```
% purify_xdb testHash.pure
      200: *    value should be there or not.
      201: */
      202: int main()
      203: {
      204:   hashtable* ht;
      205:   char* testTable[TABLE_SIZE];
      206:
>     207:   fillTestTable(testTable);
      208:
      209:   ht = testMakeHashTable();
      210:   testPutHash(ht, testTable,  0, 100, FALSE);
      211:   testGetHash(ht, testTable,  0, 100, TRUE);
      212:   testRemHash(ht, testTable,  0,  50, TRUE);
      213:   testRemHash(ht, testTable,  0,  50, FALSE);
      214:   testGetHash(ht, testTable,  0,  50, FALSE);
 File: testHash.c   Procedure: main    Line: 207
Copyright Hewlett-Packard Co. 1985,1987-1992. All Rights
Reserved.
<<<< XDB Version A.09.01 HP-UX >>>>
No core file
Procedures:     15
Files: 3
>Sig Stop    Ignore  Report   Name
 18  No      Yes     No       death of child
>b purify_stop_here
Overall breakpoints state:  ACTIVE
Added:
 1: count:   1 Active     purify_stop_here: 1:
>r
Starting process 24385:  "testHash.pure"
```

```
Wait...loading shared-library map tables. Done.
Testing makeHashTable.
Testing putHash - adding from 0 to 100.

breakpoint at 0x00029a88
purify_stop_here.c: purify_stop_here: 1:
>d
```

Reading uninitialized memory

When a program attempts to perform an operation using values from uninitialized memory, the results can be unpredictable. The code often appears to work correctly until an unrelated part of the program changes, causing it to malfunction in mysterious ways. Purify calls this type of error an Uninitialized Memory Read (UMR).
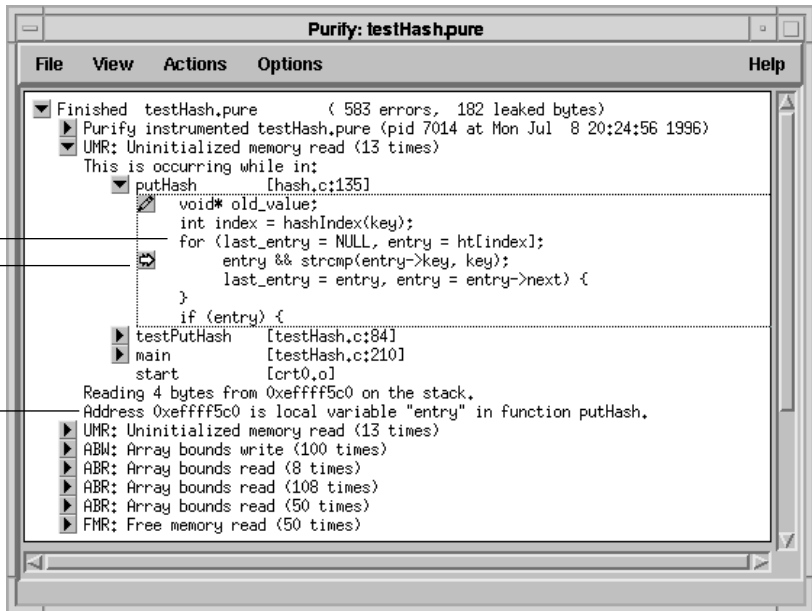
A UMR example

Expand the UMR line, then expand the putHash line.

entry is initialized to the value of `ht[index]`, so it appears that *entry* is initialized

4 bytes of uninitialized memory are used on line 135 of `putHash`

The uninitialized data is read from the local variable *entry* on the stack

Purify distinguishes between *copying* uninitialized data and *using* it in an operation. In this example, the program tests whether `entry` is non-NULL on line 135. Purify checks this access, and finds that `entry` contains uninitialized data. Purify reports a UMR error. The error is that `ht[index]` is not initialized *before* the copy to `entry`.

Notice, however, that Purify does *not* signal an error on line 134 when `ht[index]` is copied into `entry`. This is because it is common for correct code to copy uninitialized data, particularly when copying structures containing padding bytes used to align fields of differing sizes. For this reason, Purify does not report Uninitialized Memory Copy (UMC) errors by default.

In this example, the code appears to work correctly when it is tested because the value of `ht[index]` is expected to be initialized to NULL. Since the memory in `ht[index]` has not been used, it happens to be NULL even without being initialized. The code is not correct, but appears to run correctly in the test cases. However, if new code is added later the program can produce incorrect results.

Here is another example:

```
int i;
int j;

j=i;
printf("%d",j);
```

In this example, `i` and `j` are not initialized. The value of `i` is copied to `j` so Purify marks `j` as uninitialized also. When the value of `j` is used as an argument to `printf`, Purify reports a UMR error. Purify actually detected a UMC error at `j=i`, however, by default Purify suppresses UMC error messages.

Note: You can temporarily unsuppress the UMC messages in the `testHash` example by selecting **Suppressed Messages** in the View menu. For more information about how to suppress and unsuppress messages, see Chapter 7, "Suppressing Purify Messages."

Finding the cause of the UMR error

To correct this error, you must determine where ht[index] should have been initialized. By looking at Purify's initial error message, you can see that putHash is called by testPutHash, which is called by main.

1 Click the Edit 🖉 button in the message to display the source file.

2 Notice that ht is initially allocated in the function makeHashTable in hash.c.

```
hashtable* makeHashTable()
{
 hashtable* ht;
 ht = (hashtable*)malloc(HASHTABLE_SIZE*sizeof(hashEntry*));
 return(ht);
}
```

The memory that ht points to is never initialized.

Correcting the UMR error

To correct this UMR error:

1 Add the initialization code:

```
hashtable* makeHashTable()
{
 hashtable* ht;
 ht = (hashtable*)malloc(HASHTABLE_SIZE*sizeof(hashEntry*));
 /* fix umr by initializing all hash pointers to null */
 memset(ht, 0, HASHTABLE_SIZE*sizeof(hashEntry*));
 return(ht);
}
```

Add this code ——

2 Recompile the program and run it again.

Purify should no longer signal an uninitialized memory read error on line 135 of hash.c. You have successfully corrected a UMR error.
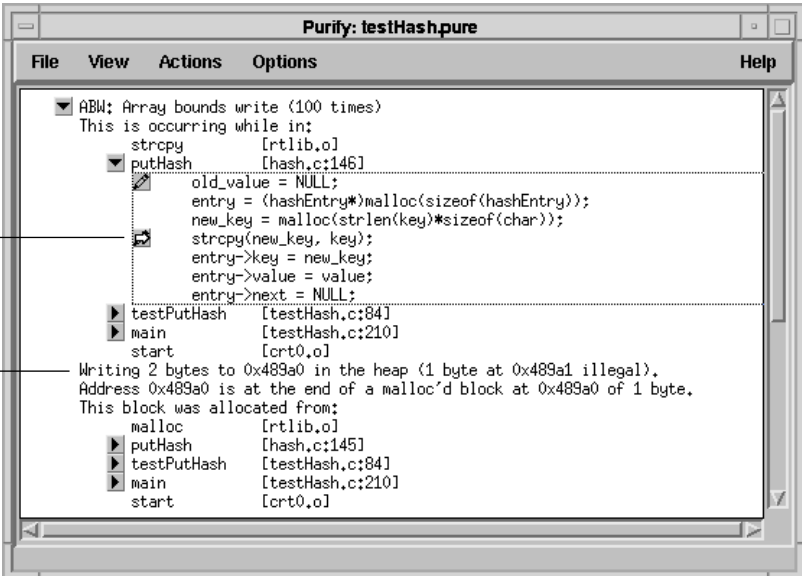
Reading and writing beyond the bounds of an array

Reading before the beginning or after the end of an array uses data that is not intended to be used. If another part of the program writes to this memory, unexpected and incorrect values can be read and used. Similarly, writing beyond the bounds of an array can corrupt data used by other parts of a program.

Purify calls these types of errors Array Bounds Read (ABR) and Array Bounds Write (ABW) errors. Purify reports ABR and ABW errors when they occur, clearly indicating the origin of data corruption.

An ABW example

1 Expand the first ABW line, then expand the `putHash` line.

On line 146, `strcpy` is writing 2 bytes into a 1-byte array. The destination of the copy is `new_key`

The array is in the heap at memory location `0x489a0`

2 Use your debugger to verify that `new_key` is the overwritten array:

```
(dbx) print new_key
`new_key = 0x489a0 ""
```

Finding the cause of the ABW error

To find the cause of the error, you must determine why the program is writing beyond the end of `new_key`.

1 Click the Edit ✎ button in the message to display the source file.

2 Look at line 146 in `putHash` to see why it is trying to copy into a string that is not long enough. The string `new_key` is allocated on line 145, just prior to the `strcpy`.

The code attempts to create an array large enough to hold the string in `key` by getting its length from `strlen`. The problem is that `strlen` returns only the number of characters in the string `key` and does not include the NULL character terminating the string. When the NULL character is copied into `new_key` by `strcpy`, the program writes beyond the end of the array.

This error can cause intermittent failures. The `malloc` function call returns memory blocks with sizes rounded up to a multiple of 8 bytes. Most often, the NULL byte is written into padding or alignment space with no adverse effect. Occasionally, however, the write corrupts the adjacent memory. If that memory is used, the error can result in serious consequences and noticeable symptoms. Purify detects the error in *every* case.

Correcting the ABW error

To correct this ABW error:

1 Add 1 to the value returned by `strlen` on line 145.

```
...
}
else {
        old_value = NULL;
        entry = (hashEntry*)malloc(sizeof(hashEntry));
```

Add +1 to this line ────────

```
        new_key = malloc((strlen(key)+1)*sizeof(char)); /* fix abw */
        strcpy(new_key, key);
        entry->key = new_key;
        entry->value = value;
        entry->next = NULL;
        if (last_entry) {
            last_entry->next = entry;
        }
        else {
            ht[index] = entry;
        }
}
...
```

2 Recompile the program and run it again.

Purify no longer signals the ABW error. You have successfully corrected an ABW error.

An ABR example

If your program makes an improper cast it can cause an array bounds error. Consider this code fragment:

```
void badCast(key)
  void* key;
{
      hashEntry* entry = (hashEntry*)key;
      if (entry->value) {
...
```

If `key` is a pointer to a single `malloc`'d byte, the offset to `value` will go beyond the end of `key`. This causes an ABR error when the code refers to `entry->value`. The code is accessing memory illegally even if it does not appear to be running off the end of an array.

Reading or writing freed memory

When a program frees a segment of memory, but continues to read from and write to that segment, the data in that segment is no longer protected. Another part of the program might allocate and start using this freed segment, change the data, and cause the program to crash mysteriously. Purify calls these types of errors Free Memory Read (FMR) or Free Memory Write (FMW).

It is not unusual to separate the use and freeing of memory. For this reason, Purify tells you not only where you read from freed memory, but also where you freed the block and where it was originally allocated. These operations are usually widely separated when two different modules pass data back and forth, and one module frees the other module's memory.

Failures due to FMR errors can be more intermittent than ABW errors. For example, if you add timer signals to your program— perhaps to update a program busy cursor—one out of 100,000 timer interrupts might occur between the `free` and the use of `entry`. If the handler code uses heap memory, it can reuse `entry`, corrupting it and causing intermittent behavior or a crash.

To facilitate finding FMR and FMW errors, Purify does not return freed memory for re-use as soon as it is freed. Instead, Purify puts the memory on a first-in, first-out free queue, returning it to the system only when the free-queue is full or when the system is out of memory. You can change the length of the free queue by using the `-free-queue-length` option. See "Memory access options" on page 12-14.

An FMR example

1 Expand the FMR line, then expand the `remHash` line.

The FMR message indicates that the program is reading from freed memory in `remHash`.

On line 187, the program is reading freed memory

The freed memory starts at `0x48940` and is 12 bytes long

```
                                       Purify: testHash.pure
 File    View    Actions    Options                                      Help

  ▶ ABR: Array bounds read (50 times)
  ▼ FMR: Free memory read (50 times)
    This is occurring while in:
    ▼ remHash        [hash.c:187]
      ✎          last_entry->next = entry->next;
                 }
                 else {
      ⇨           ht[index] = entry->next;
                 }
               }
               else {
    ▶ testRemHash    [testHash.c:162]
    ▶ main           [testHash.c:212]
      start          [crt0.o]
    Reading 4 bytes from 0x48948 in the heap.
    Address 0x48948 is 8 bytes into a freed  block at 0x48940 of 12 bytes.
    This block was allocated from:
      malloc         [rtlib.o]
    ▶ putHash        [hash.c:144]
    ▶ testPutHash    [testHash.c:84]
    ▶ main           [testHash.c:210]
      start          [crt0.o]
```

The program is reading 4 bytes starting at `0x48948`, **8 bytes into the freed block.**

2 Use your debugger to verify that this is the freed memory:

```
(dbx) print &entry->next
& entry->next = 0x48948
(dbx) print entry
entry = 0x48940
```

The block of freed memory read is `entry->next`, **and** `entry` **is the block of memory that was freed;** `entry` is a pointer to a `hashEntry`.

Note: You can also identify the `free`'d block by looking at the allocation call chain. The message indicates that the block was allocated in `putHash`, line 144. The freed data is a `hashEntry`.

Finding the cause of the FMR error

To find the cause of the error, you must find out why memory is being read after it has been freed.

1 Click the Edit 📝 button in the message to display the source file.

2 Look at the function `remHash`. Notice that the order of freeing the block and updating the pointers of the linked list is confused.

```
void* remHash(ht, key)
     hashtable* ht;
     char* key;
{
 hashEntry* last_entry;
 hashEntry* entry;
 void* value;
 int index = hashIndex(key);
 for (last_entry = NULL, entry = ht[index];
      entry && strcmp(entry->key, key);
      last_entry = entry, entry = entry->next) {
 }
 if (entry) {
     value = entry->value;
     free(entry->key);
     free(entry);
     if (last_entry) {
      last_entry->next = entry->next;
     }
     else {
       ht[index] = entry->next;
     }
 }
 else {
     value = NULL;
 }
 return(value);
}
```

Look here ——— `free(entry);`

And here ——— `ht[index] = entry->next;`

Correcting the FMR error

To correct this FMR error:

1 Move both of the `frees` so they occur after the pointer updates.

```
if (entry) {
    value = entry->value;
    if (last_entry) {
        last_entry->next = entry->next;
    }
    else {
        ht[index] = entry->next;
    }
    free(entry->key);    /* moved free to fix fmr */
    free(entry);         /* moved free to fix fmr */
}
.
.
.
```

move `frees`
to here ⎯⎯

2 Recompile the program and run it again.

Purify no longer signals an FMR error. You have successfully corrected this FMR error.

Freeing unallocated or non-heap memory

Confusion about memory ownership can lead to freeing the same
memory several times, or freeing a block of memory that was
never allocated. Purify calls these errors Freeing Non-Heap (FNH)
or Freeing Unallocated Memory (FUM).

An FNH example

1 Expand the FNH line, then expand the `delHashTable` line.

```
┌──────────────────────────────────────────────────────────────────────┐
│ ─                        Purify: testHash.pure                    □ □  │
├──────────────────────────────────────────────────────────────────────┤
│   File    View    Actions    Options                          Help     │
├──────────────────────────────────────────────────────────────────────┤
│   ▶ ABR: Array bounds read (50 times)                               ▲  │
│   ▼ FNH: Freeing non heap memory (87 times)                            │
│     This is occurring while in:                                        │
│          free           [rtlib.o]                                      │
│     ▼ delHashTable   [hash.c:60]                                       │
│       ✎              last_entry = entry, entry = entry->next) {        │
│                    if (last_entry) {                                    │
│                      free(last_entry->key);                            │
│       ▷              free(last_entry->value);                          │
│                      free(last_entry);                                 │
│                    }                                                    │
│                  }                                                      │
│     ▶ testDelHashTable [testHash.c:193]                                │
│     ▶ main            [testHash.c:218]                                  │
│          start          [crt0.o]                                       │
│     Attempting to free block at 0xefff764 on the stack.                │
│     Address 0xefff764 is 208 bytes past start of local variable "testTable" in fu│
│     Note: Some 'free's permit freeing such memory, but this is not portable.│
│     To suppress this error, add 'suppress fnh *' to your .purify file.  │
│   ▶ Current file descriptors in use: 5                                 │
│   ▶ Memory leaked: 182 bytes (37.8%); potentially leaked: 0 bytes (0%) │
│   ▶ Program exited with status code 0.                              ▽  │
├──────────────────────────────────────────────────────────────────────┤
│  ◁                                                              ▷       │
└──────────────────────────────────────────────────────────────────────┘
```
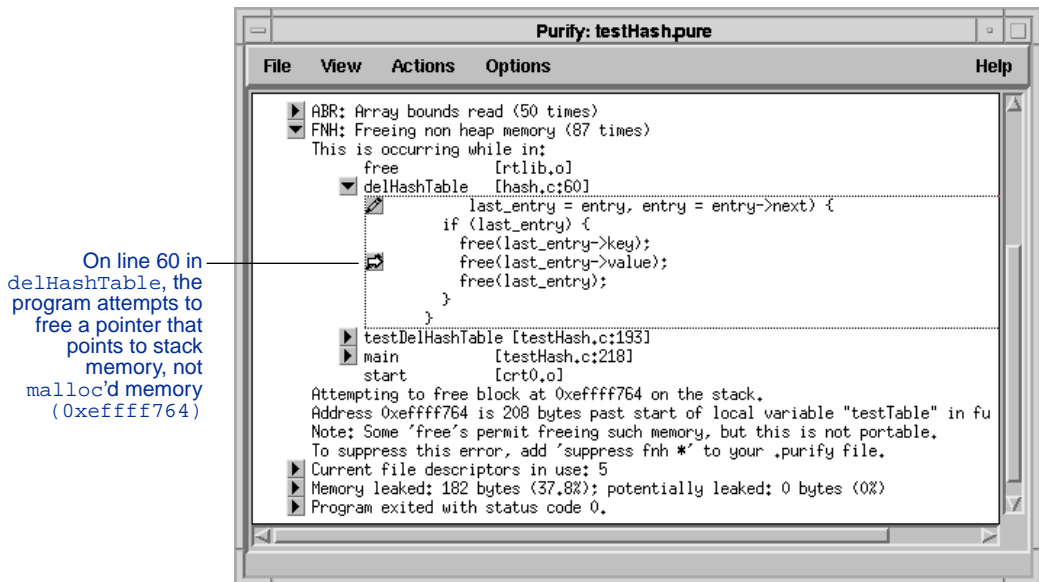
On line 60 in `delHashTable`, the program attempts to free a pointer that points to stack memory, not `malloc`'d memory (`0xefff764`)

2 Use your debugger to confirm that `last_entry->value` is the same
address that Purify reports as freed.

```
(dbx) print last_entry->value
last_entry->value = 0xefff764
```

Finding the cause of the FNH error

To find the cause of the error, you need to determine why the program is trying to free the block on the stack.

1 Click the Edit 🖉 button in the message to display the source file.

Notice that the values inserted into the hash table are pointers to the stack.

2 Look at line 60 in `delHashTable`:

```
void delHashTable(ht)
    hashtable* ht;
{
    int index;
    hashEntry* last_entry;
    hashEntry* entry;
    for (index = 0; index < HASHTABLE_SIZE; index++) {
        for (last_entry = NULL, entry = ht[index];
            entry;
            last_entry = entry, entry = entry->next) {
            if (last_entry) {
                free(last_entry->key);
                free(last_entry->value);
                free(last_entry);
```

This block of
memory does not
belong to the
hash table

As the program goes through the hash table and frees each block, it attempts to free the value that was put into the hash table. This block of memory does not belong to the hash table; it belongs to the routine that uses the hash table to store this value.

It is not uncommon for ownership of memory to become confused between modules, resulting in one module attempting to free the memory of another module.

Correcting the FNH error

To correct this FNH error:

1 Remove the incorrect call to `free`.

```
void delHashTable(ht)
  hashtable* ht;
{
    int index;
    hashEntry* last_entry;
    hashEntry* entry;
    for (index = 0; index < HASHTABLE_SIZE; index++) {
        for (last_entry = NULL, entry = ht[index];
             entry;
             last_entry = entry, entry = entry->next) {
             if (last_entry) {
                 free(last_entry->key);
                 /* free(last_entry->value); removed to fix fnh */
                 free(last_entry);
```

Remove this `free` ——————

2 Recompile the program and run it again.

Purify should not signal an FNH error at line 60 in `delHashTable`. You have successfully corrected this FNH error.

4

Memory Leaks

This chapter begins with a description of how Purify reports memory leaks; then continues using the testHash program provided with Purify to show you how to find and correct a memory leak. The testHash tutorial begins on page 4-4.

For a complete list of memory leak options and API functions, see "Memory leak options" on page 12-16 and "Memory leak API" on page 12-17.

How Purify reports memory leaks

Purify generates a Memory Leaked summary (MLK) when the program exits, that is, when the program goes through an exit(). A Purify memory leaked summary indicates the amount of memory leaked by the program during execution and identifies the origin of each leak by the functions that allocated the memory. If the program terminates because of a signal, Purify does *not* generate a report.

Purify finds memory leaks by searching for blocks that have no pointers to them. Since these blocks cannot be accessed, they are lost to the program and cannot be freed. Consider this example of a linked list:
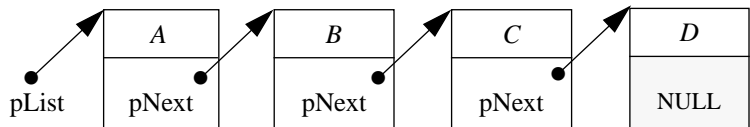
At first, memory blocks A, B, C, and D all have references to them. A is referenced through the global variable pList, and B, C, and D through the pNext pointers.

Then, block C is removed from the list by setting the `pNext` pointer in B to point to D instead of C.
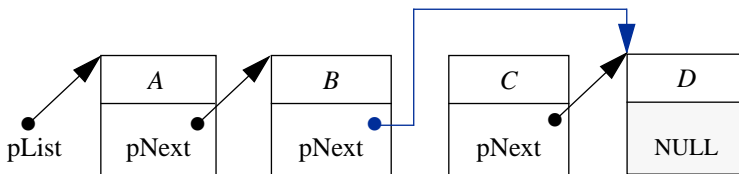
However, list block C is *not* freed. Unless there is another pointer to C, there is now no possible way to free C. It is a leaked memory block.

In the memory leaked message for this linked list, Purify shows 1 leaked block (C), 3 memory-in-use blocks (A, B, and D), and 4 allocated blocks (A, B, C, and D).

Traditional leak detectors (like mprof) report the memory blocks that are not freed before the program exits by simply matching memory allocations and corresponding frees. This is misleading, since most of these memory blocks are *not* leaks. They are either permanently allocated memory, such as symbol tables, or memory that happens to be in use when the exit function is called. In this linked list example, a typical malloc-debug leak detector would report all four blocks (A, B, C, and D) as leaked. When no distinction is made between true memory leaks and memory in use, it is difficult to identify the real problem.

Purify also reports memory blocks that do not have pointers to their beginnings, but that *do* have a pointer to their interior. These blocks are probably leaks, because there is no pointer that can be used directly to free them. Sometimes, however, these blocks are still in use. Purify calls these Potential Leaks (PLK) to distinguish them from true Memory Leaks (MLK).

Notes and limitations

Purify finds leaks of memory allocated using malloc and related functions. It cannot find memory leaks in programs that do not use malloc. Since the C++ new operation calls malloc, Purify finds memory leaks in C++ code.

By default, Purify cannot find all leaks in memory blocks handled by custom memory management routines that you create on top of malloc, new, and delete. For example, if you allocate a large block of memory and break it up into smaller blocks, you can manage the allocation and freeing of that memory on your own. Purify does not find leaks of those subdivided blocks.

See Chapter 9, "Custom Memory Managers," for details on how to control Purify's operation while using custom memory managers.

Finding the memory leaks in testHash

See "Building the testHash example program" on page 3-6 to build the testHash program. If you corrected the memory access errors in Chapter 3, you might want to start this tutorial with a fresh copy of the testHash program so that the line numbers you see will match the ones shown in this chapter.

Note: The examples in this chapter show the testHash program built on a SunOS 4.1 system. You might see different output and debugging information on Solaris 2, HP-UX, or IRIX. The memory address information in all examples is compiler and system dependent.

1 Run the Purify'd version of testHash.

```
% testHash.pure
```

When testHash exits, Purify reports 182 bytes of leaked memory

The memory leaked summary indicates that a total of 182 bytes of memory are leaked

```
Purify: testHash.pure

File    View    Actions    Options                              Help

▼ Finished  testHash.pure          ( 583 errors,  182 leaked bytes)
  ▶ Purify instrumented testHash.pure (pid 7014 at Mon Jul  8 20:24:56 1996)
  ▶ UMR: Uninitialized memory read (13 times)
  ▶ UMR: Uninitialized memory read (13 times)
  ▶ ABW: Array bounds write (100 times)
  ▶ ABR: Array bounds read (8 times)
  ▶ ABR: Array bounds read (108 times)
  ▶ ABR: Array bounds read (50 times)
  ▶ FMR: Free memory read (50 times)
  ▶ ABR: Array bounds read (50 times)
  ▶ ABW: Array bounds write (50 times)
  ▶ ABR: Array bounds read (4 times)
  ▶ ABR: Array bounds read (50 times)
  ▶ FNH: Freeing non heap memory (87 times)
  ▶ Current file descriptors in use: 5
  ▶ Memory leaked: 182 bytes (37.8%); potentially leaked: 0 bytes (0%)
  ▶ Program exited with status code 0.
```

2 Expand the memory leaked summary, then expand the MLK error messages.

Purify reports two MLK errors: a 12-byte leak and a 2-byte leak, each occurring multiple times. This results in a total of 182 bytes of leaked memory.

The last leaked block begins at address `0x4f400`

A 2-byte leak occurs 13 times resulting in 26 leaked bytes

The leaked memory is allocated in `putHash` line 145

The last leaked memory begins at address `0x4f460`

```
Purify: testHash.pure
File    View    Actions    Options                          Help

Memory leaked: 182 bytes (37.8%); potentially leaked: 0 bytes (0%)
  MLK: 156 bytes leaked in 13 blocks
  This memory was allocated from:
      malloc          [rtlib.o]
      putHash         [hash.c:144]
      testPutHash     [testHash.c:84]
      main            [testHash.c:216]
      start           [crt0.o]
  Block of 12 bytes (13 times); last block at 0x4f400
  MLK: 26 bytes leaked in 13 blocks
  This memory was allocated from:
      malloc          [rtlib.o]
      putHash         [hash.c:145]
              old_value = entry->value;
              entry->value = value;
          }
          else {
              old_value = NULL;
              entry = (hashEntry*)malloc(sizeof(hashEntry));
              new_key = malloc(strlen(key)*sizeof(char));
              strcpy(new_key, key);
              entry->key = new_key;
              entry->value = value;
              entry->next = NULL;
              if (last_entry) {
                  last_entry->next = entry;
      testPutHash     [testHash.c:84]
      main            [testHash.c:216]
      start           [crt0.o]
  Block of 2 bytes (13 times); last block at 0x4f460
  Purify Heap Analysis (combining suppressed and unsuppressed blocks)
                         Blocks     Bytes
          Leaked          26         182
  Potentially Leaked       0           0
          In-Use         100         300
          -----------------------------------------
          Total Allocated 126         482
```

3 Expand the `putHash` function. Notice that each leaked block is a `hashEntry` or its associated `key`.

Finding the source of memory leaks

To track down a memory leak, you need to know how the memory blocks are used and where they are stored, and you need to understand where they are lost.

Run the program again and look for a section of code that loses the last pointer to a block of memory. The last pointer to a block of memory can be lost if:

- The pointer is reassigned to a new value
- The pointer goes out of scope
- A memory block containing the pointer is freed or becomes a leak itself

To understand this memory leak message, you need to review how to store and use hashEntry type memory blocks. It is possible that a pointer to a hashEntry is being lost when a new one is inserted in putHash, or when the old one in remHash is removed. It is also possible that they are being lost when they are removed in delHashTable.

Notice that only 13 blocks are lost, even though many more are added or removed as the test program runs. Therefore, do not expect to see a leak on every hashEntry that is added, removed, or deleted.

Using your debugger to set breakpoints

1 Start your debugger and set breakpoints in the `testHash` program after calls to `PutHash`, `RemHash`, and `DelHashTable`.

IRIX Sun OS4 Solaris

```
% dbx testHash.pure
(dbx) file testHash.c
(dbx) stop at 99
(1) stop at "testHash.c":99
(dbx) stop at 183
(2) stop at "testHash.c":183
(dbx) stop at 194
(3) stop at "testHash.c":194
(dbx) run
```

HPUX **Note:** See page 3-10 for an explanation of how to use `purify_xdb`.

```
% purify_xdb testHash.pure
>v testHash.c
>b 99
Overall breakpoints state:  ACTIVE
Added:
 1: count:  1  Active    testPutHash: 99: }
>b 183
Overall breakpoints state:  ACTIVE
Added:
 2: count:  1  Active    testRemHash: 183: }
>b 194
Overall breakpoints state:  ACTIVE
Added:
 3: count:  1  Active    testDelHashTable: 194: }
>r
Starting process 796:  "testHash.pure"
.
.
.
```

Note: For more information, see "Calling Purify API functions from a debugger" on page 11-7. For information about using Purify's just-in-time debugging feature, see "Enabling JIT debugging" on page 6-11.

2 Rerun the `testhash` program.

Running purify_new_leaks

3 Each time the program stops (at the end of testPutHash,
testRemHash, and testDelHashTable), call purify_new_leaks.

By calling purify_new_leaks, you can get a message showing the
new leaks that occurred since the last call to purify_new_leaks. If
no leaks are reported, continue running the program to the next
breakpoint.

```
stopped in testPutHash at line 99 in file "testHash.c"
   99 }
```
Use this command in ——(dbx) **print purify_new_leaks()**
your debugger to call   .
purify_new_leaks   .
                         .
Purify finds no leaks ——purify_new_leaks() = 0
at the breakpoint   (dbx) **cont**
on line 99   .
                         .
                         .
```
stopped in testPutHash at line 194 in file "testHash.c"
   194 }
(dbx) print purify_new_leaks()
```
On line 194, at the   .
end of the function   .
testDelHashTable,   .
Purify finds 182 ——purify_new_leaks() = 182
leaked bytes

Note: In your own longer-running programs, you can use the New
Leaks button to generate a new leaks summary while the program
is running. See "Using the new leaks button" on page 4-10.

4 Use your debugger to take a closer look at the last `hashEntry` that was leaked.

```
(dbx) print *((hashEntry *)0x4f400)
*(hashEntry *) 0x4f400 = {
        key   = 0x83328 "49"
        value = 0xf7fff524
        next = (nil)
}
(dbx)
```

The debugger confirms that this is the last `hashEntry` in the list because its `next` field is NULL. Now look at the source code for `delHashTable`, looking for errors relating to the edge case of handling the last member of the list.

```
void delHashTable(ht)
     hashtable* ht;
{
     int index;
     hashEntry* last_entry;
     hashEntry* entry;
     for (index = 0; index < HASHTABLE_SIZE; index++) {
Inner loop off by 1 ───────── for (last_entry = NULL, entry = ht[index];
                   entry;
                   last_entry = entry, entry = entry->next) {
               if (last_entry) {
                   free(last_entry->key);
                   free(last_entry);
               }
          }


     }
     free(ht);
}
```

Notice that the inner loop is off by one. The loop deallocates the `entry` prior to the present position, thereby failing to deallocate the last `entry` in the list. This means that the pointer to the last `hashEntry` in each list is being dropped. This happens because the loop is terminated prematurely while `last_entry` still points to an entry, and the memory is never freed.

Correcting the error

1 To correct this error, add a `free` for the last `hashEntry` and its `key` at the end of the loop.

```
void delHashTable(ht)
    hashtable* ht;
{
 int index;
 hashEntry* last_entry;
 hashEntry* entry;
 for (index = 0; index < HASHTABLE_SIZE; index++) {
    for (last_entry = NULL, entry = ht[index];
         entry;

         last_entry = entry, entry = entry->next) {
      if (last_entry) {
          free(last_entry->key);
          free(last_entry);
      }
    }
```

Add this code ———————
```
    if (last_entry) {            /* last_entry left dangling */
        free(last_entry->key); /* classic off-by-one error */
        free(last_entry);      /* free the last one        */
    }
 }
 free(ht);
}
```

2 Recompile the program and run it again.

This time Purify should indicate that there are no leaks. You have successfully fixed the problem.

Using the new leaks button

In longer-running Purify'd programs, you can generate a new memory leaks summary while the program is running by using the New Leaks button.

Click to generate a new leaks summary while the program is running

For example, if you are testing an X Windows word-processing program you might:

- Start the program.
- Click the New Leaks button.

  At this point, Purify might report that there are no new leaks.

- Perform some action with the program. For example, open a document.
- Click the New Leaks button again.

  If Purify reports new leaks at this point, you know that they occurred in the document-opening phase of your program.

In this way, you can isolate memory leaks that occur in a complex program.

Note: The New Leaks button can generate a leak summary only when a program is running, not while it is stopped in the debugger. To get a new leaks summary from the debugger, call the `purify_new_leaks` function directly from the debugger.

Disabling memory leaked messages

If you do not want Purify to display a memory leaked message when the program exits, set the option `-leaks-at-exit=no`.

This inhibits the automatic call of `purify_all_leaks` when the program exits. See Chapter 11, "Using Purify Options and API Functions" for details about how to use Purify options.

5

Analyzing File Descriptors

File descriptors represent handles to input and output streams available to a running program. They are small integer indices into a fixed table in the kernel's per-process data structure.

A common problem occurs when a file is opened within a program loop, or in response to user input, and is never closed. When no more file descriptors are available, programs usually fail, reporting a mysterious inability to open a file that can apparently be opened.

Purify displays a File Descriptors in Use (FIU) message when your program exits, to help you discover such cases. Purify reports each open descriptor and where it was opened.

File descriptors in use messages

When a program starts, it can inherit file descriptors from a parent process. The origin of such inherited descriptors is invisible to Purify, and the message indicates only `<inherited from parent>`. Descriptors `0`, `1`, and `2` are often used in this manner to provide `stdin`, `stdout`, and `stderr`, respectively, for programs. If these descriptors are marked as inherited descriptors, the `stdin`, `stdout` and `stderr` synonyms are attached.

If a file is open, Purify notes the file's name and mode in the message. If it is not a special file, Purify might also be able to determine the current file offset where the next byte would be read or written. The call chain shows where in the program the file was opened.

If a file descriptor is obtained from `socketpair`, `pipe`, or certain other system calls, Purify shows the call chain indicating the origin. No additional information is available.

If you duplicate a file descriptor from another file using the `dup` or `dup2` system calls, Purify notes the call chain of the `dup`, but copies any other file information available from the source of the `dup`.

Purify prints a message for all file descriptors for which `select` or `poll` returns no error. Under some circumstances (for example, descriptors obtained by `ioctls` issued to non-standard device drivers), Purify might not be able to determine details about the origin of the file, and simply prints the text `<unknown>`.

Note: Purify reserves file descriptors 26 and 27 for its own use. To change Purify's reserved file descriptors, use the `-fds` option. See "File descriptor options" on page 12-12.

File descriptor leak example

Consider this example of a file descriptor leak:

File descriptors for standard I/O streams inherited from the shell

File descriptor used by the program

Files opened here

return without closing here

File descriptors reserved for Purify

Purify indicates that 7 file descriptors are in use at the end of the program execution. Two of these are file descriptors opened by the program.

Look at the function `get_number_from_file`. These files are opened by the call to `fopen`, and should be closed by the call to `fclose`. Notice, however, that if the file does not contain a valid number, the function returns without closing the opened file. To correct this file descriptor leak, add a call to `fclose` before this `return`.

Analyzing FIU messages

A *safe* FIU message shows the three standard I/O streams and the two Purify internal file descriptors. You do not need to be concerned with a few additional file descriptors open at exit, if they are allocated from functions called only once in the program.

If more than one file descriptor allocated with the same call chain is still open at exit, it can indicate a program error and you should investigate it. In the previous example, if the function get_number_from_file were called on a large number of files, the program could run out of file descriptors.

Note: Purify's file descriptor data structures are shared across parent and child when the child is created using a vfork. If the child process manipulates files descriptors, it can result in erroneous messages about the origin of the parent's descriptors when the parent exits.

Disabling FIU messages

If you do not want Purify to display an FIU message when the program exits, set the option -fds-inuse-at-exit=no. This inhibits the call of purify_all_fds_inuse when the program exits.

Use the Purify API function purify_clear_fds_inuse to ignore file descriptors that have been opened since the last call to a file descriptor API function. These file descriptors are not reported by the next call to purify_new_fds_inuse. They will however be reported by purify_all_fds_inuse.

For a complete description of the file descriptor API, see "File descriptor API" on page 12-12.

Notes and limitations

By default, Purify generates FIU messages before calling any cleanup functions you may have registered with atexit(3); therefore, Purify reports file descriptors closed by such functions as in-use.

6

Customizing Purify

This chapter explains how to customize Purify. It includes:

- Controlling Purify output
- Mailing Purify output to developers
- Annotating Purify output
- Customizing Purify messages
- Customizing the thread summary message
- Enabling just-in-time debugging
- Reporting Purify status at exit
- Running shell scripts at exit
- Customizing the Purify Viewer
- Customizing Purify scripts
- Managing cached object files

Controlling Purify output

By default, Purify displays its output in the Purify Viewer. However, you can also direct Purify output to a log file as ASCII text, or to a compact-binary *view file* that you can open later in the Viewer.

You can generate any two or all three forms of output from the same run by setting the appropriate options. For a complete list of options that control Purify output, see "Output mode options" on page 12-21.

Saving Purify output as ASCII text

If Purify cannot connect to an X Window display or if you specify the option `-windows=no`, Purify automatically generates ASCII text to `stderr`, interleaving error messages with the program output. In text mode, Purify discards repeating error messages by default rather than counting them.

Using shell file redirection syntax

You can redirect Purify output to a text file using standard shell file redirection syntax. For example:

```
csh     % a.out.pure >& a.out.messages

sh, ksh $ a.out.pure 2> a.out.messages
```

Creating a log file automatically

Use the `-log-file` option to automatically redirect Purify messages to a log file. For example:

```
-log-file=<filename>.plog
```

You can use conversion characters in log file names. See "Using conversion characters in filenames" on page 11-2.

Purify sends the same information to the log file that you see in the Viewer. Messages are printed fully expanded; however, to avoid excessive report size Purify does not expand source files, even when line-number and filename information is available.

When you use the `-log-file` option, Purify does not display the Viewer by default. To save Purify output to a log file and also display the Viewer, use the `-windows=yes` option along with the `-log-file` option.

Set the `-log-file` option to `-log-file=stderr` to send Purify output to `stderr`.

Use the `-output-limit` option to restrict the size of the log file and conserve disk space. The `-output-limit` option specifies the maximum size of the Purify message in bytes. Purify truncates all output beyond this size.

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 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 **Save As** from the File menu.

3 Type a filename, including the extension `.pv`, to identify the run as a Purify view file.

Creating a view file automatically

You can automatically save Purify output to a view file without starting the Viewer. This is convenient when you want to run a set of nightly tests under Purify, then review the results the following morning.

To automatically save Purify output to a view file, set the option:

```
-view-file=<filename>.pv
```

You can use conversion characters in view file names. See "Using conversion characters in filenames" on page 11-2.

When you use the `-view-file` option, Purify does not display the Viewer by default. To also display the Viewer while saving output to a view file, use the `-windows=yes` option. For a description of the `-windows=yes` option, see "Output mode options" on page 12-21.

Opening a view file

To open a view file from the Viewer:

1 Select **Open** from the File menu.

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. For example, you can compare it to other runs and apply suppressions.

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.

Prestarting the Viewer

You can use the `-view` option to prestart the Viewer in order to set options before running a Purify'd program.

```
% purify -view a.out
```

This opens an empty Viewer for the program `a.out` where you can preset options such as suppressions, then open a view file or run a Purify'd program.

To prestart a Viewer on a different screen, type:

```
% purify -view -display=<other_machine>.0 a.out
```

Mailing Purify output to developers

You can use the `-mail-to-user` option to have Purify automatically send the output from a run of a Purify'd program directly to other developers instead of displaying it in the Viewer. Purify sends the output in a log-file format.

Use the `-mail-to-user` option when:

- You are doing nightly tests and want the results sent automatically to other developers.
- You distribute a Purify'd program to other developers and want the output sent to you when they run the program. (The Purify'd program must be run locally where Purify is installed and be on the same operating system as the one where the program was Purify'd.)

Using the -mail-to-user-option

You can use the `-mail-to-user` option to have Purify send Purify messages to one or more developers. For example:

```
% purify -mail-to-user=chris cc ...
% purify -mail-to-user=chris,pat cc ...
% purify -mail-to-user=devgrp cc ...
```

By default, Purify does not open the Viewer when you specify the `-mail-to-user` option. To also display messages in the Viewer, use the `-windows=yes` option along with the `-mail-to-user` option. For a description of the `-windows=yes` option, see "Output mode options" on page 12-21.

Protecting your run-time option settings

To make sure that the options you specify when you build the program remain in effect when the program is run in other locations or by other developers, use the `-ignore-runtime-environment` option with the `-mail-to-user` option.

This prevents other run-time environments from affecting your program, so developers see exactly what you want them to see. No matter what directory your program is run in or who runs it, the run-time options (including suppressions) built into the program cannot be changed.

For a complete description of the `-mail-to-user` option, see "Mail mode option" on page 12-13. For more information about when to use the `-ignore-runtime-environment` option, see "Using the -ignore-runtime-environment option" on page 11-6.

Annotating Purify's output

You can annotate Purify's output in order to help reproduce a particular run, track down an error, or relate events or phases in your program to error messages. You can also record environmental or situational details into a log file or view file to help decode the results of a test run.

For example, you can record the current directory name and the command line arguments into a log file or view file when running a batch of tests. This allows you to see how to repeat a particular run of your program if that particular run generated an error.

In an interactive program, you might record the commands issued by the user into Purify's output in order to identify the user interactions that triggered the Purify messages.

You can use these functions to annotate Purify's output:

- Use `purify_printf(const char *fmt, ...)` to add an annotation to all forms of Purify output.
- Use `purify_logfile_printf(const char *fmt, ...)` to add an annotation if the output goes to a log file.

  If the output is going to a view file, the annotation from `purify_logfile_printf` is recorded in the view file but not displayed. To print the annotation in the Viewer, specify:

  ```
  % purify -view -logfile=<filename>.plog <filename>.pv
  ```

- Use `purify_printf_with_call_chain(const char *fmt, ...)` to add an annotation similar to `purify_printf`, and also include the call chain. This has the effect of counting as an error, triggering mail if `-mail-to-user` is set. You can use this function from within your error handler functions. It is useful for copying the error message to the log file and reporting the call chain that caused the error to happen.

Each of these functions takes a format string and a variable number of arguments, just like `printf`.

Note: Purify does *not* support the full `%` conversion syntax of `printf`. You can use the simple conversion characters `%d`, `%u`, `%n`, `%s`, `%c`, `%e`, `%f`, or `%g`. No field width or precision specifiers are allowed, and the `%e, %f, %g` conversion characters are equivalent to `%10.2f`.

For more information about functions for annotating Purify output, see "Annotation API" on page 12-9.

You can also annotate Purify output by using the option `-copy-fd-output-to-logfile`. The option's value is a list of file-descriptor numbers, for example: `1,2`. This causes Purify to interleave all output written to file descriptors `1` and `2` with the Purify output.

For more information about the `-copy-fd-output-to-logfile` option, see "Annotation options" on page 12-9.

Customizing Purify messages

You can change the content and appearance of Purify messages, and control how Purify batches messages.

Controlling the content and appearance of messages

You can control the content of messages by defining the number of stack frames in the call chain of a message, and whether Purify prints full pathnames in the call chain. You can also decide whether to display various messages such as memory leaks, memory in use, and file descriptors in use when your program exits.

For a complete list of options for customizing the appearance of Purify messages, see "Message appearance options" on page 12-18.

Controlling message batching

By default, Purify displays an error message the first time the error occurs, along with a count of all subsequent occurrences of the same error. Purify considers errors to be the same if they have identical call chains.

If you are generating ASCII text in the default first-only mode, `-messages=first`, Purify does not increment counts of repeated errors. You can display all the messages Purify generates, in the order in which they occur, by setting the option `-messages=all`.

By default, Purify does not update the counts on the display continuously, because that would impact performance. Instead, Purify updates counts only when new messages are displayed. If you need to correlate repeated occurrences of errors with program activity interactively, set `-messages=all` to enable the display of all messages.

You can have Purify wait until the program terminates before displaying messages, by setting the option `-messages=batch`. Purify does not batch report entries generated when you use log file

functions such as `purify_logfile_printf`; it reports them immediately.

For a complete description of the `-messages` option, see "Message batching options" on page 12-19. For a complete list of Purify functions to control message batching, see "Message batching API" on page 12-20.

Customizing the thread summary message

Threads are separate, independent execution sequences within a single process. They share the same address space but maintain separate execution stacks.

By default, Purify displays a thread-summary message if your program is linked to a supported thread package. Purify displays the thread summary message after the memory leak summary.

For a list of supported thread packages, see the README file.

Purify tracks all the threads that are created during the execution of the program. The thread summary message contains a description of each thread including its thread id, name, and stack limits.

For example:

```
Thread Summary: 6 threads in existence
        Thread 1
        Stack: (0xef106ad8 0xef106dcc), size = 0x2f4
        Thread 2
        Stack: (0xef7b11f8 0xef7b143c), size = 0x244
        Thread 3
        Stack: (0xef005728 0xef005dcc), size = 0x6a4
        Thread 4
        Stack: (0xef207ac8 0xef207d6c), size = 0x2a4
        Thread 5 "Producer"
        Stack: (0xeee03410 0xeee03d6c), size = 0x95c
        Thread 6 "Consumer"
        Stack: (0xeef04498 0xeef04d6c), size = 0x8d4
```

Purify assigns each thread an id in order to keep track of running threads. The Purify thread id is unrelated to any id defined by the thread library.

You can specify a name for Purify to use in addition to the thread id by using the API function `purify_name_thread`. See "Threads API" on page 12-28.

How Purify identifies threads

Purify uses the stack pointer to determine the identity of a thread. When Purify notices a change to the stack pointer, it compares the new value to the stack areas of known threads. Purify assumes that a new thread has been created if the stack pointer changes by 0x1000.

You can specify how large the change to the stack pointer has to be to mark creation of a new stack by using the option `-thread-stack-change`. See "Threads options" on page 12-27.

Enabling JIT debugging

With Purify's just-in-time (JIT) debugging, you can use your debugger to investigate errors even when you run your application from outside the debugger. You can have Purify automatically attach a debugger to your application when selected types of Purify messages are reported, or have Purify ask you if you want to start a debugger at the time of the error.

You can also use JIT debugging to start your debugger when it encounters a watchpoint message. Purify stops just before the watchpoint.

Warning: If your Purify'd program is already running under a debugger, do *not* enable JIT debugging. Attempting to do so will cause the program to fail.

To enable just-in-time debugging:

1 Select **JIT Debug** from the Options menu.

Purify opens the Debugging Options dialog:

Select types of messages

You can have Purify start your debugger automatically when it encounters an error, or ask you before starting it

Type command-line arguments for your debugger
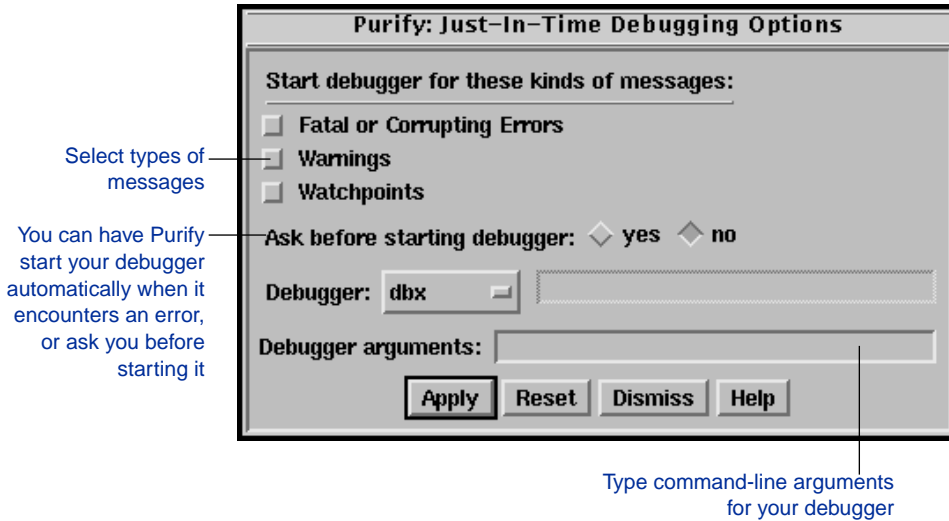
2 Select the types of messages for which you want Purify to start your debugger: fatal or corrupting, warnings, or watchpoints.

For a description of fatal, corrupting, and warning error messages, see "Message severity" on page 10-2. For information about setting watchpoints, see Chapter 8, "Setting Watchpoints."

3 Select the debugger you want Purify to start.

You can also type command-line arguments for your debugger.

Note: You can also enable JIT debugging using the `-jit-debug` option. See "Miscellaneous options" on page 12-31. You can change the list of available debuggers, and Purify's interface to them, using your `~/.purify.Xdefaults` file and the `pure_jit_debug` script which is located in the Purify installation directory. See "Customizing Purify scripts" on page 6-16.

Reporting Purify status at exit

By default, Purify does not modify the normal exit status of your program. However, you can have your program exit with a special exit status if Purify finds any access errors or memory leaks. This is a convenient way to flag failing runs in test suites.

Use the option `-exit-status=yes` to enable Purify to insert flags into your program's status on exit that indicate types of run-time errors. These are:

Types of unsuppressed Purify errors	Bit OR'ed in exit status
Memory access errors	0x40
Memory leaks	0x20
Potential memory leaks	0x10

Alternatively, you can replace the call to `exit(status)` in your code, or the `return status` in `main`, with a call to the `purify_exit(status)` function. If you are concerned only about memory access errors, you can either turn off leak detection at exit using the option `-leaks-at-exit=no`, or you can suppress memory leak and potential leak messages. You can also ignore the appropriate bits of exit status.

The program summary message from Purify shows your original exit status before any other bits are OR'ed in.

For a description of exit-processing options, see "Exit processing options" on page 12-10. For a description of the `purify_exit` function, see "Exit processing API" on page 12-11.

Running shell scripts at exit

You can invoke an arbitrary shell script when your program exits or otherwise terminates by using the `-run-at-exit` option. For example, you can use a script to move or remove log files or view files, to note test failure, or to interact with PureCoverage data.

In addition to the `%V`, `%v`, and `%p` conversion characters described in "Using conversion characters in filenames" on page 11-2, you can use these conversion characters:

Character	Converts to
`%z`	The string value "true" or "false" indicating whether any call chains were printed, for example, in error or leak reports
`%x`	The program's exit status (0 if the program did not call `exit`)
`%e`	The number of distinct access errors printed
`%E`	The total number of error occurrences noted
`%l`	The number of bytes of memory leaked
`%L`	The number of bytes of memory potentially leaked

Use the `%z` sequence to have your exit script act conditionally when Purify finds something of interest to you. If a call chain is printed in an access error, a memory leak, or as a result of calling `purify_printf_with_call_chain()`, the string is `true`; otherwise, it is `false`.

For example, if you set the option:

```
setenv PURIFYOPTIONS '-run-at-exit="if %z ; then \
    echo \"%v: %e errors, %l+%L bytes leaked.\" ; fi"'
```

When your program exits, you might see on `stdout`:

```
testprog: 2 errors, 1+10 bytes leaked.
```

Note: If your program is running in the Viewer, Purify sends any command output to the X Window where you started the Viewer.

For a description of exit-processing options, see "Exit processing options" on page 12-10. For a description of the purify_exit function, see "Exit processing API" on page 12-11.

Customizing the Purify Viewer

When you exit your first Purify session, Purify automatically creates a .purify.Xdefaults file in your home directory. You can edit this file directly from the Viewer in order to customize the Viewer. For example, you can:

- Define the number of lines of source code displayed in a message
- Change background and highlight colors, and the size and color of fonts
- Color code messages

Purify simplifies the task of customizing the Viewer by letting you edit the .purify.Xdefaults file directly from the Viewer:

1 Select **Edit X resources** from the Options menu to open the .purify.Xdefaults file.

2 Edit the .purify.Xdefaults file using your editor.

The .purify.Xdefaults file uses a typical .Xdefaults file format. For more information about editing an .Xdefaults file, see your *X Window System Users Guide*.

3 Restart the Viewer for the changes to take effect.

Customizing Purify scripts

Purify includes several scripts in the `<purifyhome>` directory:

- `pure_invoke_ddts` starts the PureDDTS application when you click the PureDDTS icon ⊕ in the toolbar.
- `pure_invoke_purecov` starts the PureCoverage application when you click the PureCoverage icon ▨ in the toolbar.
- `pure_run` starts a new run of the current Purify'd program when you click the Run button in the program controls.
- `pure_debug` starts a debugger when you click the Debug button in the program controls.
- `pure_edit` starts an editor when you click the Edit program control or the Edit ▨ button.
- `pure_print` sends an ASCII version of your Purify run to the printer when you select **Print** from the File menu.
- `pure_jit_debug` starts just-in-time debugging when you select **JIT Debug** from the Options menu.

For more information about the scripts for the program controls, see "Customizing the program controls" on page 6-17.

To customize a Purify script:

1 Copy the script to a separate directory.

2 Edit the script.

3 Set your path to include the path of the modified script.

Customizing the program controls

The program controls allow you to control the building and execution of your Purify'd application. To display the program controls toolbar, select **Program controls** from the View menu.
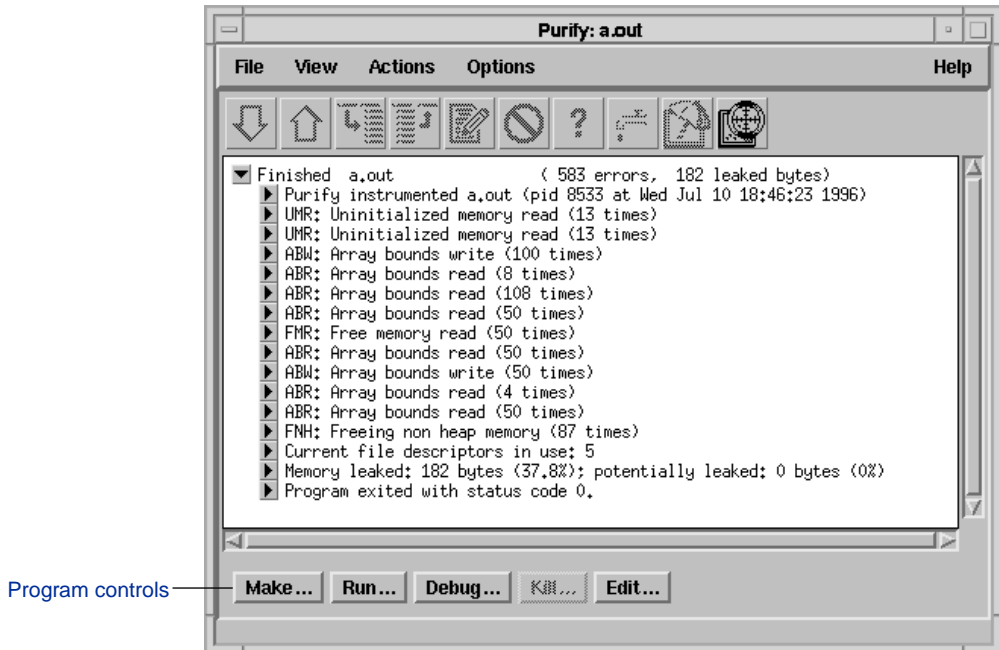
Program controls

Each program control button invokes a default command. To edit the scripts for the program controls, see "Customizing Purify scripts" on page 6-16.

> Make...

Runs the shell command `pure_run make <program-name>`, where `<program-name>` is the name of the Purify'd program currently being viewed.

The Make button uses the `pure_run` script to run the command in a new window. The terminal window remains open after the make is completed. To close the window, press RETURN. You can change how the new window is created by editing the `pure_run` script.

Run...

Starts a new run of the Purify'd program currently being viewed. The first time you click Run, the command is filled in with the previous arguments to the program, if known. If Purify is currently viewing a running program, the Run button is inactive.

You can change the behavior of the Run button by editing the `pure_run` script.

Debug...

Starts a debugger on the currently viewed Purify'd program, in a new terminal window. If the program is currently running, the debugger attaches to it. You can specify the debugger you want Purify to start by editing the `pure_debug` script.

Kill...

Runs the shell command `kill %p` to kill the currently running Purify'd program. The Kill button is active only when a Purify'd program is running. Each time this dialog is used, the `%p` is expanded into the current process id (`pid`).

To send a different signal to your program, you can modify the command before clicking OK. For example, to terminate a program with extreme prejudice, modify the command to be `kill -9 %p`.

Edit...

Starts an editor on the specified file in a new terminal window. You can specify the editor you want Purify to start by editing the `pure_edit` script.

Warning: The Edit 🖊 button also uses the `pure_edit` script. Therefore, use caution when you modify this script.

Managing cached object files

To improve build-time performance, Purify caches its instrumented versions of all the libraries and object files that are used by the program. When you re-build a program, Purify updates only the new or modified files; otherwise it uses the cached versions.

You can identify an instrumented cache file by its name. It includes _pure_, a Purify version number, and might also include information about the size of the original file, or the name and version number of the operating system.

Purify writes Purify'd files to the original file's directory if that is writable, or to the global cache directory otherwise.

Purify lets you control how instrumented libraries and files are cached. You can:

▪ Specify the global cache directory

▪ Direct Purify to save all cache files in the global cache directory

▪ Restrict Purify from caching files in certain directories

See "Build-time options" on page 12-6.

Deleting cached object files

Since Purify rebuilds cached instrumented files as needed, you can remove them at any time in order to conserve disk space.

Using the pure_remove_old_files script

To remove cache files, use the pure_remove_old_files script located in the <purifyhome> directory. For example, to remove all instrumented cache files that are 14 days or older:

```
% pure_remove_old_files / 14
```

The first argument (/) specifies the path, the second argument (14) specifies the number of days. This command removes files 14 days or older recursively from the root directory /.

Using a cron job

To automate the removal of cache files, create a `cron` job that periodically removes the files. For example, to remove files that have not been accessed in two weeks, type:

```
% crontab -e
```

Add this entry to the `crontab` file:

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

This runs `pure_remove_old_files` each day at 2:15 am, and removes all cached files starting at the root directory `/` that have not been read within the last 14 days.

To remove all of the cache files in the current directory and subdirectories, use:

```
% pure_remove_old_files . 0
```

This is useful in `clean` targets of makefiles.

7

Suppressing Purify Messages

You can prevent messages from being displayed in the Viewer by suppressing them. When you suppress error messages, Purify still checks all of the code in your program, but it doesn't display the suppressed messages. Suppressing messages is useful when:

- You cannot correct an error, such as an error in a third-party library for which you do not have the source code
- You want to focus on specific errors

You can suppress messages directly from the Viewer or by specifying suppression directives in a `.purify` file.

This chapter explains how to suppress Purify messages. It includes:

- Suppressing messages in the Viewer
- Making suppressions permanent
- Specifying suppressions in a `.purify` file
- Viewing suppressed messages
- Removing and editing suppressions
- Temporarily unsuppressing messages
- Sharing suppressions between programs
- Using the `-suppression-filenames` option

Suppressing messages in the Viewer

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

To suppress an error message in the Viewer:

1 Select the message you want to suppress.

2 Select **Suppressions** from the Options menu.

This opens a Suppressions dialog containing information about the selected message.

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

The suppression directive

When you make a suppression permanent, Purify adds the suppression directive to the `.purify` file

You can save the suppression directive to another `.purify` file
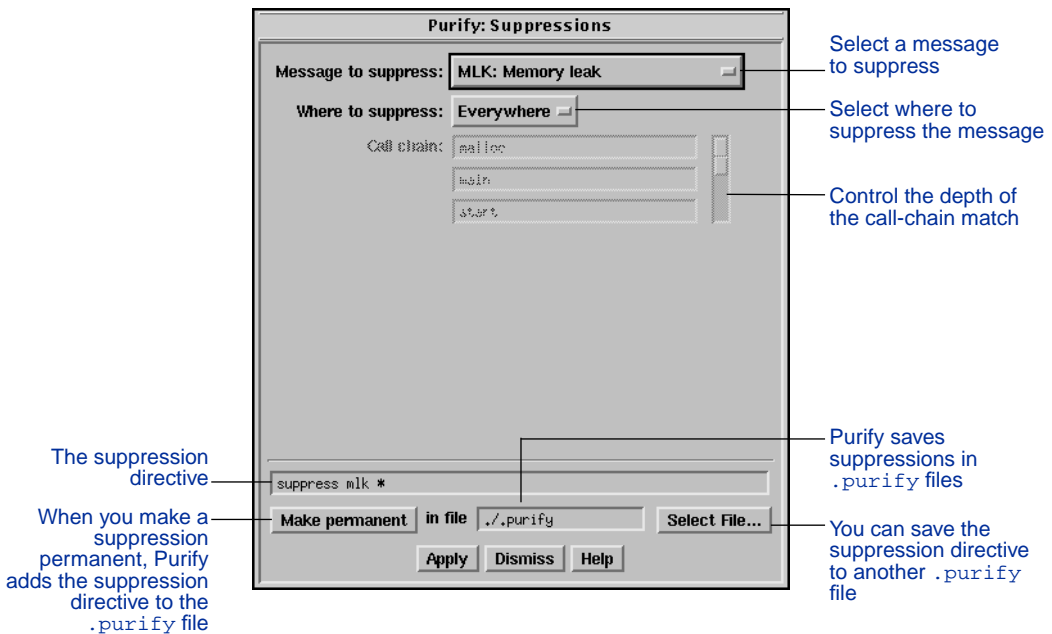
Suppressions created in the Viewer take precedence over suppressions in `.purify` files. However, they apply to the current Purify session only. Unless you click **Make permanent**, they do not remain when you restart the Viewer.

Selecting where to suppress a message

For **Where to suppress**, you can select:

- **In call chain:** Suppresses a message in a specific call chain. Use the slider to change the number of functions to match against.
- **In file:** Suppresses a message in the source or object file containing the last function in the call chain.
- **In library:** Suppresses a message in the library containing the last function called in the call chain. This option is available only if the function is obtained from an archive or a shared library.

    Note: Do *not* suppress errors in third-party code unless you are sure that the library itself is at fault. Often, errors that appear in library functions are caused by a program's misuse of those functions, or misinterpretation of the programming interface. If you are sure the library is at fault, report it to the library vendor.

- **In class:** For C++ member functions, this suppresses messages in the class containing the last member function in the call chain.
- **Everywhere:** Suppresses the message in all locations where it occurs.

Making a suppression permanent

To make a suppression permanent, click **Make permanent** in the Suppressions dialog. This saves the specified suppression directive to the `./.purify` file.

Saving a suppression directive to another .purify file

To save a suppression directive to another `.purify` file and have it also apply to the current program:

1 Click **Select file** in the Suppressions dialog.

This opens the File Selection dialog.

2 Specify where you want to save the suppression.

For information about how to suppress messages across multiple programs, "Sharing suppressions between programs" on page 7-9.

Specifying suppressions in a .purify file

You can specify suppressions directly in a `.purify` file. The syntax for specifying suppressions in a `.purify` file is:

```
suppress <message-type> <function-call-chain>
```

For `<message-type>`, specify the three-character acronym for the message to be suppressed. You can use upper or lower case, and wildcards. For example, AB* matches ABR and ABW, while *W matches ABW, FMW, IPW, NPW, SBW, WPW and ZPW. For a list of Purify message acronyms, see "Message quick reference" on page 10-1.

For `<function-call-chain>`, specify a list of function names separated by semi-colons. You can replace or augment a function name by specifying a filename enclosed in double quotes. Purify compares the filename to source, object, or library names. You can use wildcards for function and filenames. An unqualified filename (containing no '/') matches the specified file in any directory; a qualified filename must match exactly to suppress the message.

For example,

```
suppress ABR sortFunction; sort*; qsort "libc*"
```

suppresses ABR messages in the `sortFunction` function whenever it is called from a function with the prefix `sort`, which in turn is called from the `qsort` function in the `libc` library in any directory.

Using "..." syntax

You can use `"..."` syntax for floating or unanchored matches. For example,

```
suppress UMR ...; Xm*
```

suppresses UMR messages in any function beginning with `Xm` or any function called directly or indirectly from such a function.

You can use `"..."` syntax in the middle of a chain. For example,

```
suppress ABW tzsetwall; ...; main
```

suppresses ABW messages in `tzsetwall` when called at any level below `main`.

Using `"..."` syntax is more time-consuming than simple pattern matches because it causes Purify to match every address in the call chain with the function name. Without `"..."`, Purify can usually make a match at the first name.

Note: `"..."` reaches only as far as the recorded call chain. For example, Purify does not suppress a UMR message that is 27 levels below `XmInit`, if the `-chain-length` option is set to the default of 6.

Suppressing error messages in C++ code

To suppress messages in C++ code, use natural C++ notation, including the function or method argument signature. You cannot use a mangled function name as reported by tools like `nm`.

If you do not include the argument signature, Purify matches any overloaded function of the specified name. For example,

```
suppress FNH Test::Test
```

suppresses FNH errors in all `Test::Test` functions regardless of their argument signature. However,

```
suppress FNH Test::Test()
```

suppresses *only* the error in `Test` with zero arguments. So, for example, it would not suppress errors in a `Test::Test(const char*)`function.

Suppressing messages in the Hello World example

In the Hello World program example discussed in Chapter 2, Purify reports an ABR error. You can suppress this message by using any of the following suppression directives:

- To suppress all ABR errors in _doprnt, use:

  ```
  suppress ABR _doprnt
  ```

- To suppress ABR errors in _doprnt, called from printf, called from main, use:

  ```
  suppress ABR _doprnt; printf; main
  ```

- To suppress all ABR errors in the libc library, use:

  ```
  suppress ABR "libc*"
  ```

- To suppress array bounds errors from descendants of printf, use:

  ```
  suppress AB* ...; printf
  ```

- To suppress all errors in descendants of libc, use:

  ```
  suppress *    ...; "libc*"
  ```

Displaying suppressed messages

To display a message that has been suppressed, select **Suppressed messages** from the View menu.

Purify displays the message with the identifier [Suppressed] in front of it. You can expand a suppressed message to see the location of the .purify file containing the suppression directive as well as the details of the message itself.

Removing and editing suppressions

Note: If you know the location of the `.purify` file containing the suppression, you can edit that file directly without starting the Viewer.

To remove or edit a suppression from the Viewer:

1 Select **Suppressed messages** from the View menu to display the suppressed messages.

2 Expand the suppressed message.

3 Click the Edit 🖊 button in the suppressed message.

Purify opens the editor positioned at the line matching the suppression in the `.purify` file.

4 Edit the `.purify` file:

To remove the suppression, delete the line or add a `#` to comment out the suppression directive. For example:

```
# suppress ABR _doprnt
```

5 Save your changes and exit the editor.

6 Select **Re-read .purify files** from the File menu.

Purify redisplays the message.

Temporarily unsuppressing messages

You can use the `unsuppress` directive to temporarily override a suppression in a `.purify` file. The last directive specified in the `.purify` file takes precedence over earlier directives. If the last directive that matches a particular message type and call chain is an `unsuppress` directive, Purify does *not* suppress the message.

Using the unsuppress directive

To use the `unsuppress` directive:

1 Select **Suppressions** from the Options menu.

This opens the Suppressions dialog.

2 Edit the suppression directive by changing the word `suppress` to `unsuppress`.

You can also specify the `unsuppress` directive directly in a `.purify` file. See "Removing and editing suppressions" on page 7-7.

Use the same syntax for the `unsuppress` directive as you use for the `suppress` directive. For example, adding

```
unsuppress umc *
```

to the `./.purify` file unsuppresses all the UMC messages that are suppressed by default in `<purifyhome>/.purify`.

You can also unsuppress a message in a specific function. For example, if you specify

```
unsuppress umc this_func
```

Purify displays all of the UMC messages that occur in `this_func`. All other suppressed messages remain suppressed.

For more information about syntax for specifying suppression directives in a `.purify` file, see "Specifying suppressions in a .purify file" on page 7-4.

Note: Avoid using the `unsuppress` directive to permanently remove suppressions. It clutters your `.purify` file. To permanently remove suppressions, see "Removing and editing suppressions" on page 7-7.

Sharing suppressions between programs

Purify maintains suppressions in .purify files. You can share suppressions between programs by copying suppressions into directories used by other programs, and by creating standard suppressions for your entire site.

Suppression precedence

Purify reads suppressions from all .purify files, processing them in the order listed below (highest precedence first). Within each .purify file, the suppression directive at the end of the file takes precedence.

- To create suppressions for all Purify'd programs in a specific directory, use the .purify file in that directory. When you click **Make permanent** in the Suppressions dialog, this is where Purify saves the suppression directive by default.

- To create suppressions for all Purify'd programs that you run, use the .purify file in your home directory, $HOME.

- To create suppressions for all Purify'd programs at your site, use the default .purify file in the <purifyhome> directory. If this file is not writable, request that someone with permission add the directive.

  By default, Purify suppresses UMC errors in the <purifyhome>/.purify file. You can unsuppress UMC errors in a location with higher precedence than this one.

Creating suppressions for specific operating systems

To create suppressions for a specific operating system, use the .purify.sunos4, .purify.solaris2, .purify.hpux, or .purify.irix files in any of the above three locations. These files take precedence over other suppressions in the same directory.

Using the -suppression-filenames option

You can use the `-suppression-filenames` option to instruct Purify to look for suppressions in one or more specified files. This is helpful when you want to create different suppressions for two programs located in the same directory.

For example, if you specify:

```
% purify -suppression-file-names=".purify,.purify.sunos4,\
     $HOME/purify_suppressions"
```

Purify reads suppressions from:

```
<purifyhome>/.purify
<purifyhome>/.purify.sunos4
$HOME/.purify
$HOME/.purify.sunos4
<progdir>/.purify
<progdir>/.purify.sunos4
$HOME/purify_suppressions
```

If you specify an unqualified filename (containing no '/' characters) in the `-suppression-filenames` option, Purify looks for that file in the `<purifyhome>`, `$HOME`, and program directories. Purify interprets qualified filenames as absolute or relative to the current working directory.

The default setting for the `-suppression-filenames` option depends on your operating system. For a list of the default settings for each operating system, see "Suppression options" on page 12-26.

8

Setting Watchpoints

You can monitor a region of memory for specific kinds of memory accesses by setting a Purify watchpoint on it. Using watchpoints simplifies the task of debugging problems where memory mysteriously changes between the time it is initialized and the time it is used.

Purify watchpoints can report:

- Reads (WPR)
- Writes (WPW)
- Allocations (WPM)
- Frees (WPF)
- Coming into scope at function entry (WPN)
- Going out of scope at function exit (WPX)

When you set a watchpoint, Purify automatically reports the exact cause and result of each memory access, even when you are not using a debugger. Since Purify already intercepts every memory access as part of its dynamic error detection, you can use watchpoints without any performance degradation.

When to use watchpoints

Watchpoints are useful when:

- Memory is being improperly freed. Set a watchpoint on the memory to have Purify report when the memory is allocated and freed, and who is doing the allocation and freeing.
- System calls fail intermittently. Set a watchpoint on the global system variable `errno` to have Purify report a diagnostic message whenever `errno` is written.

- A counter is not incrementing properly. Set a watchpoint on the counter and wait for the improper change.
- Global data is being overwritten improperly. Set a watchpoint on the global memory and wait for writes to that region.
- A segment of read-only data is being changed. Set a watchpoint on the memory segment that should not change during execution. The memory segment does not need to begin on page boundaries, and there is no limit on the size of the memory segment that Purify can watch.

Why use Purify's watchpoints?

Watchpoints in debuggers such as gdb are implemented by single-stepping the program, and checking whether the value of a watched variable changes after each instruction. Under gdb, watching a single 4-byte word can slow the program by a factor of 1,000 or more.

Purify implements watchpoints by monitoring the addresses of the loads and stores performed by the program. This makes using Purify's watchpoints fast; there is no performance loss for requesting watchpoints, even those covering large regions of memory.

Purify's watchpoints are also more sensitive than those of a debugger. Purify warns you when watched data is read, when watched data in the heap is allocated or freed, and when watched data on the stack comes into or goes out of scope as it is included in the local variables of a function. Purify also catches a write to a watched variable when the value being written is unchanged.

Note: You can set a watchpoint and then enable Purify's Just-In-Time debugging feature to start your debugger when Purify encounters the watchpoint. For more information about JIT debugging, see "Enabling JIT debugging" on page 6-11.

Calling Purify watchpoint functions

You can set watchpoints by calling one of Purify's watchpoint functions, either from your debugger or from the program itself. Each watchpoint function takes the address of the beginning of the memory segment to watch and defines a watchpoint over a certain number of bytes in that segment.

Depending on the function, Purify watchpoints trap when any of the specified bytes are read, written, allocated, or freed. For example, the default watchpoint function

```
purify_watch(char *addr)
```

watches 4 bytes of memory, trapping when any of the 4 bytes are written, allocated, or freed. It does not trap when the bytes are read.

Purify provides other functions to handle the common cases of watching writes (and optionally reads) over 1, 2, 4, and 8 bytes. For maximum flexibility, the function

```
purify_watch_n(char *addr, size *size, char *type)
```

watches a segment of `size` bytes starting at `addr`, trapping whenever any of those bytes are allocated or freed and, depending on the `type` argument, trapping when any of those bytes are read (`"r"`), written (`"w"`), or either read or written (`"rw"`).

Purify assigns a number to each watchpoint so you can easily identify it. To print the list of current watchpoints, call the function `purify_watch_info`. To remove a specific watchpoint, call `purify_watch_remove` with the appropriate watchpoint number. Call `purify_watch_remove_all` to remove all the watchpoints.

For a complete list of the watchpoint functions, see "Watchpoint API" on page 12-29.

Stopping at watchpoints in a debugger

To stop at a watchpoint in a debugger, place a breakpoint on `purify_stop_here`. Purify stops at this point for both memory access errors and for watchpoints.
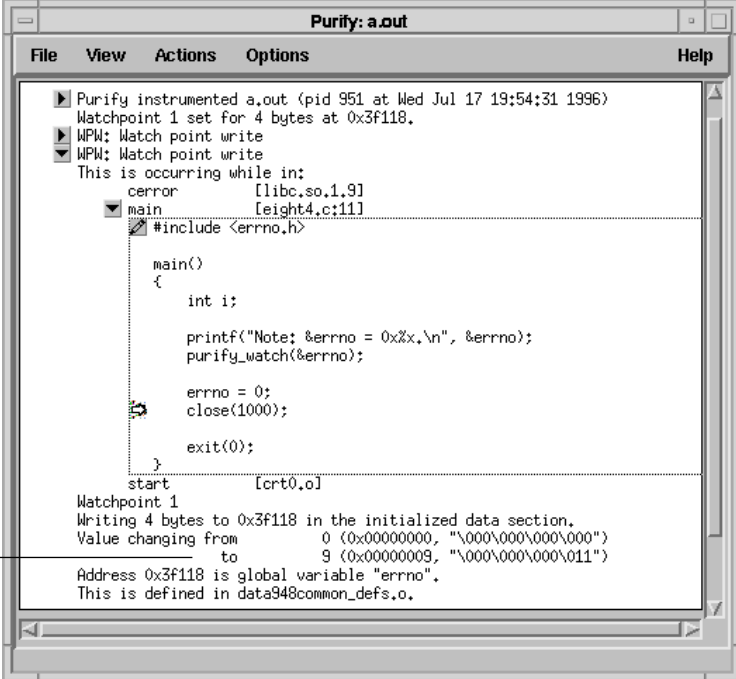
A watchpoint example

In this example, a watchpoint is set in the program using the function `purify_watch` on line 8. This watchpoint monitors all writes to `errno`, a system variable that records error codes for system calls.
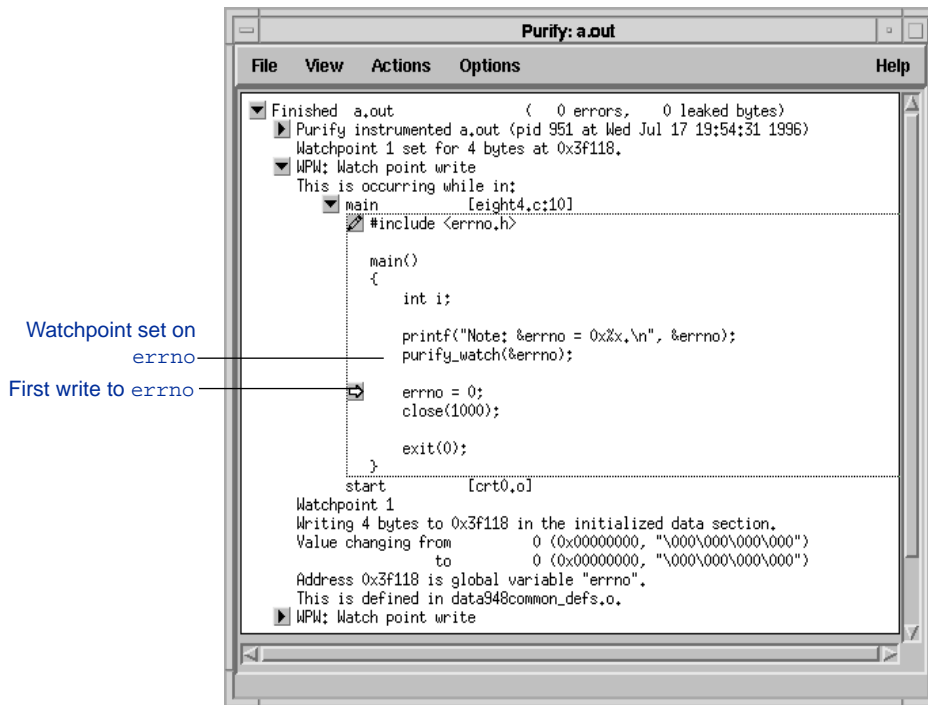
On line 10, `errno` is initialized to 0. On line 11, the system call `close` is invoked using an invalid file descriptor 1000. A watchpoint message is triggered when `errno` is set to 0, and again when the `close` call sets that variable.

The watchpoint message is triggered when `close` is invoked:

The system call `close` sets `errno` to 9

```
Purify: a.out

File   View   Actions   Options                                    Help

▶ Purify instrumented a.out (pid 951 at Wed Jul 17 19:54:31 1996)
  Watchpoint 1 set for 4 bytes at 0x3f118.
▶ WPW: Watch point write
▼ WPW: Watch point write
  This is occurring while in:
      cerror          [libc.so.1.9]
  ▼ main             [eight4.c:11]
      ▨ #include <errno.h>

        main()
        {
            int i;

            printf("Note: &errno = 0x%x.\n", &errno);
            purify_watch(&errno);

            errno = 0;
        ▷   close(1000);

            exit(0);
        }
      start            [crt0.o]
  Watchpoint 1
  Writing 4 bytes to 0x3f118 in the initialized data section.
  Value changing from        0 (0x00000000, "\000\000\000\000")
                       to     9 (0x00000009, "\000\000\000\011")
  Address 0x3f118 is global variable "errno".
  This is defined in data948common_defs.o.
```

Note: The function on the top of the stack in this message is `cerror` instead of `close`. This is because all system calls that set `errno` tail-call the function `cerror`, which sets `errno` and then returns to the original caller of the system function.

Saving watchpoints

Purify automatically saves watchpoints between runs of the Purify'd program in a file called `./<program-name>.watchpoints`. Purify writes this file to the current directory as you set the watchpoints, then reads it when you restart the program.

You can override where Purify reads and writes this file by setting the `-watchpoints-file=<filename>` option:

```
csh     % setenv PURIFYOPTIONS "-watchpoints-file=$HOME/wps"

sh, ksh $ PURIFYOPTIONS="-watchpoints-file=$HOME/wps"; export \
           PURIFYOPTIONS
```

You can use conversion characters in `<filename>`. See "Using conversion characters in filenames" on page 11-2.

To remove these saved watchpoints, you can either delete the `<filename>.watchpoints` file before executing the program, or call the function `purify_watch_remove_all` that removes the watchpoints and the `<filename>.watchpoints` file.

To stop Purify from automatically saving watchpoints, set the `-watchpoints-file` option to an empty filename:

```
csh     % setenv PURIFYOPTIONS "-watchpoints-file="

sh, ksh $ PURIFYOPTIONS="-watchpoints-file="; export PURIFYOPTIONS
```

For a complete description of the `-watchpoints-file` option, see "Watchpoint options" on page 12-29.

Notes and limitations

You can watch memory in the `stack`, `heap`, `bss`, `data`, `text`, and `mmap`'d segments of your program.

You can monitor variables or memory addresses within the scope of a function call (local variables). Purify's watchpoints indicate when the variable comes into or goes out of scope.

Watchpoints work for system calls, but they do not work for kernel trap handlers. If you set a watchpoint on the stack, a trap handler running in kernel mode can set that stack memory without triggering a Purify watchpoint. Code that processes an interrupt signal *does* trigger watchpoints—although the kernel trap handler does not.

Unlike the `.purify` file, Purify reads the `./%v.watchpoints` file only when the program starts up.

In optimized code, the compiler can store a value in a register for later use instead of generating a read each time your program uses that variable. In these cases, Purify's read watchpoints are triggered *only* on the original access of the variable and not by subsequent reuses of the register value.

9

Custom Memory Managers

An important part of Purify's error detection involves tracking `malloc` and `free` calls. If you use custom memory-management layers rather than calling `malloc` or `free` directly, you might need to modify them slightly to make them compatible with Purify.

You can preserve your custom memory manager's semantics and implementation during normal operation, but provide an alternate implementation that uses `malloc` and `free` when the program is Purify'd.

Types of custom memory managers

There are five categories of custom memory managers:

- **Malloc veneers** are layers over `malloc` and `free` that check error returns so that the callers do not have to. You do not need to modify `malloc` veneers when you use Purify because the actual allocation and freeing is still done by `malloc` and `free`.

- **Improved mallocs** are allocation packages that are better than the `malloc` and `free` in the C library. For example, they can reduce memory fragmentation or perform garbage collection. Improved `mallocs` present the same interface as the standard `malloc` and `free` functions.

  You do not need to modify improved `mallocs` when you use Purify because Purify does not replace the `malloc`, it merely intercepts calls to it.

- **Fixed-size allocators** work by requesting a large block of memory using `malloc` and then allocating many fixed size sub-blocks. These allocators are more efficient than `malloc` because the sub-blocks do not need any size overhead.

You must make minor changes to fixed-size allocators when you use Purify. See "Modifying fixed-size allocators" on page 9-3.

- **Pool allocators** allow an application to specify a location identifier in addition to a size when making a memory allocation request, thus reducing the amount of paging. These memory managers can also support pool-level operations, such as printing or freeing an entire pool.

  You must make minor changes to pool allocators when you use Purify. See "Modifying pool allocators" on page 9-5.

- **Sbrk allocators** have application-specific semantics and use the system call `sbrk` to allocate memory rather than using `malloc`. To preserve compatibility with library routines, `sbrk` allocators generally implement a `malloc` interface in addition to their application-specific interface.

  You must make several changes to `sbrk` allocators when you use Purify. See "Modifying sbrk allocators" on page 9-7.

Modifying fixed-size allocators

Suppose you have a special purpose allocator for cells, called `AllocateCell`, that first searches its own free list and then, if needed, calls `malloc` to allocate an additional block to hold one hundred cells. `FreeCell` puts the cell on its own free list.

Since this allocator exists solely for efficiency, the simplest way to modify it for use with Purify is to override the efficiency measure by turning `AllocateCell` and `FreeCell` into the `malloc` veneer category. You can do this by conditionally compiling the allocation code depending on whether or not the program is Purify'd.

```
struct Cell *AllocateCell() {
#ifdef PURIFY
    return (struct Cell *)malloc(sizeof(struct Cell));
#else
    ... original code for AllocateCell ...
#endif /* PURIFY */
}
void FreeCell(struct Cell* c) {
#ifdef PURIFY
     free((char*)c);
#else
    ... original code for FreeCell ...
#endif /* PURIFY */
}
```

Using purify_is_running instead of #ifdef

You can use the compile-time flag `#ifdef PURIFY`; however, it requires that you re-compile your program with the `–DPURIFY` flag set. A simpler solution is to use the `purify_is_running` function to determine if the program is Purify'd. For example:

```
struct Cell *AllocateCell() {
    if (purify_is_running()) {
        return (struct Cell *)malloc(sizeof(struct Cell));
    } else {
        ... original code for AllocateCell ...
    }
}
```

To use `purify_is_running`, add the `<purifyhome>/purify_stubs.a` library to the end of your library list for your link line. This provides a definition of the `purify_is_running` function that returns FALSE when you do not use Purify. When you use Purify, the stubs file is ignored.

Note: The `purify_is_running` function is useful for purposes other than modifying custom memory managers. See "Miscellaneous API" on page 12-33.

If you are using ANSI C, C++, or a compiler that requires function prototypes, include the `<purifyhome>/purify.h` file in your source code in order to declare the `purify_is_running` function. The run-time cost of calling `purify_is_running` is negligible.

Note: Purify includes the source code for `purify_stubs.a` (`pure_stubs.c` and `purify.h`) without copyright. You can include it in the libraries you ship to customers, or compile it on other platforms. This allows you to include calls to Purify API functions throughout development and testing, without having to re-compile or change your code for final shipment.

Modifying pool allocators

Simple pool allocators allow an application to specify a location identifier when allocating memory. More sophisticated pool allocators also support pool-level operations such as freeing or printing an entire pool with one function call.

If you are using a simple pool allocator, you can defeat the pool allocator when the program is Purify'd in the same manner described for fixed-size allocators. See "Modifying fixed-size allocators" on page 9-3.

If you are using a sophisticated pool allocator, you can take advantage of Purify's pool support to implement the pool-level operations such as freeing a pool. Purify can handle each allocation request by calling malloc, and also labeling the returned block with the correct pool-id. Pool-level operations can then be performed later by mapping a function over all blocks with a given pool-id.

As with fixed-size allocators, you can use a run-time flag instead of a compile-time flag. See "Using purify_is_running instead of #ifdef" on page 9-4, for details.

Purify's pool interface functions assume that pool-id is a 32-bit sized datum. For a complete list of functions for pool allocation, see "Pool allocation API" on page 12-23.

```
PoolId AllocatePool() {
    if (purify_is_running()) {
      static PoolId pool_counter = 0;
      /* return unique id, but allocate no mem */
      return pool_counter++;
    } else {
      ... original code for AllocatePool() ...
    }
}
```

```
char* AllocateFromPool(PoolId id, int size) {
    if (purify_is_running()) {
     char* ret = malloc(size);
     purify_set_pool_id(ret, id);
     return ret;
    } else {
     ... original code for AllocateFromPool ...
    }
}

void FreeToPool(char* mem) {
    if (purify_is_running()) {
     free(mem); /* this clears the pool id */
    } else {
     ... original code for FreeToPool ...
    }
}

void FreeEntirePool(PoolId id) {
    if (purify_is_running()) {
     /* call 'free' on each block in this pool */
     purify_map_pool(id, free);
    } else {
     ... original code for FreeEntirePool ...
    }
}

void PrintPool(PoolId id) {
    if (purify_is_running()) {
     /* PrintBlock operates on a single block */
     /* call PrintBlock on every block in this pool */
     purify_map_pool(id, PrintBlock);
    } else {
      ... original code for PrintPool ...
    }
}
```

Modifying sbrk allocators

Custom allocators are written to obtain their memory directly from the operating system using brk, sbrk, mmap, or some other method. Usually such allocators also supply a definition of malloc and free because many libc functions call malloc and the default malloc does not work when other functions are calling sbrk.

To use Purify with sbrk allocators, you need to write an alternate implementation of your allocator calling malloc instead of sbrk. For example, if you have an external function GetPages that calls sbrk, and your own malloc implementation on top of GetPages, you could use:

```
char* GetPages(int n) {
    if (purify_is_running()) {
     return malloc(n * PAGSIZ);
    } else {
     return sbrk(n * PAGSIZ); /* original code */
    }
}

/* for Purify, don't define malloc, free: use default from libc
*/
#ifndef PURIFY    /* original case */
char* malloc(int n) {
    ... original code for malloc on top of GetPages ...
}
void free(char* x) {
    ...
}
#endif /* PURIFY */
```

Accessing auxiliary data

Special purpose custom memory allocation facilities can store additional data with the allocated memory. This data might include allocation time statistics or pointers to clean up functions to be called when the memory is freed. Often this is accomplished by allocating a few extra bytes of memory at the start of each block to store these values, and returning a pointer to the memory past these values.

Purify allows you to associate user data with every allocated block of memory by using the functions purify_set_user_data and purify_get_user_data.

Auxiliary data example

Extending the pool allocator example described on page 9-5, assume that the function AllocateFromPool stores the pool-id and the block size with each block allocated. Without Purify, you might use the two words immediately preceding the block returned to the user for these values.

```
char* AllocateFromPool(int pool_id, int size) {
    char *mem = ... get size+8 bytes ..
    /* Store the pool_id and size. */
     *((int*)mem + 0) = pool_id;
     *((int*)mem + 1) = size;
    /* Return the rest of the chunk to the user. */
    return (char*)((int*)mem + 2);
}

int BlockSize(char* mem) {
    return *((int*)mem - 1);

}
```

With Purify, you can implement this by using:

```
#ifdef PURIFY
char* AllocateFromPool(int pool_id, int size) {
    char *mem = malloc(size);
    purify_set_pool_id(mem, pool_id);
    purify_set_user_data(mem, size);
    return mem;
}

int BlockSize(char* mem) {
    return purify_get_user_data(mem);
}
#endif /* PURIFY */
```

You can also use the user data field to associate a string name to the memory for debugging purposes:

```
char* Malloc(int size, char* debug_name)
    char* mem = malloc(size);
    purify_set_pool_id(mem, 0);
    purify_set_user_data(mem, debug_name);
    return mem;
}

void PrintBlock(char* mem) {
    char* debug_name = purify_get_user_data(mem);
    printf("%s\n", debug_name);
    purify_describe(mem);
}
```

Note: Auxiliary data can only be associated with memory in a pool. If necessary, you can assign a dummy pool-id of 0 in order to establish a storage area for the auxiliary data.

See also, the `-pointer-offset` option in "Memory leak options" on page 12-16.

10

Purify Messages Reference

Message quick reference

For definitions of message severities, see page 10-2.

Message	Description	Severity	Page
ABR	Array Bounds Read	Warning	10-3
ABW	Array Bounds Write	Corrupting	10-4
BRK	Misuse of Brk or Sbrk	Corrupting	10-5
BSR	Beyond Stack Read	Warning	10-6
BSW	Beyond Stack Write	Warning	10-7
COR	Core Dump Imminent	Fatal	10-8
FIU	File Descriptors In Use	Informational	10-9
FMM	Freeing Mismatched Memory	Corrupting	10-10
FMR	Free Memory Read	Warning	10-11
FMW	Free Memory Write	Corrupting	10-12
FNH	Freeing Non Heap Memory	Corrupting	10-13
FUM	Freeing Unallocated Memory	Corrupting	10-14
IPR	Invalid Pointer Read	Fatal	10-15
IPW	Invalid Pointer Write	Fatal	10-16
MAF	Malloc Failure	Informational	10-17
MIU	Memory In-Use	Informational	10-18
MLK	Memory Leak	Warning	10-19
MRE	Malloc Reentrancy Error	Corrupting	10-20
MSE	Memory Segment Error	Warning	10-21
NPR	Null Pointer Read	Fatal	10-22
NPW	Null Pointer Write	Fatal	10-23

Message	Description	Severity	Page
PAR	Bad Parameter	Warning	10-24
PLK	Potential Memory Leak	Warning	10-25
SBR	Stack Array Bounds Read	Warning	10-26
SBW	Stack Array Bounds Write	Corrupting	10-27
SIG	Signal	Informational	10-28
SOF	Stack Overflow	Warning	10-29
UMC	Uninitialized Memory Copy	Warning	10-30
UMR	Uninitialized Memory Read	Warning	10-31
WPF	Watchpoint Free	Informational	10-32
WPM	Watchpoint Malloc	Informational	10-33
WPN	Watchpoint Entry	Informational	10-34
WPR	Watchpoint Read	Informational	10-35
WPW	Watchpoint Write	Informational	10-36
WPX	Watchpoint Exit	Informational	10-37
ZPR	Zero Page Read	Fatal	10-38
ZPW	Zero Page Write	Fatal	10-39

Message severity

Purify identifies each message by a three-character acronym, such as ABR for Array Bounds Read error, and classifies it according to four levels of severity:

- Fatal: This message indicates imminent abnormal program termination, unless you have installed error condition handlers.
- Corrupting: These messages indicate a major program malfunction.
- Warning: These messages indicate anomalous program behavior. Programs with these errors fail sporadically and often mysteriously.
- Informational: These messages are simply informational, providing additional debugging data.

Message descriptions

ABR   Array Bounds Read

An ABR message indicates that your program is about to read a
value from before or after a block or static data item. ABR is a
warning message.

```
┌─────────────────────────────────────────────────────────────────────┐
│ ─                          Purify: a.out                        · □  │
├─────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                          Help   │
├─────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (  1 error,    0 leaked bytes)        │
│    ▶ Purify instrumented a.out  (pid 10800 at Wed Mar 30 15:40:16 1994)│
│ ▼ ABR: Array bounds read                                            │
│    This is occurring while in:                                      │
│       ▼ main          [test.c:9]                                   │
│         ▨ /* Sample ABR (Array Bounds Read) code */                │
│                                                                     │
│            #define ELEMS   10                                       │
│            main() {                                                 │
│                char *ptr = (char *)calloc(ELEMS, sizeof(char));     │
│                int num;                                             │
│                char sum = '\0';                                     │
│                for (num = 0; num <= ELEMS; num++)                   │
│         ⇨          sum += ptr[num];/* ABR because "ptr[10]" not in bounds */│
│                free(ptr);                                           │
│                exit(0);                                             │
│            }                                                        │
│         start          [crt0.o]                                    │
│    Reading 1 byte from 0x6d38a in the heap.                        │
│    Address 0x6d38a is 1 byte past end of a malloc'd block at 0x6d380 of 10 by│
│    This block was allocated from:                                  │
│         malloc         [crt0.o]                                    │
│         calloc         [crt0.o]                                    │
│       ▶ main          [test.c:5]                                   │
│         start          [crt0.o]                                    │
│ ▶ Current file descriptors in use: 5                               │
│ ▶ Memory leaked: 0 bytes (100%); potentially leaked: 0 bytes (100%)│
│ ◀                                                              ▶    │
└─────────────────────────────────────────────────────────────────────┘
```

An ABR message can be caused by any of the following:

- Making an array too small, for example failing to account for the
  terminating NULL in a string

- Being off by one in copying elements up or down an array

- Forgetting to multiply by `sizeof(type)` when allocating for an
  array of objects

- Using an array index that is too large or negative

- Failing to NULL terminate a string

ABW   Array Bounds Write

An ABW message indicates that your program is about to write a
value before or after a block or static data item. ABW indicates a
corrupting error.

```
┌─────────────────────────────────────────────────────────────────────┐
│ ─  │                         Purify: a.out                      │ □ │ □ │
├─────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                            Help   │
├─────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (   1 error,    0 leaked bytes)        │
│     ▶ Purify instrumented a.out (pid 8383 at Wed Jul 10 17:55:32 1996)│
│   ▼ ABW: Array bounds write                                           │
│     This is occurring while in:                                       │
│       ▼ main          [abw.c:9]                                       │
│         🖉 int array[10];                                             │
│                                                                       │
│            main() {                                                   │
│                int i;                                                 │
│                                                                       │
│                for ( i = 0; i <= 10; i++ )                            │
│         ➡           array[i] = i;        /* ABW because 10 is beyond array */ │
│            }                                                          │
│         start        [crt0.o]                                         │
│       Writing 4 bytes to 0x421b8 in the zero'd data, bss section.     │
│       Address 0x421b8 is 40 bytes past start of global variable "array".│
│       This is defined in data8225common_defs.o.                       │
│     ▶ Current file descriptors in use: 5                              │
│       No dynamic memory is in use.  No leaks are possible.            │
│     ▶ Program exited with status code 1.                              │
└─────────────────────────────────────────────────────────────────────┘
```
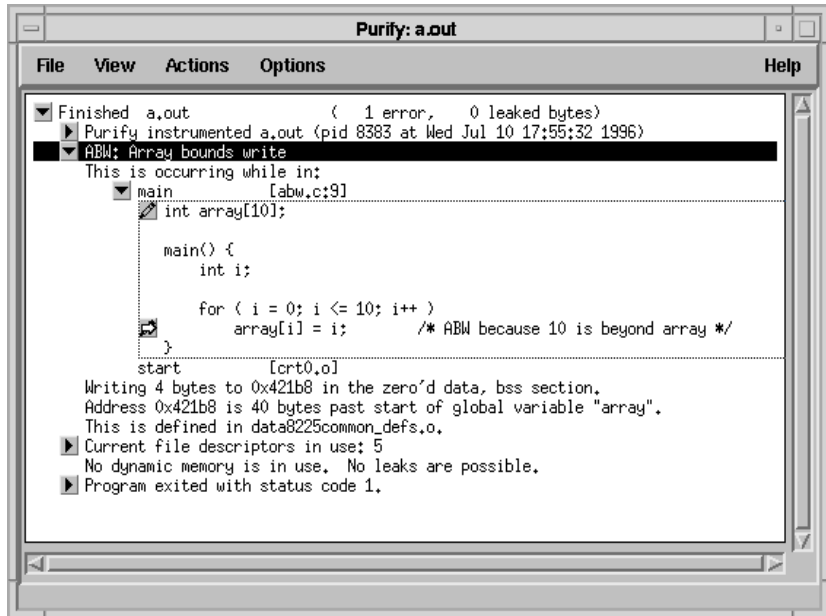
An ABW message can be caused by any of the following:

- Making an array too small, for example failing to account for the
  terminating NULL in a string

- Being off by one in copying elements up or down an array

- Forgetting to multiply by sizeof(type) when allocating for an
  array of objects

- Using an array index that is too large or negative

- Failing to NULL terminate a string

BRK   Misuse of Brk or Sbrk

A BRK message indicates that your program is using brk or sbrk
directly to allocate memory. Use of brk or sbrk is incompatible with
the use of most implementations of malloc and free. BRK indicates
a *corrupting* error.
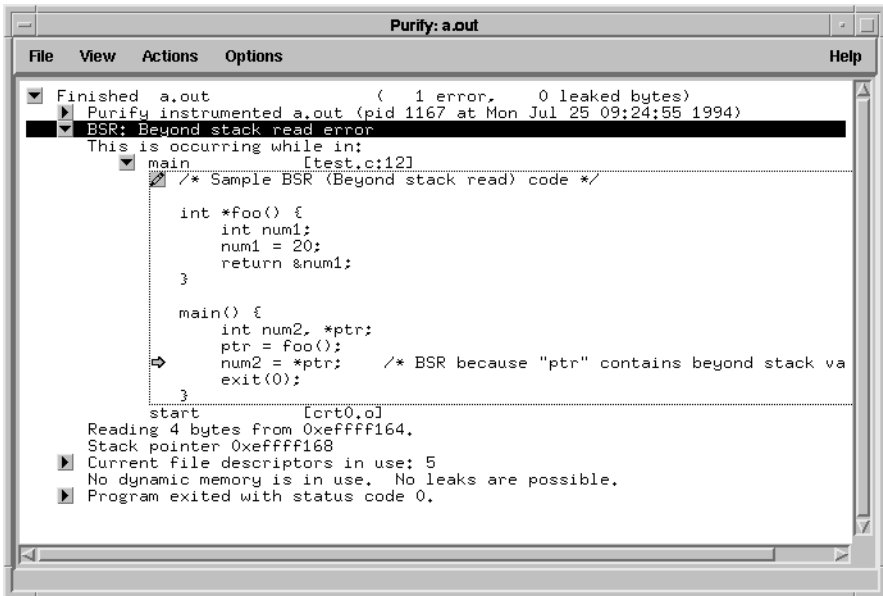
```
┌────────────────────────────────────────────────────────────────────┐
│ ─                          Purify: a.out                      □   □ │
├────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                           Help  │
├────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out               (   1 error,    0 leaked bytes)     │
│    ▶ Purify instrumented a.out (pid 841 at Wed Jul 17 18:50:35 1996)│
│    ▼ BRK: Brk or sbrk called directly                              │
│      This is occurring while in:                                   │
│           sbrk          [rtlib.o]                                   │
│        ▼ main           [brk.c:5]                                   │
│           ▨ /* Sample BRK (Misuses of Brk or Sbrk) code */          │
│                                                                     │
│             main() {                                                │
│                  char *ptr = (char *)malloc(10);                    │
│           ⇨      sbrk(100000);    /* BRK because sbrk combined with malloc */ │
│                  free(ptr);                                         │
│                  exit(0);                                           │
│             }                                                       │
│           start          [crt0.o]                                   │
│      You are calling sbrk(100000) directly, rather than from malloc(). │
│      Note: If you are using your own memory manager to call sbrk()  │
│      rather than malloc, please refer to the Custom Memory Managers │
│      chapter in the Purify User's Guide.                           │
│    ▶ Current file descriptors in use: 5                             │
│    ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  │
│    ▶ Program exited with status code 0.                             │
│                                                                     │
│                                                                     │
└────────────────────────────────────────────────────────────────────┘
```

BSR

Beyond Stack Read

A BSR message indicates that a function in your program is about to read beyond the stack pointer. BSR is a *warning* message.

```
+----------------------------------------------------------------------------+
| -                        Purify: a.out                              * []   |
+----------------------------------------------------------------------------+
| File   View   Actions   Options                                    Help    |
+----------------------------------------------------------------------------+
| ▼ Finished  a.out                    (  1 error,    0 leaked bytes)        |
|   ▶ Purify instrumented a.out (pid 1167 at Mon Jul 25 09:24:55 1994)       |
|   ▼ BSR: Beyond stack read error                                           |
|     This is occurring while in:                                            |
|       ▼ main          [test.c:12]                                          |
|           ▨ /* Sample BSR (Beyond stack read) code */                      |
|                                                                            |
|              int *foo() {                                                  |
|                   int num1;                                                |
|                   num1 = 20;                                               |
|                   return &num1;                                            |
|              }                                                             |
|                                                                            |
|              main() {                                                      |
|                   int num2, *ptr;                                          |
|                   ptr = foo();                                             |
|              ⇨   num2 = *ptr;    /* BSR because "ptr" contains beyond stack va|
|                   exit(0);                                                 |
|              }                                                             |
|         start        [crt0.o]                                              |
|     Reading 4 bytes from 0xeffff164.                                       |
|     Stack pointer 0xeffff168                                               |
|   ▶ Current file descriptors in use: 5                                     |
|     No dynamic memory is in use.  No leaks are possible.                   |
|   ▶ Program exited with status code 0.                                     |
+----------------------------------------------------------------------------+
```
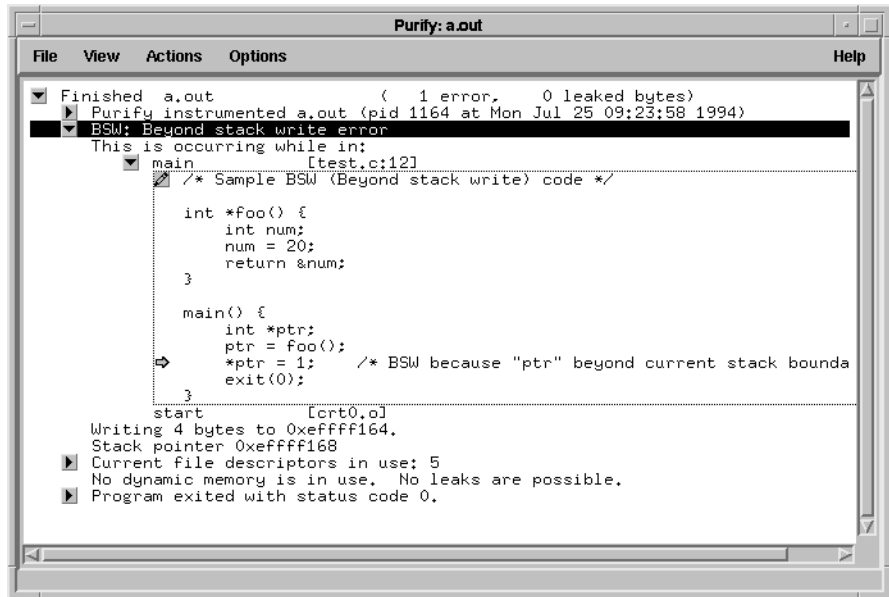
A BSR message is commonly caused by a function returning a pointer to a local variable that has gone out of scope. If the caller attempts to use that variable, it can result in a BSR message. To keep the value valid after the called function returns, make such variables `static`.

BSW   Beyond Stack Write

A BSW message indicates that a function in your program is about to write beyond the stack pointer. BSW is a *warning* message.

```
┌─────────────────────────────────────────────────────────────────────────┐
│ —                            Purify: a.out                          ⌐ □  │
├─────────────────────────────────────────────────────────────────────────┤
│ File   View   Actions   Options                                     Help │
├─────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out                    (  1 error,   0 leaked bytes)       │
│   ▶ Purify instrumented a.out (pid 1164 at Mon Jul 25 09:23:58 1994)     │
│   ▼ BSW: Beyond stack write error                                        │
│     This is occurring while in:                                          │
│       ▼ main           [test.c:12]                                       │
│         ▨ /* Sample BSW (Beyond stack write) code */                     │
│                                                                           │
│              int *foo() {                                                 │
│                   int num;                                                │
│                   num = 20;                                               │
│                   return &num;                                            │
│              }                                                            │
│                                                                           │
│              main() {                                                     │
│                   int *ptr;                                               │
│                   ptr = foo();                                            │
│              ⇨    *ptr = 1;    /* BSW because "ptr" beyond current stack bounda│
│                   exit(0);                                                │
│              }                                                            │
│          start          [crt0.o]                                         │
│     Writing 4 bytes to 0xeffff164.                                       │
│     Stack pointer 0xeffff168                                             │
│   ▶ Current file descriptors in use: 5                                   │
│     No dynamic memory is in use.  No leaks are possible.                 │
│   ▶ Program exited with status code 0.                                   │
└─────────────────────────────────────────────────────────────────────────┘
```

A BSW message is commonly caused by a function returning a pointer to a local variable that has gone out of scope. If the caller attempts to use that variable, it can result in a BSW error. To keep the value valid after the called function returns, make such variables `static`.

Note: Unlike other write errors, such as an ABW, this is not a corrupting error, since it is always legal to write a value beyond the end of the stack. However, values beyond the current stack pointer are subject to change without notice. For example, if your program takes a context switch or a signal, the value written by this access might not be reliably re-read.

A COR message indicates that your program has received a signal that is normally terminal. COR indicates a *fatal* error.

```
┌─────────────────────────────────────────────────────────────────────┐
│ ─                         Purify: a.out                        ▫  □  │
├─────────────────────────────────────────────────────────────────────┤
│  File   View   Actions    Options                              Help  │
├─────────────────────────────────────────────────────────────────────┤
│ ▶ Purify instrumented a.out (pid 3604 at Wed Jun 26 20:57:35 1996)   │
│ ▶ IPR: Invalid pointer read                                          │
│ ▼ COR: Fatal core dump                                               │
│    This is occurring while in:                                       │
│    ▼ main          [test.c:12]                                       │
│      ▨ /* Sample COR (Core Dump Imminent) code */                    │
│                                                                       │
│          #include <signal.h>                                         │
│                                                                       │
│          void handler() {                                            │
│              exit(0);                                                 │
│          }                                                           │
│          main()                                                      │
│          {                                                           │
│              int num;                                                │
│              signal(SIGSEGV, handler);                               │
│        ⇨     num = *(int *)(sbrk(0) + 10000);   /* COR because of bad pointer */ │
│              exit(0);                                                 │
│          }                                                           │
│      start      [crt0.o]                                             │
│    Received signal 11 (SIGSEGV - Segmentation fault)                 │
│    Signal bit = 0x00000400                                           │
│    Handler function:                                                 │
│       ▶ handler        [test.c:5]                                    │
│    Faulting address = 0x448b0                                        │
│    Signal mask: (SIGSEGV)                                            │
│    Pending signals:                                                  │
│ ▶ Current file descriptors in use: 5                                 │
│    No dynamic memory is in use.  No leaks are possible.              │
│ ▶ Program exited with status code 0.                                 │
└─────────────────────────────────────────────────────────────────────┘
```
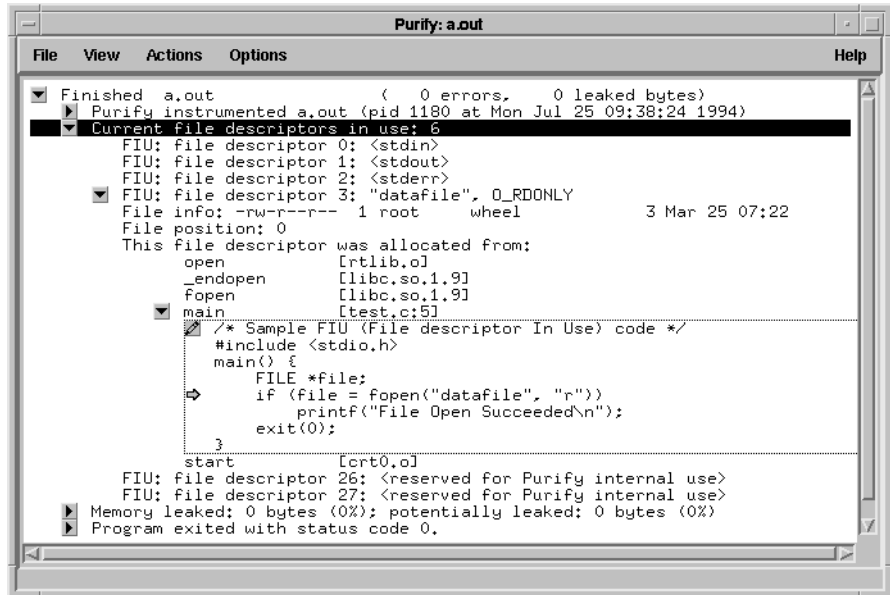
This is an example of a core dump due to a bad pointer—either bad pointer arithmetic or pointer corruption (indicated by the preceding IPR).

To turn off COR messages for signals that you handle within your program, use the `-handle-signals` and `-ignore-signals` options. For details, see "Miscellaneous options" on page 12-31.

FIU   File Descriptors in Use

An FIU message describes file descriptors that are in use by your
program. FIU is an *informational* message.

```
┌─────────────────────────────────────────────────────────────────────────┐
│ ─                            Purify: a.out                            ◆ □ │
├─────────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                                 Help  │
├─────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out                 (   0 errors,    0 leaked bytes)        │
│   ▶ Purify instrumented a.out (pid 1180 at Mon Jul 25 09:38:24 1994)      │
│   ▼ Current file descriptors in use: 6                                    │
│       FIU: file descriptor 0: <stdin>                                     │
│       FIU: file descriptor 1: <stdout>                                    │
│       FIU: file descriptor 2: <stderr>                                    │
│   ▼ FIU: file descriptor 3: "datafile", O_RDONLY                         │
│       File info: -rw-r--r--  1 root      wheel           3 Mar 25 07:22   │
│       File position: 0                                                    │
│       This file descriptor was allocated from:                           │
│             open          [rtlib.o]                                       │
│             _endopen      [libc.so.1.9]                                   │
│             fopen         [libc.so.1.9]                                   │
│   ▼     main          [test.c:5]                                          │
│         ▨  /* Sample FIU (File descriptor In Use) code */                 │
│              #include <stdio.h>                                           │
│            main() {                                                       │
│                 FILE *file;                                               │
│         ⇨       if (file = fopen("datafile", "r"))                        │
│                     printf("File Open Succeeded\n");                      │
│                 exit(0);                                                  │
│              }                                                            │
│             start         [crt0.o]                                        │
│       FIU: file descriptor 26: <reserved for Purify internal use>         │
│       FIU: file descriptor 27: <reserved for Purify internal use>         │
│ ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)           │
│ ▶ Program exited with status code 0.                                      │
│ ◁                                                                      ▷   │
└─────────────────────────────────────────────────────────────────────────┘
```
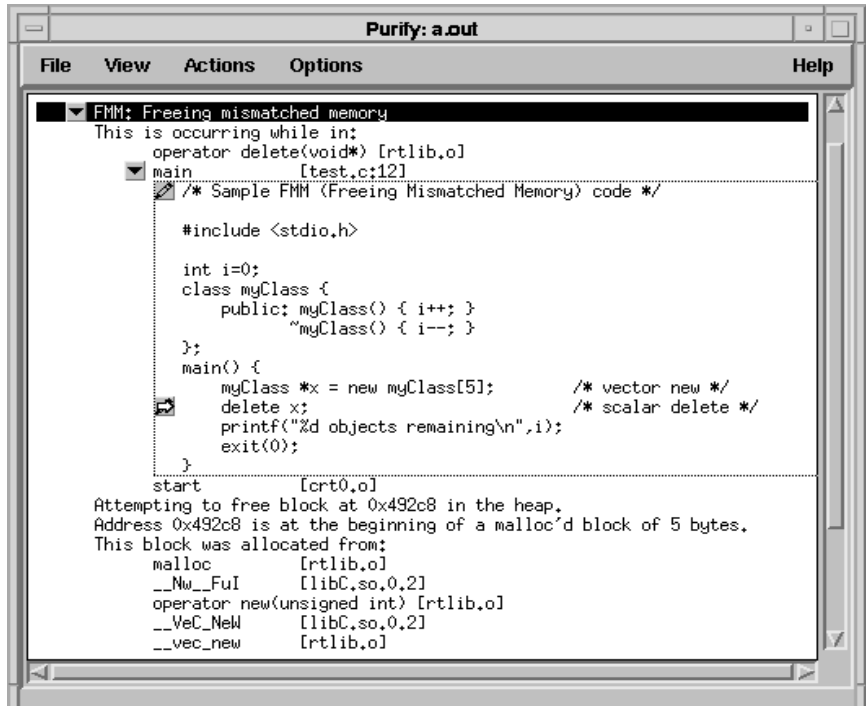
To generate a list of file descriptors in use, set the option
-purify-fds-inuse-at-exit=yes (the default), or the API function
purify_new_fds_inuse, or purify_all_fds_inuse.

Each FIU message describes what is known about the origin of one
open file descriptor. If you see multiple descriptors for the same
file, or from the same call chain in the program, you should be
concerned that you have a file descriptor leak, and your program
might run out of file descriptors.

FMM   Freeing Mismatched Memory

An FMM message indicates that your program is deallocating
memory using a function from a different family than the one used
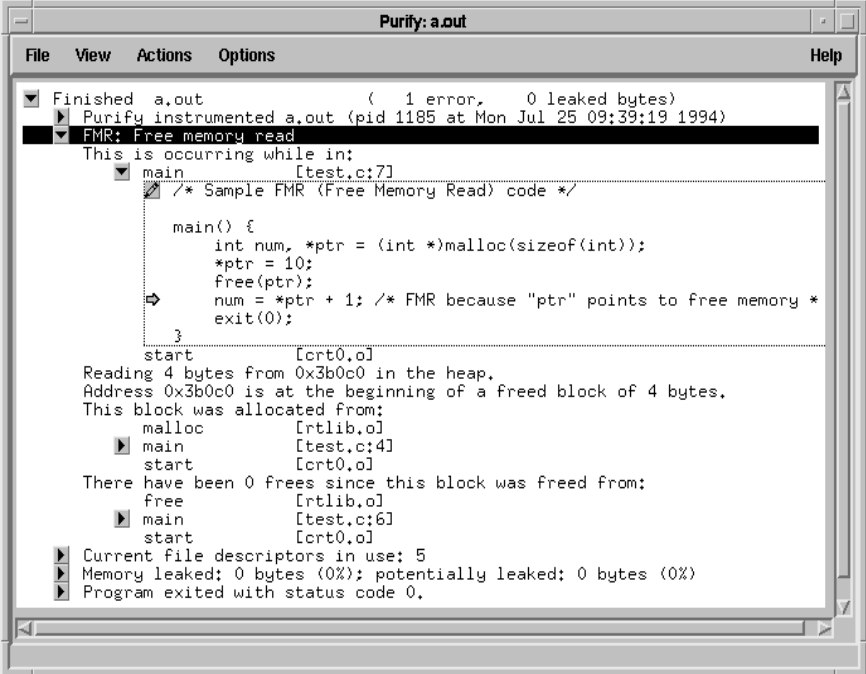to allocate the memory. FMM indicates a *corrupting* error.

An FMM error can occur when you use new[] to allocate memory
and delete to free the memory. You should use delete[] instead,
otherwise the destructor associated with the memory cannot be
run. Purify reports an FMM message when your program allocates
memory from one family of APIs and then deallocates the memory
from a mismatching family. Purify checks these families:

```
new/delete
new[]/delete[]
malloc/free
calloc/free
realloc/free
XtMalloc/XtFree
```

FMR   Free Memory Read

An FMR message indicates that your program is about to read
from heap memory that has already been freed. FMR is a *warning*
message.
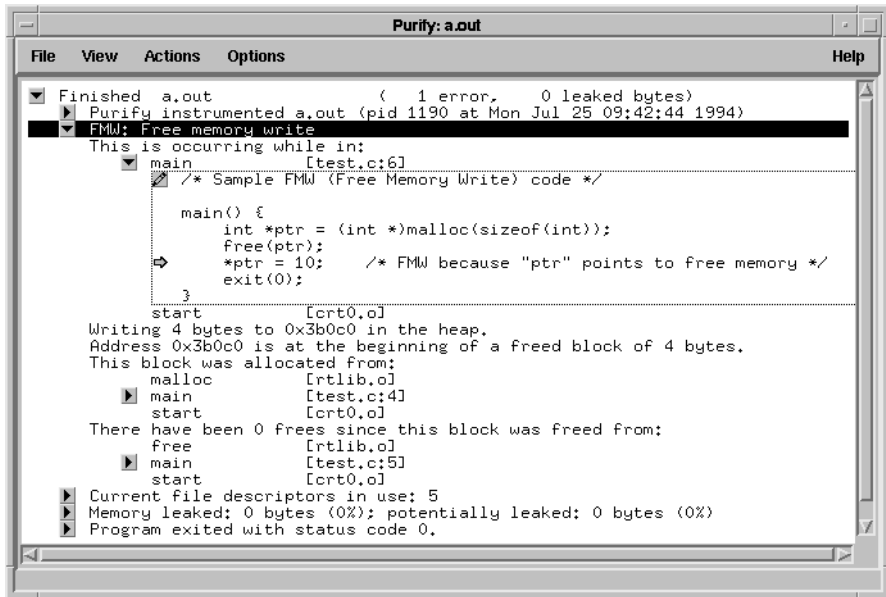
```
┌─────────────────────────────────────────────────────────────────────────┐
│ ─                        Purify: a.out                              ▲ │□│ │
├─────────────────────────────────────────────────────────────────────────┤
│ File   View   Actions   Options                                    Help   │
├─────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (   1 error,    0 leaked bytes)          ▲ │
│    ▶ Purify instrumented a.out (pid 1185 at Mon Jul 25 09:39:19 1994)     │
│   ▼ FMR: Free memory read                                                 │
│      This is occurring while in:                                          │
│        ▼ main           [test.c:7]                                        │
│         ▨ /* Sample FMR (Free Memory Read) code */                        │
│                                                                           │
│            main() {                                                       │
│                 int num, *ptr = (int *)malloc(sizeof(int));               │
│                 *ptr = 10;                                                │
│                 free(ptr);                                                │
│           ⇨     num = *ptr + 1; /* FMR because "ptr" points to free memory *│
│                 exit(0);                                                  │
│            }                                                              │
│           start         [crt0.o]                                          │
│      Reading 4 bytes from 0x3b0c0 in the heap.                            │
│      Address 0x3b0c0 is at the beginning of a freed block of 4 bytes.     │
│      This block was allocated from:                                       │
│           malloc        [rtlib.o]                                         │
│         ▶ main          [test.c:4]                                        │
│           start         [crt0.o]                                          │
│      There have been 0 frees since this block was freed from:             │
│           free          [rtlib.o]                                         │
│         ▶ main          [test.c:6]                                        │
│           start         [crt0.o]                                          │
│    ▶ Current file descriptors in use: 5                                   │
│    ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)        │
│    ▶ Program exited with status code 0.                                 ▼ │
│ ◁                                                                       ▷ │
└─────────────────────────────────────────────────────────────────────────┘
```
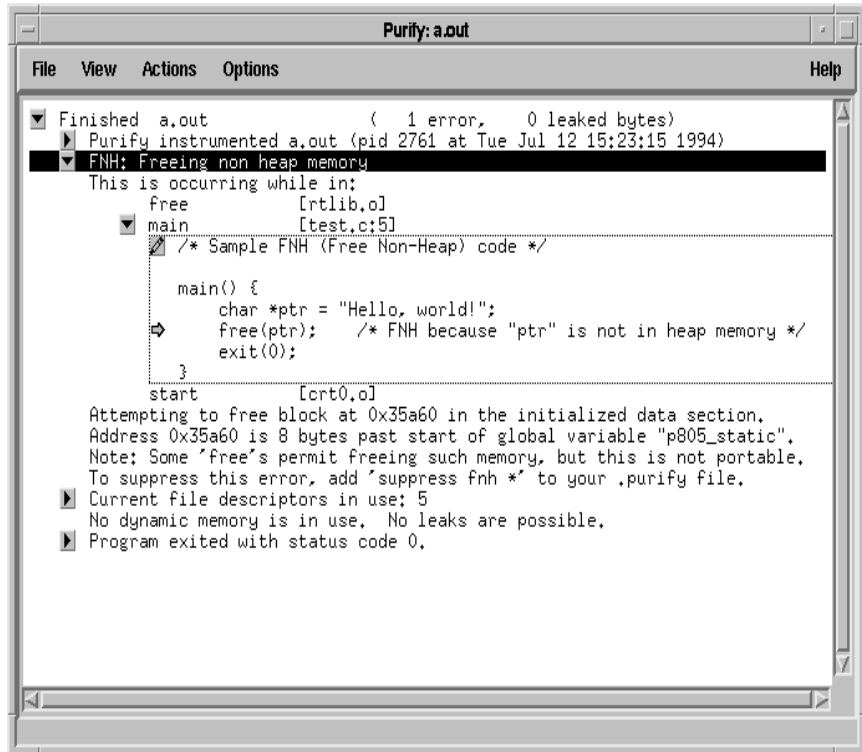
An FMR message can be caused by reading via a dangling pointer
to a block of memory that has already been freed. It could also be
the result of indexing far off the end of a valid block, or using a
completely random pointer that happens to fall within the heap
segment.

FMW   Free Memory Write

An FMW message indicates that your program is about to write to heap memory that has already been freed. FMW indicates a *corrupting* error.

```
┌─────────────────────────────────────────────────────────────────────────┐
│  —                              Purify: a.out                        · □ │
├─────────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                              Help    │
├─────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out                    (   1 error,   0 leaked bytes)       │
│    ▶ Purify instrumented a.out (pid 1190 at Mon Jul 25 09:42:44 1994)     │
│ ▼  FMW: Free memory write                                                 │
│     This is occurring while in:                                           │
│        ▼ main          [test.c:6]                                         │
│          ▨  /* Sample FMW (Free Memory Write) code */                     │
│                                                                           │
│             main() {                                                      │
│                   int *ptr = (int *)malloc(sizeof(int));                  │
│                   free(ptr);                                              │
│             ⇨     *ptr = 10;    /* FMW because "ptr" points to free memory */ │
│                   exit(0);                                                │
│             }                                                             │
│          start          [crt0.o]                                          │
│     Writing 4 bytes to 0x3b0c0 in the heap.                               │
│     Address 0x3b0c0 is at the beginning of a freed block of 4 bytes.      │
│     This block was allocated from:                                        │
│          malloc         [rtlib.o]                                         │
│        ▶ main           [test.c:4]                                        │
│          start          [crt0.o]                                          │
│     There have been 0 frees since this block was freed from:              │
│          free           [rtlib.o]                                         │
│        ▶ main           [test.c:5]                                        │
│          start          [crt0.o]                                          │
│ ▶ Current file descriptors in use: 5                                      │
│ ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)           │
│ ▶ Program exited with status code 0.                                      │
│  ◁                                                                     ▷   │
└─────────────────────────────────────────────────────────────────────────┘
```

An FMW message can be caused by writing via a dangling pointer to a block of memory that has already been freed. It could also be the result of indexing far off the end of a valid block, or using a completely random pointer that happens to fall within the heap segment.

FNH   Freeing Non Heap Memory

An FNH message indicates that your program is calling `free` with
a memory address that is not in the heap (memory in stack, data
or bss). FNH indicates a *corrupting* error.
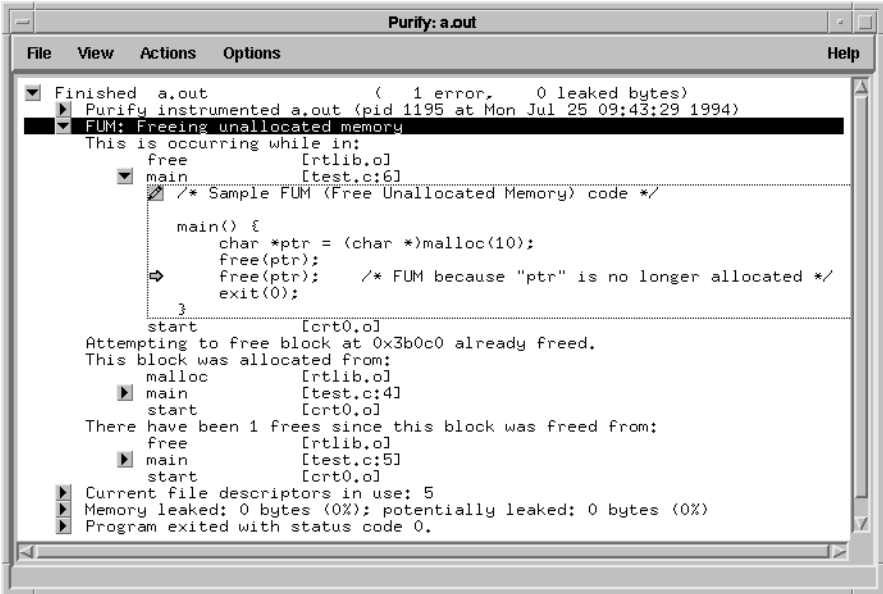
```
┌─────────────────────────────────────────────────────────────────────────┐
│ ─                          Purify: a.out                           ◢  □  │
├─────────────────────────────────────────────────────────────────────────┤
│ File   View   Actions   Options                                    Help  │
├─────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (   1 error,    0 leaked bytes)           │
│   ▶ Purify instrumented a.out (pid 2761 at Tue Jul 12 15:23:15 1994)     │
│   ▼ FNH: Freeing non heap memory                                         │
│     This is occurring while in:                                          │
│           free           [rtlib.o]                                       │
│       ▼ main           [test.c:5]                                        │
│         ╱ /* Sample FNH (Free Non-Heap) code */                          │
│                                                                          │
│           main() {                                                       │
│               char *ptr = "Hello, world!";                               │
│         ⇨     free(ptr);   /* FNH because "ptr" is not in heap memory */ │
│               exit(0);                                                    │
│           }                                                              │
│         start          [crt0.o]                                          │
│     Attempting to free block at 0x35a60 in the initialized data section. │
│     Address 0x35a60 is 8 bytes past start of global variable "p805_static".│
│     Note: Some 'free's permit freeing such memory, but this is not portable.│
│     To suppress this error, add 'suppress fnh *' to your .purify file.    │
│   ▶ Current file descriptors in use: 5                                   │
│     No dynamic memory is in use.  No leaks are possible.                  │
│   ▶ Program exited with status code 0.                                   │
└─────────────────────────────────────────────────────────────────────────┘
```

An FNH error often occurs due to confusion about pointer
ownership. Look for pointers to strings or objects that are
normally allocated on the heap being initialized with pointers to
constants in the program data or text segments, or on the stack.
This FNH error is caused by attempts to free such addresses.

FUM   Freeing Unallocated Memory

An FUM message indicates that your program is trying to `free` unallocated memory (duplicate free or free of bad heap pointer). FUM indicates a *corrupting* error.

```
┌─────────────────────────────────────────────────────────────────────────┐
│ −                              Purify: a.out                          · □ │
├─────────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                                 Help │
├─────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (   1 error,    0 leaked bytes)         ▲ │
│     ▶ Purify instrumented a.out (pid 1195 at Mon Jul 25 09:43:29 1994)    │
│     ▼ FUM: Freeing unallocated memory                                    │
│        This is occurring while in:                                        │
│             free          [rtlib.o]                                       │
│        ▼ main            [test.c:6]                                       │
│          ▨ /* Sample FUM (Free Unallocated Memory) code */               │
│                                                                           │
│             main() {                                                      │
│                  char *ptr = (char *)malloc(10);                          │
│                  free(ptr);                                               │
│             ⇨    free(ptr);    /* FUM because "ptr" is no longer allocated */ │
│                  exit(0);                                                 │
│             }                                                             │
│             start         [crt0.o]                                        │
│        Attempting to free block at 0x3b0c0 already freed.                 │
│        This block was allocated from:                                     │
│             malloc        [rtlib.o]                                       │
│          ▶ main          [test.c:4]                                      │
│             start         [crt0.o]                                        │
│        There have been 1 frees since this block was freed from:           │
│             free          [rtlib.o]                                       │
│          ▶ main          [test.c:5]                                      │
│             start         [crt0.o]                                        │
│   ▶ Current file descriptors in use: 5                                   │
│   ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)        │
│   ▶ Program exited with status code 0.                                ▽ │
│ ◁                                                                     ▷  │
└─────────────────────────────────────────────────────────────────────────┘
```

An FUM error often occurs due to confusion about pointer ownership. Only the owner should free heap objects.

If there are many references to a heap object with no one reference being clearly the longest lived, the object referenced might have a reference count. Failure to maintain the reference count properly can also lead to this error.
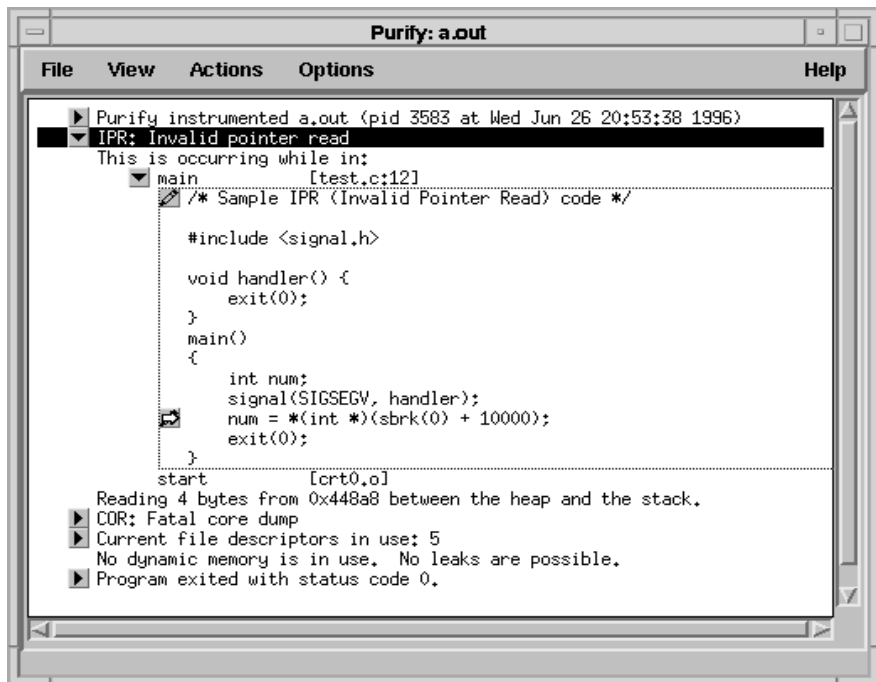
IPR   Invalid Pointer Read

An IPR message indicates that your program is about to read from an address that is outside any valid segment of your program. Valid segments include program text, data, heap, stack, `mmap`'d regions, and shared memory. This usually results in a segmentation violation.

IPR messages are similar to NPR and ZPR messages, except that they indicate an invalid reference to memory outside of the zeroth page. IPR indicates a *fatal* error.

```
┌──────────────────────────────────────────────────────────────┐
│ □                        Purify: a.out                    □  □ │
├──────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                     Help  │
├──────────────────────────────────────────────────────────────┤
│  ▶ Purify instrumented a.out (pid 3583 at Wed Jun 26 20:53:38 1996)│
│  ▼ IPR: Invalid pointer read                                  │
│    This is occurring while in:                                │
│      ▼ main          [test.c:12]                              │
│        ▨ /* Sample IPR (Invalid Pointer Read) code */         │
│                                                               │
│           #include <signal.h>                                 │
│                                                               │
│           void handler() {                                    │
│               exit(0);                                        │
│           }                                                   │
│           main()                                              │
│           {                                                   │
│               int num;                                        │
│               signal(SIGSEGV, handler);                       │
│        ▱     num = *(int *)(sbrk(0) + 10000);                 │
│               exit(0);                                        │
│           }                                                   │
│        start          [crt0.o]                                │
│      Reading 4 bytes from 0x448a8 between the heap and the stack.│
│  ▶ COR: Fatal core dump                                       │
│  ▶ Current file descriptors in use: 5                         │
│    No dynamic memory is in use.  No leaks are possible.       │
│  ▶ Program exited with status code 0.                         │
└──────────────────────────────────────────────────────────────┘
```
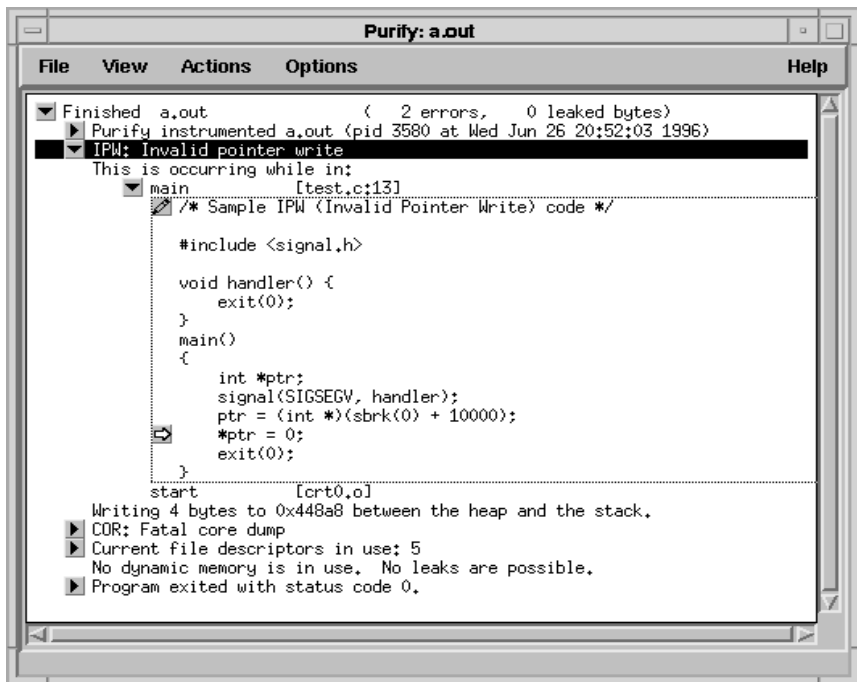
Note: Earlier versions of Purify reported this type of error as an MSE.

IPW   Invalid Pointer Write

An IPW message indicates that your program is trying to write to an address that is outside any valid segment of your program. Valid segments include program text, data, heap, stack, `mmap`'d regions, and shared memory. This usually results in a segmentation violation.

IPW messages are similar to NPW and ZPW messages, except that they indicate an invalid reference to memory outside of the zeroth page. IPW indicates a *fatal* error.
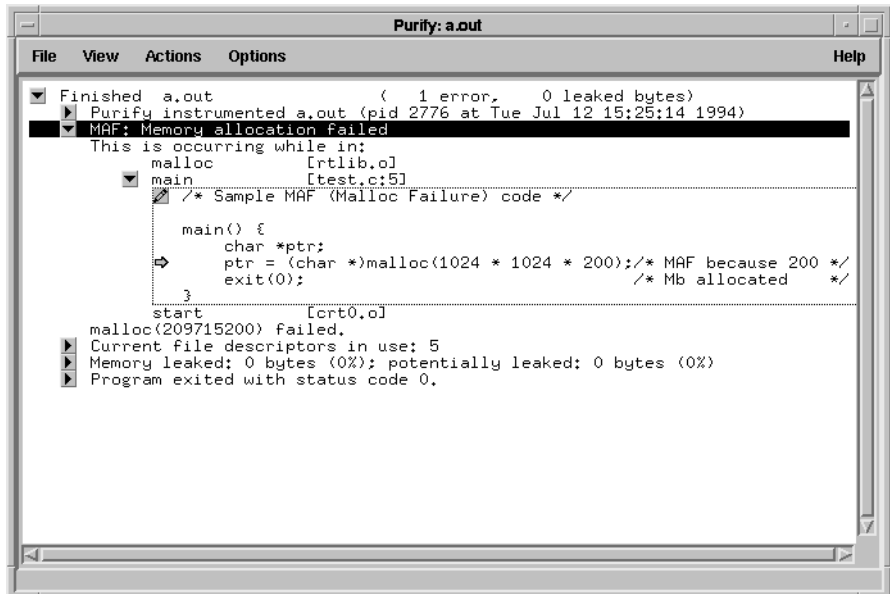
```
┌──────────────────────────────────────────────────────────────────┐
│ ─                       Purify: a.out                        □  □ │
├──────────────────────────────────────────────────────────────────┤
│  File   View   Actions   Options                           Help  │
├──────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out           (   2 errors,    0 leaked bytes)     │
│   ▶ Purify instrumented a.out (pid 3580 at Wed Jun 26 20:52:03 1996) │
│   ▼ IPW: Invalid pointer write                                   │
│     This is occurring while in:                                  │
│       ▼ main          [test.c:13]                                │
│          /* Sample IPW (Invalid Pointer Write) code */           │
│                                                                  │
│          #include <signal.h>                                     │
│                                                                  │
│          void handler() {                                        │
│              exit(0);                                            │
│          }                                                       │
│          main()                                                  │
│          {                                                       │
│              int *ptr;                                           │
│              signal(SIGSEGV, handler);                           │
│              ptr = (int *)(sbrk(0) + 10000);                     │
│          ➪  *ptr = 0;                                            │
│              exit(0);                                            │
│          }                                                       │
│        start          [crt0.o]                                   │
│     Writing 4 bytes to 0x448a8 between the heap and the stack.   │
│   ▶ COR: Fatal core dump                                         │
│   ▶ Current file descriptors in use: 5                           │
│     No dynamic memory is in use.  No leaks are possible.         │
│   ▶ Program exited with status code 0.                           │
└──────────────────────────────────────────────────────────────────┘
```
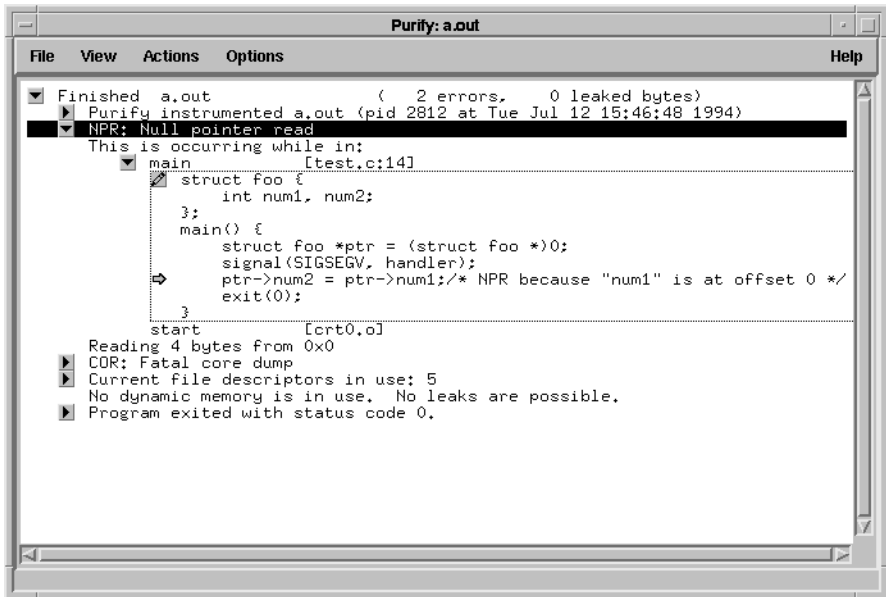
Note: Earlier versions of Purify reported this type of error as an MSE.

MAF   Malloc Failure

An MAF message indicates that `malloc` has failed—you have run out of swap space for the heap to grow. After the message is delivered, `malloc` returns `NULL` in the normal manner. MAF is an *informational* message about memory.
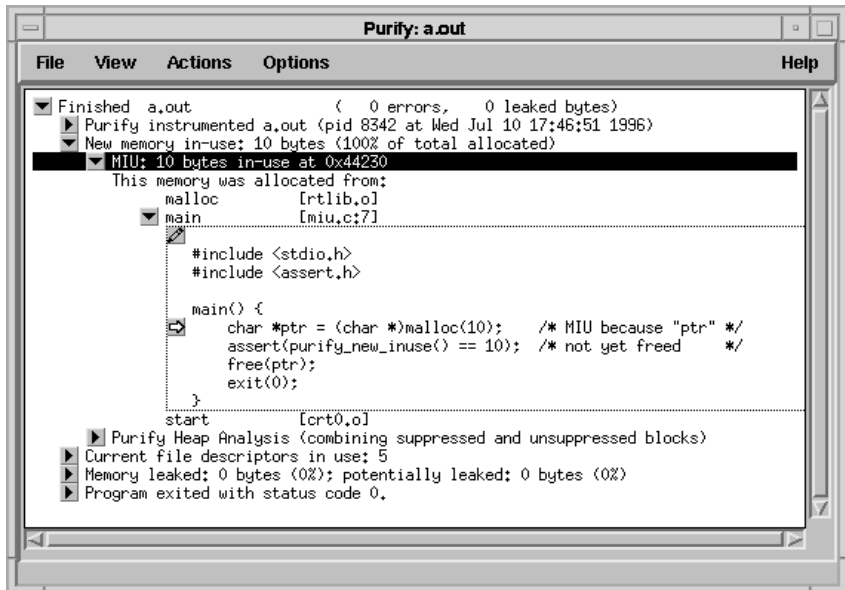
```
┌─────────────────────────────────────────────────────────────────────────────┐
│ ─                              Purify: a.out                            ▫ ▫   │
├─────────────────────────────────────────────────────────────────────────────┤
│ File    View    Actions    Options                                     Help   │
├─────────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (  1 error,   0 leaked bytes)                ▲ │
│   ▶ Purify instrumented a.out (pid 2776 at Tue Jul 12 15:25:14 1994)          │
│     ▼ MAF: Memory allocation failed                                           │
│       This is occurring while in:                                             │
│           malloc        [rtlib.o]                                             │
│       ▼ main            [test.c:5]                                            │
│         ▨ /* Sample MAF (Malloc Failure) code */                             │
│                                                                               │
│            main() {                                                           │
│                char *ptr;                                                      │
│         ⇨      ptr = (char *)malloc(1024 * 1024 * 200);/* MAF because 200 */  │
│                exit(0);                             /* Mb allocated     */    │
│            }                                                                  │
│           start         [crt0.o]                                              │
│       malloc(209715200) failed.                                               │
│   ▶ Current file descriptors in use: 5                                        │
│   ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)             │
│   ▶ Program exited with status code 0.                                        │
│                                                                            ▼  │
│ ◀                                                                          ▶  │
└─────────────────────────────────────────────────────────────────────────────┘
```

Ideally, programs should handle out-of-swap conditions gracefully, but often do not. If your program next generates an NPR, NPW, ZPR or ZPW, and then a COR, a caller of `malloc` has failed to check the return status and is dereferencing the `NULL` pointer.

An MIU message describes heap memory that you are currently using (memory to which there is a pointer). MIU is an *informational* message about memory.

```
┌──────────────────────────────────────────────────────────────────────────┐
│ ─                              Purify: a.out                         □   □ │
├──────────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                                  Help  │
├──────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (   0 errors,   0 leaked bytes)              │
│   ▶ Purify instrumented a.out (pid 8342 at Wed Jul 10 17:46:51 1996)       │
│   ▼ New memory in-use: 10 bytes (100% of total allocated)                  │
│     ▼ MIU: 10 bytes in-use at 0x44230                                      │
│        This memory was allocated from:                                     │
│          malloc        [rtlib.o]                                           │
│     ▼ main          [miu.c:7]                                              │
│        ▨                                                                   │
│            #include <stdio.h>                                              │
│            #include <assert.h>                                             │
│                                                                            │
│            main() {                                                        │
│        ⇨      char *ptr = (char *)malloc(10);   /* MIU because "ptr" */    │
│               assert(purify_new_inuse() == 10); /* not yet freed     */    │
│               free(ptr);                                                   │
│               exit(0);                                                     │
│            }                                                               │
│          start         [crt0.o]                                           │
│     ▶ Purify Heap Analysis (combining suppressed and unsuppressed blocks)  │
│   ▶ Current file descriptors in use: 5                                     │
│   ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)          │
│   ▶ Program exited with status code 0.                                     │
│                                                                            │
│ ◁                                                                        ▷ │
└──────────────────────────────────────────────────────────────────────────┘
```
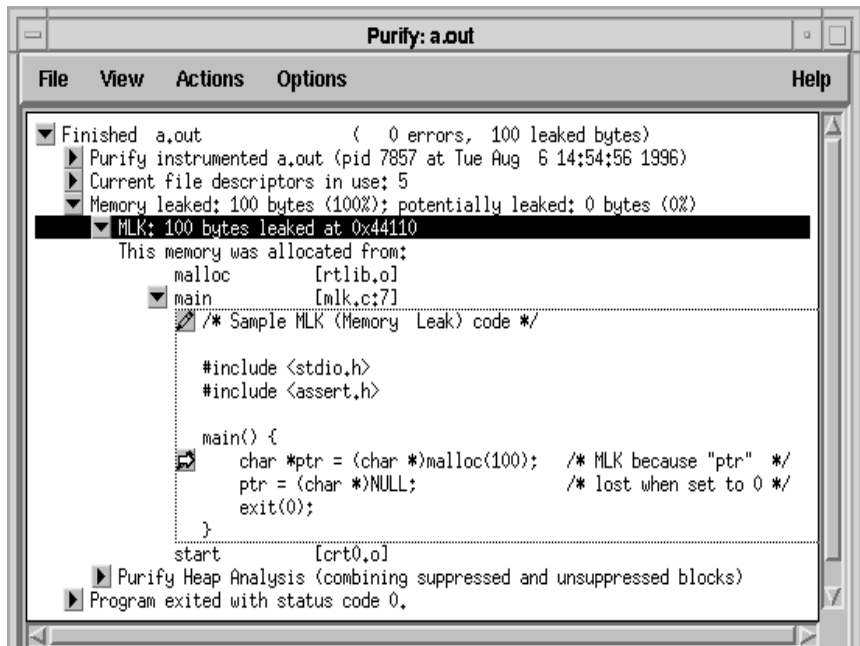
To generate a list of memory blocks in use, use the API function `purify_new_inuse` or `purify_all_inuse`, or set the option `-inuse-at-exit=yes`.

MLK   Memory Leak

An MLK message describes heap memory that you have leaked.
There are no pointers to this block, or to anywhere within this
block. MLK is a *warning* message.

```
┌─────────────────────────────────────────────────────────────────────┐
│ ─ │                       Purify: a.out                       │  □  □ │
├─────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                           Help   │
├─────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (   0 errors,  100 leaked bytes)       │
│    ▶ Purify instrumented a.out (pid 7857 at Tue Aug  6 14:54:56 1996) │
│    ▶ Current file descriptors in use: 5                              │
│    ▼ Memory leaked: 100 bytes (100%); potentially leaked: 0 bytes (0%)│
│       ▼ MLK: 100 bytes leaked at 0x44110                             │
│          This memory was allocated from:                             │
│             malloc          [rtlib.o]                                │
│          ▼ main             [mlk.c:7]                                │
│             ▨ /* Sample MLK (Memory  Leak) code */                   │
│                                                                      │
│               #include <stdio.h>                                     │
│               #include <assert.h>                                    │
│                                                                      │
│               main() {                                               │
│             ▱    char *ptr = (char *)malloc(100);   /* MLK because "ptr"  */│
│                  ptr = (char *)NULL;                /* lost when set to 0 */│
│                  exit(0);                                            │
│               }                                                     │
│             start           [crt0.o]                                │
│       ▶ Purify Heap Analysis (combining suppressed and unsuppressed blocks)│
│    ▶ Program exited with status code 0.                             │
└─────────────────────────────────────────────────────────────────────┘
```

To generate a list of leaked memory blocks, use the API function
`purify_new_leaks` or `purify_all_leaks`, or set the option
`-leaks-at-exit=yes` (the default).
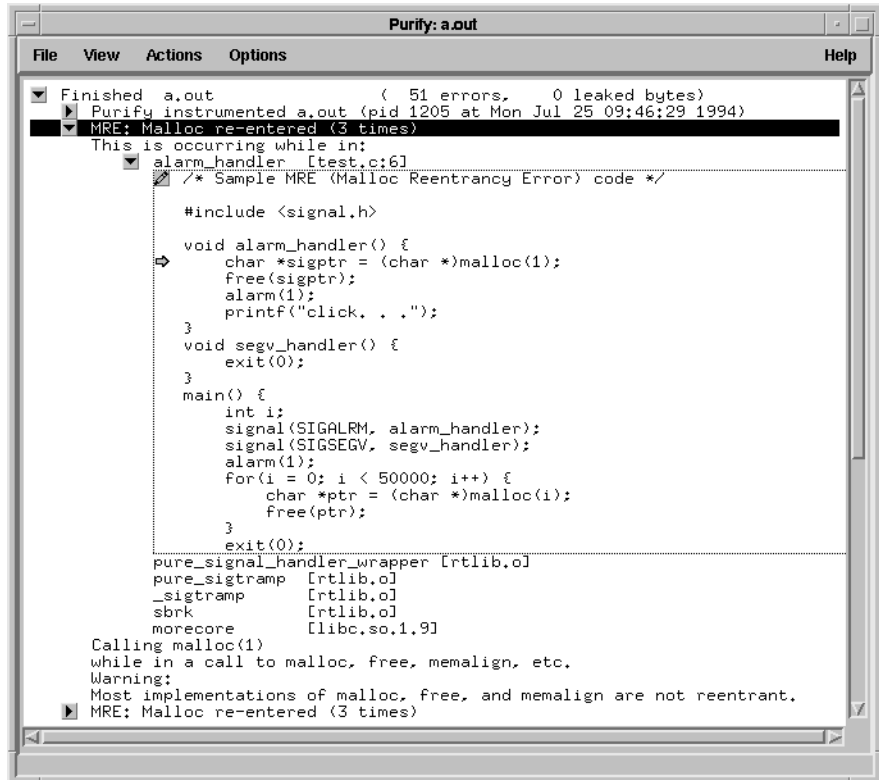
A memory leak is caused when the last pointer referencing a block
of memory is cleared, changed, or goes out of scope. If the section
of the program where the memory is allocated and leaked is
executed repeatedly, you might eventually run out of swap space.
This is a serious problem for long-running applications.

Memory that is allocated once, referenced by a pointer (perhaps
static or global) and never freed is not a leak and does not
generate an MLK message. Since it is allocated only once, you
cannot run out of memory during extended use of the program.

MRE   Malloc Reentrancy Error

An MRE message indicates that a reentrant call to `malloc`, `free`, or
a related function has been made. Since most default `malloc`
implementations are not reentrant, this will likely cause
problems. MRE indicates a *corrupting* error.
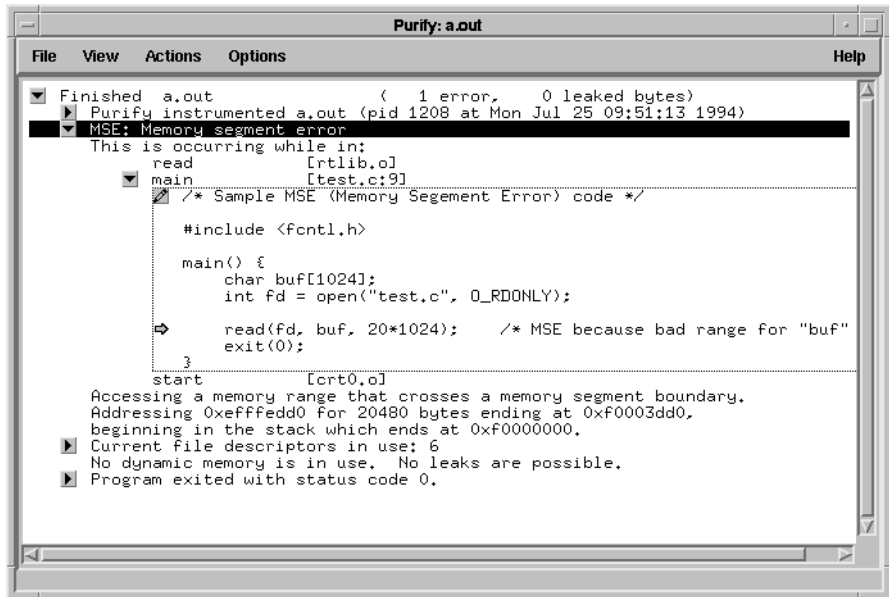
```
Purify: a.out

File   View   Actions   Options                                    Help

▼ Finished  a.out                    ( 51 errors,    0 leaked bytes)
  ▶ Purify instrumented a.out (pid 1205 at Mon Jul 25 09:46:29 1994)
  ▼ MRE: Malloc re-entered (3 times)
    This is occurring while in:
    ▼ alarm_handler   [test.c:6]
      /* Sample MRE (Malloc Reentrancy Error) code */

      #include <signal.h>

      void alarm_handler() {
   ⇨      char *sigptr = (char *)malloc(1);
          free(sigptr);
          alarm(1);
          printf("click. . .");
      }
      void segv_handler() {
          exit(0);
      }
      main() {
          int i;
          signal(SIGALRM, alarm_handler);
          signal(SIGSEGV, segv_handler);
          alarm(1);
          for(i = 0; i < 50000; i++) {
              char *ptr = (char *)malloc(i);
              free(ptr);
          }
          exit(0);
      pure_signal_handler_wrapper [rtlib.o]
      pure_sigtramp   [rtlib.o]
      _sigtramp       [rtlib.o]
      sbrk            [rtlib.o]
      morecore        [libc.so.1.9]
    Calling malloc(1)
    while in a call to malloc, free, memalign, etc.
    Warning:
    Most implementations of malloc, free, and memalign are not reentrant.
  ▶ MRE: Malloc re-entered (3 times)
```

Note: A number of C library functions call `malloc` as a side effect.
Avoid using these in interrupt/signal handlers.

MSE   Memory Segment Error

An MSE message indicates that your program is attempting to
address a piece of memory that spans potentially non-contiguous
segments of memory. The segments identified include the text
segment, the data segment, the heap, the stack and memory
mapped regions. MSE is a *warning* message.

```
─                               Purify: a.out                              ·  □

 File    View    Actions    Options                                        Help

 ▼ Finished   a.out              (   1 error,    0 leaked bytes)
  ▶  Purify instrumented a.out (pid 1208 at Mon Jul 25 09:51:13 1994)
    ▼ MSE: Memory segment error
      This is occurring while in:
          read            [rtlib.o]
       ▼  main            [test.c:9]
         ▨ /* Sample MSE (Memory Segement Error) code */

            #include <fcntl.h>

            main() {
                char buf[1024];
                int fd = open("test.c", O_RDONLY);

         ⇨      read(fd, buf, 20*1024);    /* MSE because bad range for "buf"
                exit(0);
            }
          start           [crt0.o]
      Accessing a memory range that crosses a memory segment boundary.
      Addressing 0xefffedd0 for 20480 bytes ending at 0xf0003dd0,
      beginning in the stack which ends at 0xf0000000.
  ▶  Current file descriptors in use: 6
      No dynamic memory is in use.  No leaks are possible.
  ▶  Program exited with status code 0.
```

An MSE message can be caused by any of the following:

- Calling a string or block-copy function with too large a size on a
  block of memory near the end of the data segment, so that the
  access spills into the heap. For example, calling `strlen` for a
  string not properly terminated can have this effect.

- Incorrect size calculation for read or write buffers, leading to
  requests for transactions with buffers of negative or large size.

- Infinite recursion, causing stack overflow.

NPR   Null Pointer Read

An NPR message indicates that your program is about to read from address zero (read from a NULL pointer). An SEGV signal will result. NPR indicates a *fatal* error.

One common cause of an NPR error is failure to check return status for a function expected to return a pointer to a string or an object. If the function returns NULL on failure, use of the NULL pointer leads to an NPR error.
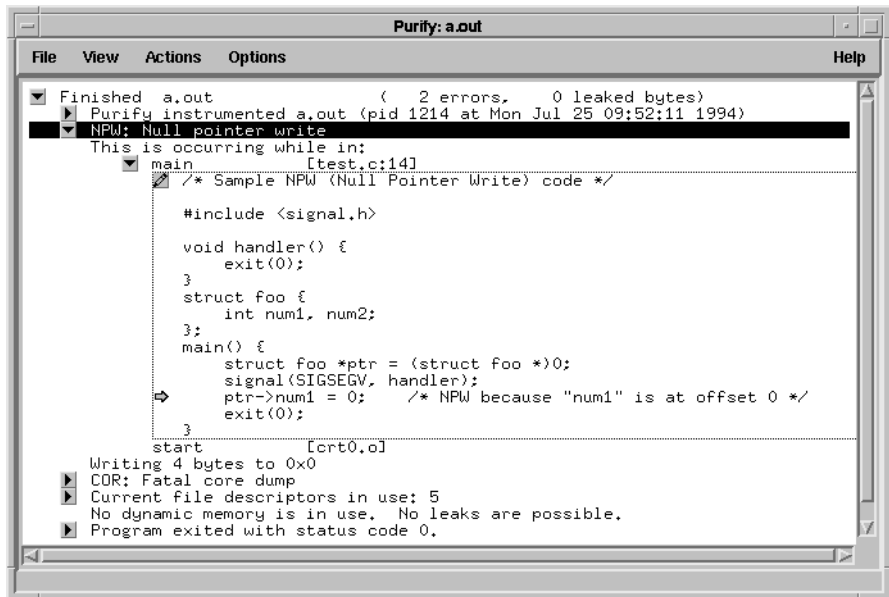
(HPUX) **Note:** HP-UX can be configured so that NPR messages are not fatal. However, they still represent serious errors. An SEGV happens only if you use the -z compiler option.

NPW   Null Pointer Write

An NPW message indicates that your program is about to write to
address zero (store to a NULL pointer). An SEGV signal will result.
NPW indicates a *fatal* error.

```
┌─────────────────────────────────────────────────────────────────────────┐
│ ─                              Purify: a.out                         ◢ □ │
├─────────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                                Help  │
├─────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out                  (   2 errors,    0 leaked bytes)      │
│    ▶ Purify instrumented a.out (pid 1214 at Mon Jul 25 09:52:11 1994)    │
│    ▼ NPW: Null pointer write                                             │
│      This is occurring while in:                                         │
│         ▼ main          [test.c:14]                                      │
│            ▨  /* Sample NPW (Null Pointer Write) code */                 │
│                                                                          │
│               #include <signal.h>                                        │
│                                                                          │
│               void handler() {                                           │
│                   exit(0);                                               │
│               }                                                          │
│               struct foo {                                               │
│                   int num1, num2;                                        │
│               };                                                         │
│               main() {                                                   │
│                   struct foo *ptr = (struct foo *)0;                     │
│                   signal(SIGSEGV, handler);                              │
│              ⇨    ptr->num1 = 0;    /* NPW because "num1" is at offset 0 */│
│                   exit(0);                                               │
│               }                                                          │
│           start          [crt0.o]                                        │
│         Writing 4 bytes to 0x0                                           │
│    ▶ COR: Fatal core dump                                                │
│    ▶ Current file descriptors in use: 5                                  │
│      No dynamic memory is in use.  No leaks are possible.                │
│    ▶ Program exited with status code 0.                                  │
│ ◀                                                                     ▶  │
└─────────────────────────────────────────────────────────────────────────┘
```
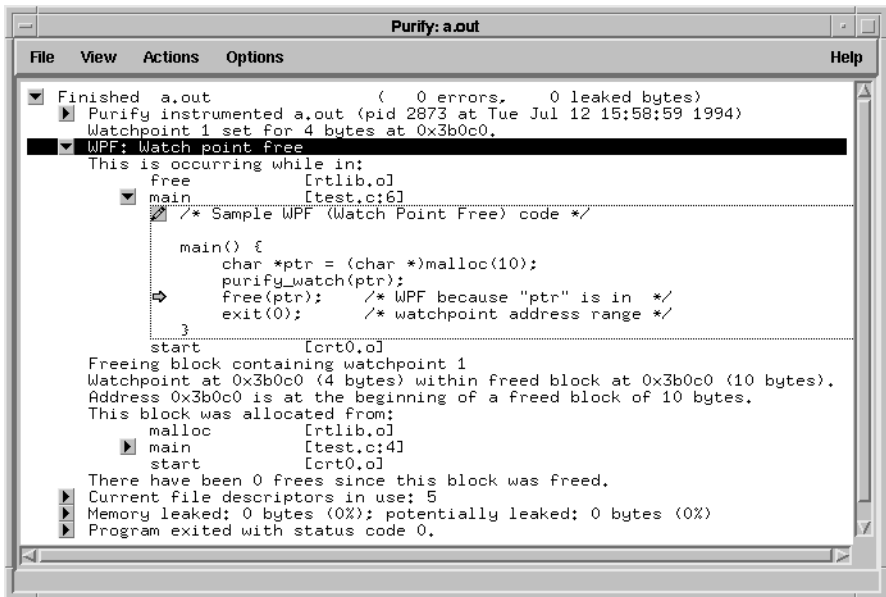
One common cause of an NPW error is failure to check return
status for a function expected to return a pointer to a string or an
object. If the function returns NULL on failure, use of the NULL
pointer leads to an NPW error.

PAR   Bad Parameter
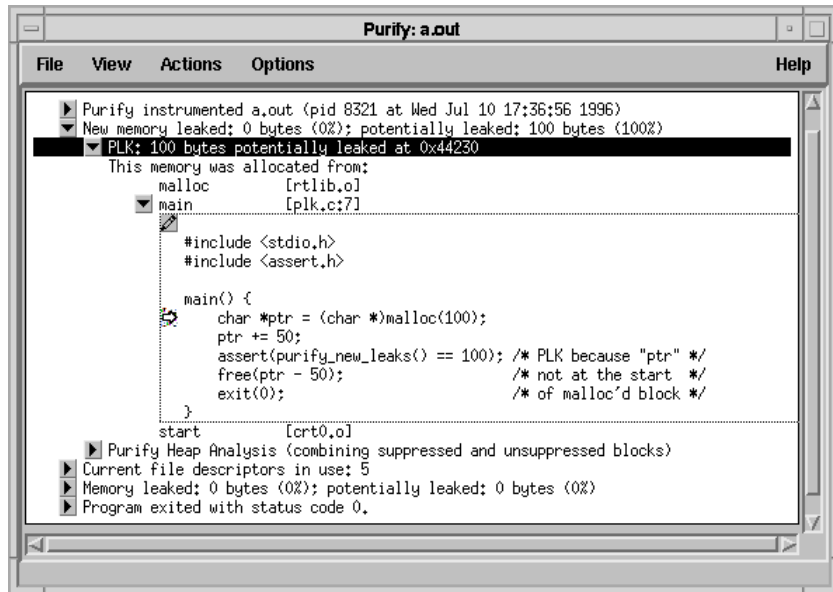
A PAR message indicates that your program has called a common
library function, such as write, with a bad parameter. Typically
Purify warns about bad parameters which involve pointer abuse,
such as passing NULL as the buffer to read or write. PAR is a
warning message.

```
Purify: a.out                                                    □

File   View   Actions   Options                                            Help

▼ Finished  a.out              (   2 errors,    0 leaked bytes)
  ▶ Purify instrumented a.out (pid 2822 at Tue Jul 12 15:48:06 1994)
  ▼ PAR: Bad function parameter
    This is occurring while in:
         malloc         [rtlib.o]
    ▼ main           [test.c:5]
      ▨  /* Sample PAR (Parameter) code */

           main() {
               char *ptr;
      ⇨       ptr = (char *)malloc(-1000);    /* PAR because "-1000" */
               free(ptr);                     /* to malloc is bad    */
               exit(0);
           }
         start          [crt0.o]
    malloc(-1000) size requested is negative.
  ▶ COR: Fatal core dump
```

PLK   Potential Memory Leak

A PLK message describes heap memory that you might have leaked. You have pointers only to the middle of the region. PLK is a *warning* message.

```
┌──────────────────────────────────────────────────────────────────┐
│ ⊟                          Purify: a.out                        ▫ ▢ │
├──────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                          Help  │
├──────────────────────────────────────────────────────────────────┤
│ ▶ Purify instrumented a.out (pid 8321 at Wed Jul 10 17:36:56 1996) │
│ ▶ New memory leaked: 0 bytes (0%); potentially leaked: 100 bytes (100%) │
│ ▼ PLK: 100 bytes potentially leaked at 0x44230                     │
│      This memory was allocated from:                               │
│          malloc        [rtlib.o]                                   │
│      ▼ main            [plk.c:7]                                   │
│        ▱                                                           │
│             #include <stdio.h>                                    │
│             #include <assert.h>                                   │
│                                                                    │
│             main() {                                              │
│        ▷        char *ptr = (char *)malloc(100);                  │
│                 ptr += 50;                                         │
│                 assert(purify_new_leaks() == 100); /* PLK because "ptr" */ │
│                 free(ptr - 50);                    /* not at the start  */ │
│                 exit(0);                           /* of malloc'd block */ │
│             }                                                     │
│          start           [crt0.o]                                 │
│ ▶ Purify Heap Analysis (combining suppressed and unsuppressed blocks) │
│ ▶ Current file descriptors in use: 5                              │
│ ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)   │
│ ▶ Program exited with status code 0.                              │
└──────────────────────────────────────────────────────────────────┘
```

In this example, 100 bytes are reported as potentially lost, not leaked. `ptr` *does not* point to the start of the block; it points 50 bytes into it. The `free` on line 10 assures that there is no leaked memory.
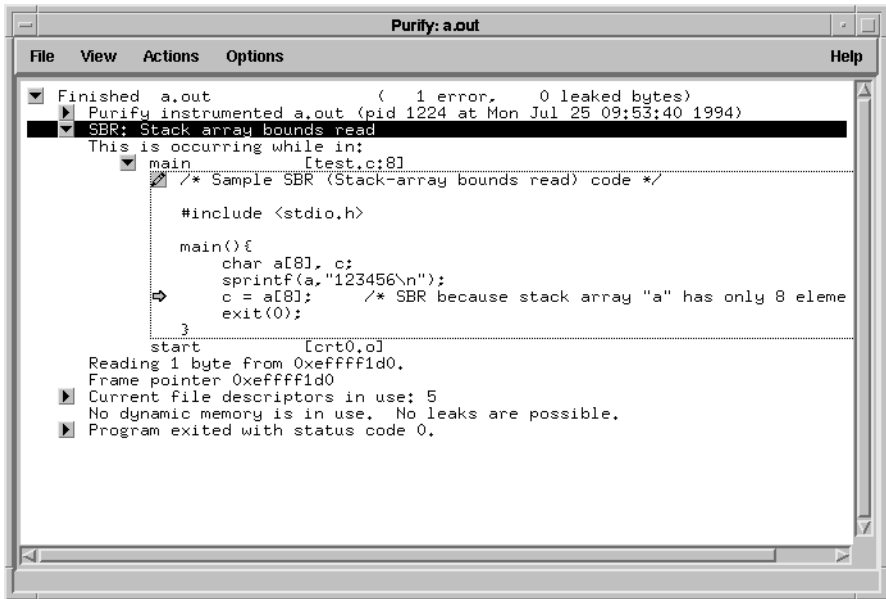
Memory in use can sometimes appear as a PLK if the pointer returned by `malloc` is offset. A common cause is referencing a substring within a large string. Another example is when a pointer to a C++ object is cast to the second or later base class of a multiply-inherited object. It is offset past the other base class objects.

Truly leaked memory can sometimes appear as a PLK, if some non-pointer integer within the program space, when interpreted as a pointer, points within an otherwise leaked block of memory. This is rather rare. Inspect the code to differentiate between these causes of PLK reports.

SBR   Stack Array Bounds Read

An SBR message indicates that your program is about to read across stack frame boundaries. This is similar to an ABR, but concerns a local variable instead of a malloc'd block. SBR is a *warning* message.
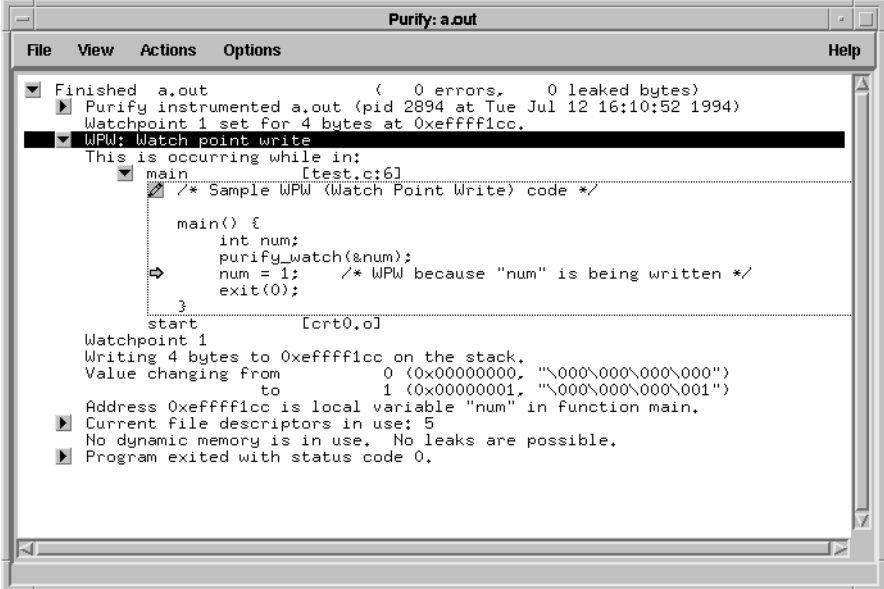
```
Purify: a.out

File   View   Actions   Options                                    Help

  Finished  a.out             (   1 error,    0 leaked bytes)
     Purify instrumented a.out (pid 1224 at Mon Jul 25 09:53:40 1994)
     SBR: Stack array bounds read
     This is occurring while in:
        main          [test.c:8]
          /* Sample SBR (Stack-array bounds read) code */

          #include <stdio.h>

          main(){
              char a[8], c;
              sprintf(a,"123456\n");
        ⇨    c = a[8];     /* SBR because stack array "a" has only 8 eleme
              exit(0);
          }
        start          [crt0.o]
     Reading 1 byte from 0xeffff1d0.
     Frame pointer 0xeffff1d0
     Current file descriptors in use: 5
     No dynamic memory is in use.  No leaks are possible.
     Program exited with status code 0.
```

An SBR error can be caused by any of the following:

- Making an automatic array too small, for example, failing to account for the terminating NULL in a string

- Forgetting to multiply by sizeof(type) when allocating for an array of objects

- Using an array index that is too large or negative

- Failing to NULL terminate a string

- Being off by one in copying elements up or down an array

- Passing too few arguments to a function
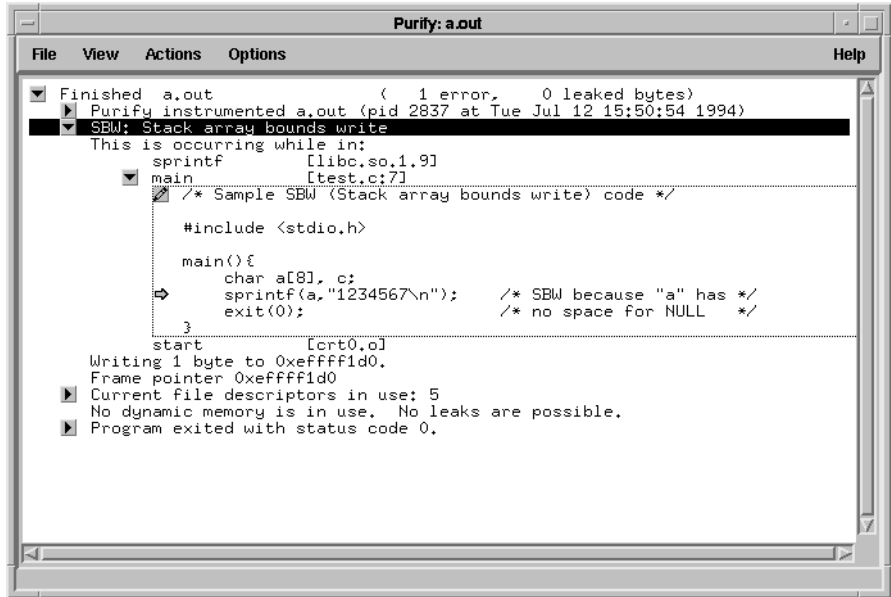
IRIX HPUX Purify does not support SBR messages on IRIX or HP-UX

SBW   Stack Array Bounds Write

An SBW message indicates that your program is about to write
across stack frame boundaries. This is similar to an ABW, but
concerns a local variable instead of a `malloc`'d block. SBW
indicates a *corrupting* error.

```
┌─────────────────────────────────────────────────────────────┐
│ ─                        Purify: a.out                   · □ │
├─────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                    Help  │
├─────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out            (  1 error,   0 leaked bytes)   │
│   ▶ Purify instrumented a.out (pid 2837 at Tue Jul 12 15:50:54 1994) │
│   ▼ SBW: Stack array bounds write                            │
│     This is occurring while in:                              │
│         sprintf       [libc.so.1.9]                          │
│     ▼ main           [test.c:7]                              │
│        ▨ /* Sample SBW (Stack array bounds write) code */    │
│                                                              │
│           #include <stdio.h>                                 │
│                                                              │
│           main(){                                            │
│               char a[8], c;                                  │
│           ⇨   sprintf(a,"1234567\n");    /* SBW because "a" has */ │
│               exit(0);                   /* no space for NULL   */ │
│           }                                                  │
│         start         [crt0.o]                               │
│     Writing 1 byte to 0xeffff1d0.                            │
│     Frame pointer 0xeffff1d0                                 │
│   ▶ Current file descriptors in use: 5                       │
│     No dynamic memory is in use.  No leaks are possible.     │
│   ▶ Program exited with status code 0.                       │
│                                                              │
└─────────────────────────────────────────────────────────────┘
```

An SBW error can be caused by any of the following:

- Making an automatic array too small, for example failing to
  account for the terminating NULL in a string

- Forgetting to multiply by `sizeof(type)` when allocating for an
  array of objects

- Using an array index that is too large or negative

- Failing to NULL terminate a string

- Being off by one in copying elements up or down an array
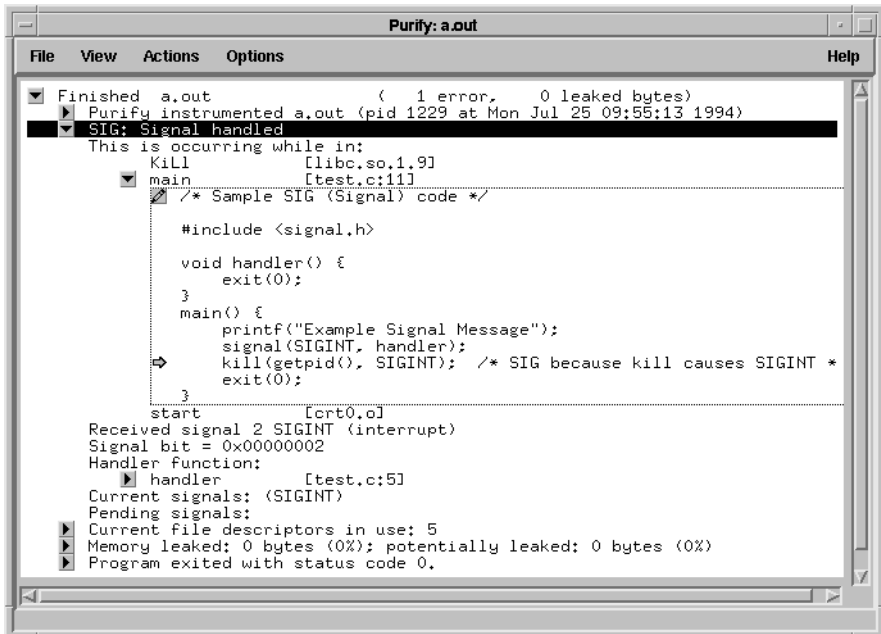
(IRIX) (HPUX) Purify does not support SBW messages on IRIX or HP-UX.

SIG   Signal

An SIG message indicates that your program has received a signal.
SIG is an *informational* message about signals.

```
  ┌─                          Purify: a.out                          ┌─┐┌─┐
  │ File  View   Actions   Options                                   Help │
  │ ──────────────────────────────────────────────────────────────────── │
  │ ▼ Finished  a.out                 (  1 error,   0 leaked bytes)      ▲ │
  │   ▶ Purify instrumented a.out (pid 1229 at Mon Jul 25 09:55:13 1994)   │
  │   ▼ SIG: Signal handled                                               │
  │     This is occurring while in:                                       │
  │          KiLl           [libc.so.1.9]                                 │
  │       ▼ main            [test.c:11]                                   │
  │         ▧ /* Sample SIG (Signal) code */                              │
  │                                                                       │
  │            #include <signal.h>                                        │
  │                                                                       │
  │            void handler() {                                           │
  │                exit(0);                                               │
  │            }                                                          │
  │            main() {                                                   │
  │                printf("Example Signal Message");                      │
  │                signal(SIGINT, handler);                               │
  │            ⇨   kill(getpid(), SIGINT);  /* SIG because kill causes SIGINT * │
  │                exit(0);                                               │
  │            }                                                          │
  │          start          [crt0.o]                                      │
  │     Received signal 2 SIGINT (interrupt)                              │
  │     Signal bit = 0x00000002                                           │
  │     Handler function:                                                 │
  │       ▶ handler         [test.c:5]                                    │
  │     Current signals: (SIGINT)                                         │
  │     Pending signals:                                                  │
  │   ▶ Current file descriptors in use: 5                               │
  │   ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)    │
  │   ▶ Program exited with status code 0.                               ▼ │
  │ ◁                                                                  ▷  │
  └───────────────────────────────────────────────────────────────────────┘
```
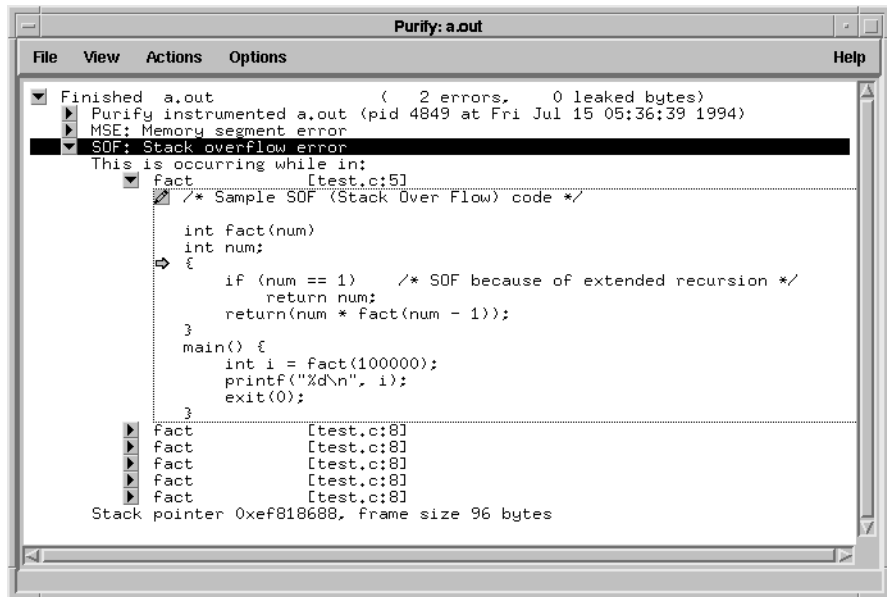
By default, Purify notifies you only about signals that normally
terminate the program. See -handle-signals and -ignore-signals
options on page 12-32.

SOF   Stack Overflow

An SOF message indicates that your program has overflowed the
stack, probably due to runaway recursion. SOF is a *warning*
message.

```
┌──────────────────────────────────────────────────────────────────┐
│ ─                       Purify: a.out                        ▪ □  │
├──────────────────────────────────────────────────────────────────┤
│ File   View   Actions   Options                             Help  │
├──────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out                ( 2 errors,   0 leaked bytes)    │
│   ▶ Purify instrumented a.out (pid 4849 at Fri Jul 15 05:36:39 1994)│
│   ▶ MSE: Memory segment error                                    │
│   ▼ SOF: Stack overflow error                                    │
│     This is occurring while in:                                  │
│     ▼ fact         [test.c:5]                                    │
│       ▨  /* Sample SOF (Stack Over Flow) code */                 │
│                                                                   │
│          int fact(num)                                           │
│          int num;                                                │
│       ⇨ {                                                        │
│              if (num == 1)    /* SOF because of extended recursion */│
│                  return num;                                     │
│              return(num * fact(num - 1));                        │
│          }                                                       │
│          main() {                                                │
│              int i = fact(100000);                               │
│              printf("%d\n", i);                                  │
│              exit(0);                                            │
│          }                                                       │
│     ▶ fact         [test.c:8]                                    │
│     ▶ fact         [test.c:8]                                    │
│     ▶ fact         [test.c:8]                                    │
│     ▶ fact         [test.c:8]                                    │
│     ▶ fact         [test.c:8]                                    │
│     Stack pointer 0xef818688, frame size 96 bytes                │
│ ◀                                                            ▷    │
└──────────────────────────────────────────────────────────────────┘
```

The SunOS 4 dynamic linker (for shared libraries) uses the low
area of the stack segment for special purposes. If the stack size
grows into this area, subsequent calls to shared library functions
can cause the linker to crash the program.

A UMC message indicates that an uninitialized value is being copied from one memory location to another (e.g. an assignment). Such copies are normally harmless copying of padding fields in structures. UMC is a *warning* message.

```
┌─────────────────────────────────────────────────────────────────────┐
│ ─                         Purify: a.out                       ⋅ │□│
├─────────────────────────────────────────────────────────────────────┤
│ File   View   Actions   Options                               Help  │
├─────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (   1 error,   0 leaked bytes)        │
│   ▶ Purify instrumented a.out (pid 2863 at Tue Jul 12 15:57:40 1994) │
│   ▼ UMC: Uninitialized memory copy                                   │
│     This is occurring while in:                                      │
│     ▼ main          [test.c:4]                                       │
│       ▨  /* Sample UMC (Uninitialized memory copy) code */           │
│          main() {                                                    │
│               int num1, num2;                                        │
│        ⇨      num1 = num2;    /* UMC because "num2" copied uninitialized */ │
│          }                                                           │
│        start          [crt0.o]                                       │
│     Copying 4 bytes from 0xeffff1c8 on the stack.                    │
│     Address 0xeffff1c8 is local variable "num2" in function main.    │
│   ▶ Current file descriptors in use: 5                               │
│     No dynamic memory is in use.  No leaks are possible.             │
│   ▶ Program exited with status code 1.                              │
│                                                                      │
└─────────────────────────────────────────────────────────────────────┘
```

By default, Purify suppresses UMC messages in the global .purify file because they can generate excessive output and reduce your program's performance. To unsuppress UMC messages, comment out the line in the <purifyhome>/.purify file that reads:

```
suppress umc *
```

by adding a hash mark (#) at the beginning of the line:

```
#suppress umc *
```

or add the line:

```
unsuppress umc *
```
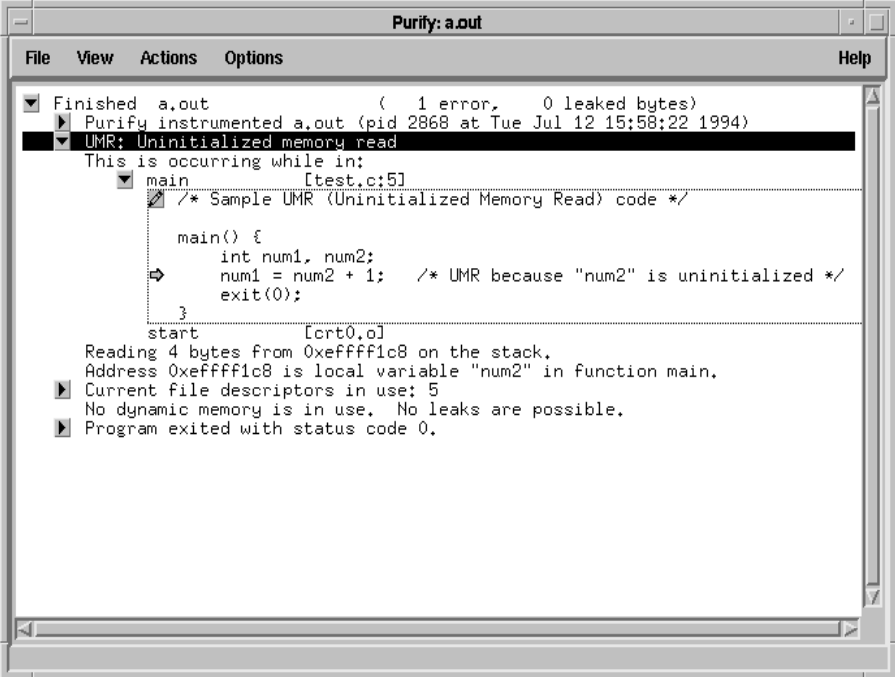
to the .purify file in the directory where your program resides.

UMR   Uninitialized Memory Read

A UMR message indicates that your program is about to read uninitialized memory. UMR is a *warning* message.

Often, uninitialized memory will be zero, especially during unit testing. Your program will seem to perform correctly but the UMR can eventually cause incorrect behavior.

It is common, and correct behavior, for a program to copy uninitialized data from one variable to another. A frequent case is during structure assignment when the structure being copied has inaccessible padding bytes. For this reason, Purify does not report UMR messages on copies, but instead reports a (suppressed) UMC and propagates the uninitialized status to the destination of the copy.

WPF   Watchpoint Free

A WPF message indicates that your program is about to `free` a block of memory containing a watchpoint. WPF is an *informational* message about memory.
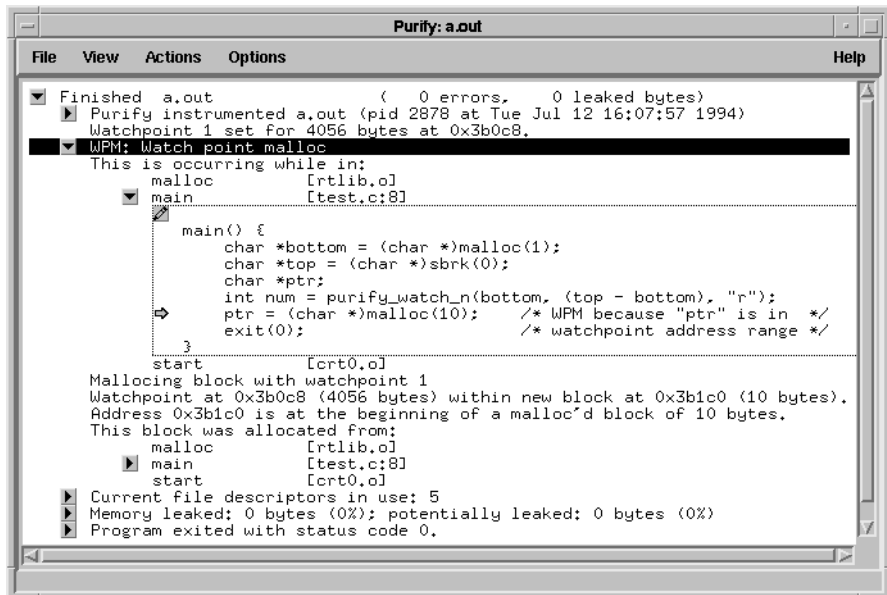
```
┌─────────────────────────────────────────────────────────────────────────┐
│ ─                         Purify: a.out                            · □ │
├─────────────────────────────────────────────────────────────────────────┤
│ File   View   Actions   Options                                    Help │
├─────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (   0 errors,    0 leaked bytes)        ▲ │
│   ▶ Purify instrumented a.out (pid 2873 at Tue Jul 12 15:58:59 1994)     │
│     Watchpoint 1 set for 4 bytes at 0x3b0c0.                             │
│ ▼ WPF: Watch point free                                                  │
│     This is occurring while in:                                          │
│         free            [rtlib.o]                                        │
│     ▼ main             [test.c:6]                                        │
│      ▨ /* Sample WPF (Watch Point Free) code */                         │
│                                                                          │
│         main() {                                                         │
│             char *ptr = (char *)malloc(10);                             │
│             purify_watch(ptr);                                           │
│         ⇨   free(ptr);    /* WPF because "ptr" is in  */                │
│             exit(0);      /* watchpoint address range */                │
│         }                                                                │
│         start           [crt0.o]                                        │
│     Freeing block containing watchpoint 1                               │
│     Watchpoint at 0x3b0c0 (4 bytes) within freed block at 0x3b0c0 (10 bytes). │
│     Address 0x3b0c0 is at the beginning of a freed block of 10 bytes.   │
│     This block was allocated from:                                       │
│         malloc          [rtlib.o]                                        │
│     ▶ main             [test.c:4]                                        │
│         start           [crt0.o]                                        │
│     There have been 0 frees since this block was freed.                 │
│ ▶ Current file descriptors in use: 5                                    │
│ ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)         │
│ ▶ Program exited with status code 0.                                 ▽ │
│ ◁                                                                    ▷ │
└─────────────────────────────────────────────────────────────────────────┘
```

WPM   Watchpoint Malloc

A WPM message indicates that your program is about to `malloc` a
block of memory containing a watchpoint. WPM is an
informational message about memory.

```
┌──────────────────────────────────────────────────────────────────────────────┐
│ ─│                          Purify: a.out                                │ ■ □ │
├──────────────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                                      Help  │
├──────────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (   0 errors,   0 leaked bytes)                 │
│   ▶ Purify instrumented a.out (pid 2878 at Tue Jul 12 16:07:57 1994)           │
│     Watchpoint 1 set for 4056 bytes at 0x3b0c8.                                │
│   ▼ WPM: Watch point malloc                                                    │
│     This is occurring while in:                                                │
│         malloc          [rtlib.o]                                              │
│      ▼ main            [test.c:8]                                              │
│        ┌──┐                                                                     │
│        │🖉 │    main() {                                                        │
│        └──┘        char *bottom = (char *)malloc(1);                           │
│                    char *top = (char *)sbrk(0);                                │
│                    char *ptr;                                                   │
│                    int num = purify_watch_n(bottom, (top - bottom), "r");      │
│            ⇨       ptr = (char *)malloc(10);     /* WPM because "ptr" is in  */ │
│                    exit(0);                       /* watchpoint address range */│
│                }                                                                │
│         start           [crt0.o]                                               │
│     Mallocing block with watchpoint 1                                          │
│     Watchpoint at 0x3b0c8 (4056 bytes) within new block at 0x3b1c0 (10 bytes). │
│     Address 0x3b1c0 is at the beginning of a malloc'd block of 10 bytes.       │
│     This block was allocated from:                                             │
│         malloc          [rtlib.o]                                              │
│      ▶ main            [test.c:8]                                              │
│         start           [crt0.o]                                               │
│   ▶ Current file descriptors in use: 5                                         │
│   ▶ Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)              │
│   ▶ Program exited with status code 0.                                         │
└──────────────────────────────────────────────────────────────────────────────┘
```
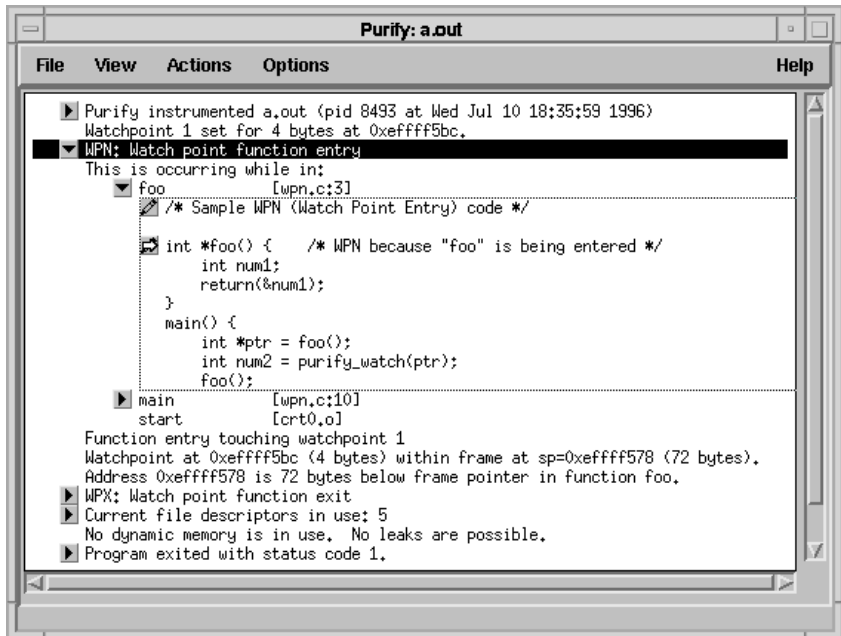
WPN   Watchpoint Entry

A WPN message indicates that your program has just entered a
function that is allocating local variables on the stack in watched
memory. WPN is an *informational* message about memory.

```
┌─────────────────────────────────────────────────────────────────────┐
│ ▭                          Purify: a.out                     ▫  □     │
├─────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                            Help   │
├─────────────────────────────────────────────────────────────────────┤
│ ▶ Purify instrumented a.out (pid 8493 at Wed Jul 10 18:35:59 1996)  ▲ │
│   Watchpoint 1 set for 4 bytes at 0xeffff5bc.                         │
│ ▼ WPN: Watch point function entry                                    │
│   This is occurring while in:                                        │
│   ▼ foo             [wpn.c:3]                                         │
│     ▨ /* Sample WPN (Watch Point Entry) code */                      │
│                                                                      │
│     ▨ int *foo() {    /* WPN because "foo" is being entered */       │
│           int num1;                                                  │
│           return(&num1);                                             │
│       }                                                              │
│       main() {                                                       │
│           int *ptr = foo();                                          │
│           int num2 = purify_watch(ptr);                              │
│           foo();                                                     │
│   ▶ main            [wpn.c:10]                                        │
│     start           [crt0.o]                                         │
│   Function entry touching watchpoint 1                               │
│   Watchpoint at 0xeffff5bc (4 bytes) within frame at sp=0xeffff578 (72 bytes).│
│   Address 0xeffff578 is 72 bytes below frame pointer in function foo.│
│ ▶ WPX: Watch point function exit                                     │
│ ▶ Current file descriptors in use: 5                                 │
│   No dynamic memory is in use.  No leaks are possible.               │
│ ▶ Program exited with status code 1.                                ▼ │
│ ◁─────────────────────────────────────────────────────────────────▷ │
└─────────────────────────────────────────────────────────────────────┘
```
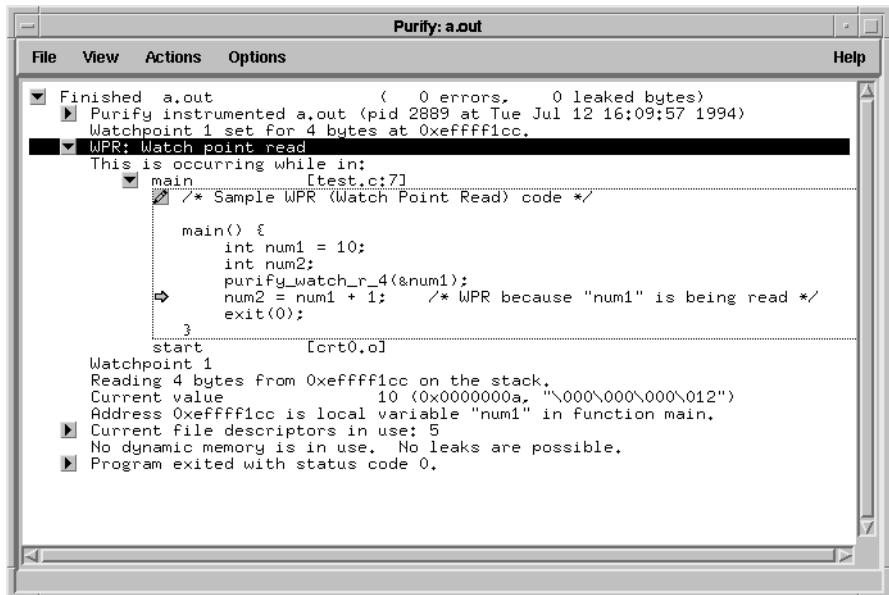
WPR   Watchpoint Read

A WPR message indicates that your program is about to read from memory that has a read-type watchpoint on it. WPR is an *informational* message about memory.

```
┌────────────────────────────────────────────────────────────────────────────┐
│ ─                              Purify: a.out                          ▫ ▫   │
├────────────────────────────────────────────────────────────────────────────┤
│ File    View    Actions    Options                                    Help   │
├────────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out                 (   0 errors,    0 leaked bytes)           │
│    ▶ Purify instrumented a.out (pid 2889 at Tue Jul 12 16:09:57 1994)        │
│      Watchpoint 1 set for 4 bytes at 0xeffff1cc.                             │
│    ▼ WPR: Watch point read                                                   │
│      This is occurring while in:                                             │
│       ▼ main          [test.c:7]                                             │
│          ▨ /* Sample WPR (Watch Point Read) code */                          │
│                                                                              │
│             main() {                                                         │
│                 int num1 = 10;                                               │
│                 int num2;                                                    │
│                 purify_watch_r_4(&num1);                                     │
│           ⇨     num2 = num1 + 1;    /* WPR because "num1" is being read */   │
│                 exit(0);                                                     │
│             }                                                                │
│          start          [crt0.o]                                            │
│      Watchpoint 1                                                            │
│      Reading 4 bytes from 0xeffff1cc on the stack.                          │
│      Current value            10 (0x0000000a, "\000\000\000\012")           │
│      Address 0xeffff1cc is local variable "num1" in function main.          │
│    ▶ Current file descriptors in use: 5                                     │
│      No dynamic memory is in use.  No leaks are possible.                    │
│    ▶ Program exited with status code 0.                                     │
│                                                                              │
└────────────────────────────────────────────────────────────────────────────┘
```

WPW   Watchpoint Write

A WPW message indicates that your program is about to write to
memory that has a watchpoint on it. WPW is an *informational*
message about memory.
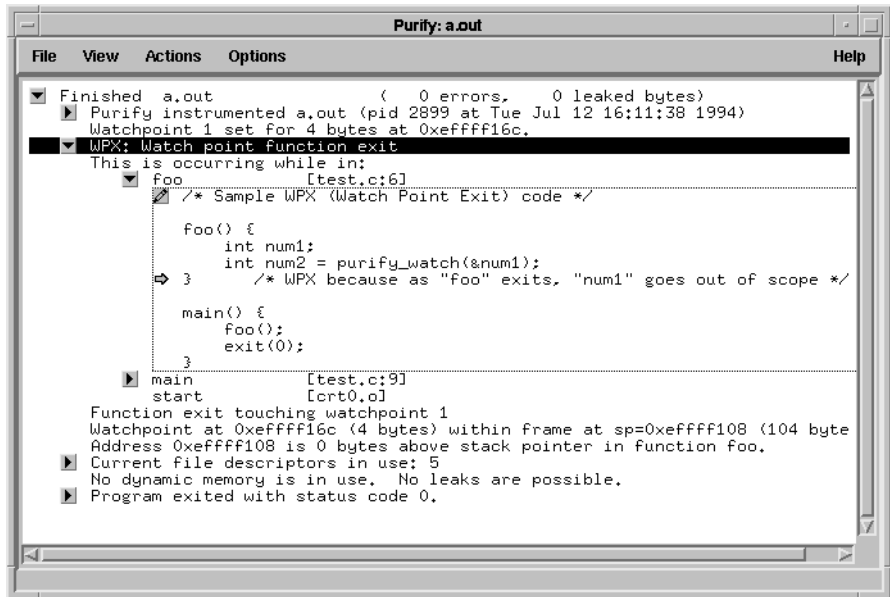
```
┌──────────────────────────────────────────────────────────────────────────┐
│ ─                           Purify: a.out                           ◁  □  │
├──────────────────────────────────────────────────────────────────────────┤
│ File   View   Actions   Options                                     Help  │
├──────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (   0 errors,    0 leaked bytes)        ▲  │
│   ▶ Purify instrumented a.out (pid 2894 at Tue Jul 12 16:10:52 1994)      │
│     Watchpoint 1 set for 4 bytes at 0xeffff1cc.                           │
│ ▼ WPW: Watch point write                                                  │
│     This is occurring while in:                                           │
│       ▼ main           [test.c:6]                                         │
│         ▨ /* Sample WPW (Watch Point Write) code */                       │
│                                                                           │
│             main() {                                                      │
│                 int num;                                                   │
│                 purify_watch(&num);                                        │
│           ⇨     num = 1;    /* WPW because "num" is being written */      │
│                 exit(0);                                                   │
│             }                                                             │
│         start          [crt0.o]                                           │
│     Watchpoint 1                                                          │
│     Writing 4 bytes to 0xeffff1cc on the stack.                          │
│     Value changing from        0 (0x00000000, "\000\000\000\000")        │
│                        to       1 (0x00000001, "\000\000\000\001")        │
│     Address 0xeffff1cc is local variable "num" in function main.          │
│   ▶ Current file descriptors in use: 5                                    │
│     No dynamic memory is in use.  No leaks are possible.                   │
│   ▶ Program exited with status code 0.                                    │
│                                                                        ▽  │
│ ◁                                                                      ▷  │
└──────────────────────────────────────────────────────────────────────────┘
```

WPX   Watchpoint Exit

A WPX message indicates that your program has exited a function that had allocated local variables on the stack in watched memory. WPX is an *informational* message about memory.

ZPR   Zero Page Read

A ZPR message indicates that your program is about to read from
the zeroth page of memory—read from a bad pointer. An SEGV
signal can result. ZPR is a *fatal* error.

A ZPR error can be caused by a failure to check return status for a
function expected to return a pointer to a structure or an object. If
the function returns NULL on failure, accessing a structure field
from the NULL pointer leads to a ZPR error.
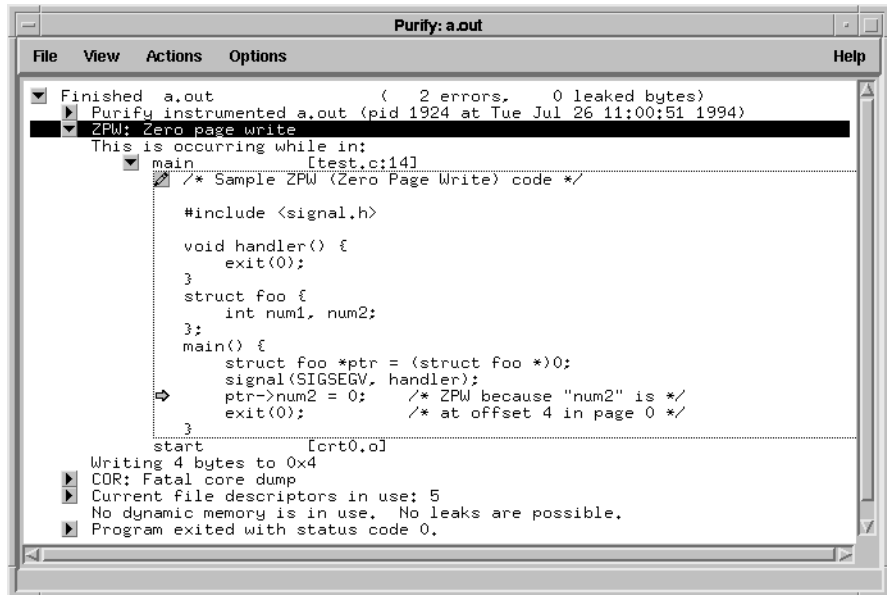
(HPUX) **Note:** HP-UX can be configured so that ZPR messages are not
fatal. However, they still represent serious errors. A SEGV happens
only if you use the −z compiler option.

ZPW   Zero Page Write

A ZPW message indicates that your program is about to write to
the zeroth page of memory—store to a bad pointer. An SEGV signal
can result. ZPW indicates a *fatal* error.

```
┌─────────────────────────────────────────────────────────────────────────┐
│ —                            Purify: a.out                        ▪ ▫    │
├─────────────────────────────────────────────────────────────────────────┤
│  File    View    Actions    Options                               Help   │
├─────────────────────────────────────────────────────────────────────────┤
│ ▼ Finished  a.out              (   2 errors,    0 leaked bytes)           │
│   ▶ Purify instrumented a.out (pid 1924 at Tue Jul 26 11:00:51 1994)      │
│   ▼ ZPW: Zero page write                                                  │
│     This is occurring while in:                                           │
│       ▼ main          [test.c:14]                                         │
│         ▨ /* Sample ZPW (Zero Page Write) code */                        │
│                                                                           │
│              #include <signal.h>                                          │
│                                                                           │
│              void handler() {                                             │
│                  exit(0);                                                  │
│              }                                                            │
│              struct foo {                                                 │
│                  int num1, num2;                                          │
│              };                                                          │
│              main() {                                                     │
│                  struct foo *ptr = (struct foo *)0;                       │
│                  signal(SIGSEGV, handler);                                │
│              ⇨   ptr->num2 = 0;     /* ZPW because "num2" is */           │
│                  exit(0);           /* at offset 4 in page 0 */           │
│              }                                                            │
│         start          [crt0.o]                                           │
│       Writing 4 bytes to 0x4                                              │
│   ▶ COR: Fatal core dump                                                  │
│   ▶ Current file descriptors in use: 5                                    │
│     No dynamic memory is in use.  No leaks are possible.                  │
│   ▶ Program exited with status code 0.                                    │
└─────────────────────────────────────────────────────────────────────────┘
```

A ZPW error can be caused by a failure to check the return status
for a function expected to return a pointer to a structure or an
object. If the function returns NULL on failure, writing to a
structure field of the NULL pointer leads to a ZPW error.

11

Using Purify Options and API Functions

This chapter describes how to use Purify options and Application Programming Interface (API) functions. It includes:

- Purify option syntax
- Purify option types
- Purify option processing

And instructions for:

- Using the `-ignore-runtime-environment` option
- Calling Purify API functions from a debugger
- Calling Purify API functions from your program
- Linking with the Purify stubs library

For a complete list of Purify options and API functions, see Chapter 12, "Options and API Reference."

Using Purify options

Purify option syntax

A Purify option consists of a word or phrase that begins with a hyphen. For example:

```
-leaks-at-exit=no
```

- The leading hyphen is required.
- No space is allowed on either side of the equal sign (=).
- Purify ignores case, hyphens, and underscores in the option name. For example, the option `-leaks-at-exit` is equivalent to `-LEAKS_AT_EXIT` and `-LeaksAtExit`.
- For options that take a list of directories, you can specify the directory names separated by spaces or colons (:). For example:

```
% purify -user-path='/usr/home/program /usr/home/program1'
```

  or

```
% purify -user-path=/usr/home/program:/usr/home/program1
```

- Specify a list of addresses or signals separated by commas (,). For example:

```
% purify -mail-to-user=chris,pat,kam
```

- You can use wildcards. For example, in filenames: `program*` matches `program4`, `/dira/dirb/program.o`, and `/dira/dirb/program1.o`.

Using conversion characters in filenames

You can use conversion characters when you specify filenames for options such as `-log-file`, `-watchpoints-file`, and `-view-file`. Purify supports these conversion characters:

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

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

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

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

Purify option types

Purify uses three types of options: boolean, string, and integer.

- Boolean options take the values `yes` or `no`, or `true` or `false`. If you do not specify an explicit value, the value is `yes`. For example, the option settings `-leaks-at-exit` and `-leaks-at-exit=yes` are identical.

- String options can be a string of any kind. String options are used for program, directory and file names, lists of file descriptor numbers, and lists of mail users.

  If you do not specify an explicit value for a string option, the value is cleared. For example, the option `-log-file=./pureout` routes Purify messages to the file `pureout` in the current directory. The option `-log-file=`, without a value, clears any default specification of a logfile.

- Integer options can be set to any whole number. For example, the option `-chain-length=10` increases the length of the printed function call chains from the default of 6 to 10. An optional sign can be specified. Integer values cannot be cleared.

Purify option processing

You can specify Purify options in the Viewer, in environment variables, and on the link line. Purify processes options in this order (highest precedence first):

- Options specified in the Viewer
- Options specified in the `PURIFYOPTIONS` or `PUREOPTIONS` environment variables
- Options specified on the link line

Specifying options in the Purify Viewer

To specify options in the Purify Viewer, select **Runtime** from the Options menu.

The Runtime dialog displays the option values in use for the current program run.

Purify applies the options set in the Viewer on subsequent runs of the program displayed in the Viewer. Options set in the Viewer do *not* modify either the default values in the program, or the environment variables. When you quit the Viewer and rerun the application, the option values revert to their original settings.

Note: Purify sends the values of options specified in the Viewer to the application at start-up time. Therefore, you cannot modify options while your program is running.

Specifying options in environment variables

You can specify any Purify option in the `PURIFYOPTIONS` and `PUREOPTIONS` environment variables. The option values specified in `PURIFYOPTIONS` take precedence over `PUREOPTIONS`.

Purify applies *build-time* options specified in environment variables when a Purify'd application is built. Any build-time options on the link line override environment variables.

Purify applies *run-time* options specified in environment variables when you run the Purify'd program. The environment values in

force when you run the program override any defaults specified on the link line.

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

```
csh     % setenv PURIFYOPTIONS "-log-file=new $PURIFYOPTIONS"
sh, ksh $ PURIFYOPTIONS="-log-file=new $PURIFYOPTIONS"; export\
          PURIFYOPTIONS
```

Setting options for all Pure Software products

You might wish to set options such as `-cache-dir=alternate/dir` to apply to all users and all Pure Software products. If your site has a central shared file that is sourced by all users' `.cshrc` or `.profile` files, you can use the `PUREOPTIONS` environment variable to set options that apply to your entire site and to *all* Pure Software products.

Specifying options on the link line

You can specify any Purify option on the link line. For example:

```
purify -cache-dir=$HOME/pcache -always-use-cache-dir $CC ...
```

Purify applies build-time options to the Purify build command being run. Purify builds run-time options into the executable so that they become the default values for the Purify'd executable. This is a convenient way to build a program with nonstandard default values for run-time options. For example:

```
purify -chain-length=12 $CC ...
```

Using the -ignore-runtime-environment option

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

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

The `-ignore-runtime-environment` option also builds in suppressions. If you do not explicitly specify the `-suppression-filenames` option, Purify uses the default suppression files `.purify` and `.purify.<platform>`.

When the Purify'd program is run, Purify ignores the current option values set in environment variables in preference to the built-in values. In the Viewer, Purify lets you display the run-time options set for the executable but does not let you change them.

Use the `-ignore-runtime-environment` option when:

- You want someone else to run your program without their run-time environment modifying your run-time option specifications.
- Your program is started automatically by another program and you cannot set the environment variable for that program.
- You have several Purify'd programs running at one time and you cannot specify options for each program.
- You use the `-mail-to-user` option. See "Mailing Purify output to developers" on page 6-6, and "Mail mode option" on page 12-13.

To find out what options are built into a Purify'd program, use:

```
% <purifyhome>/purify_what_options <program name>
```

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

Using Purify API functions

You can call Purify API functions from a debugger or from your program. Unless otherwise specified, Purify functions return 0, indicating success.

Calling Purify API functions from a debugger

You can call many Purify functions interactively from a debugger. Some, such as the watchpoint functions, the leak-detection functions, `purify_describe`, and `purify_what_colors`, are especially useful when used with a debugger:

```
(gdb)      print purify_describe(addr)
(dbx)      call purify_what_colors(buf, sizeof(buf))
(xdb)      p purify_describe(addr)
```

Using the function purify_stop_here

To enhance the power of your debugger, set a breakpoint on the function `purify_stop_here`. This causes your debugger to stop on every Purify error message, right after the message is displayed and *before* the error actually occurs in your program:

```
(gdb)      break purify_stop_here
(dbx)      stop in purify_stop_here
(xdb)      b purify_stop_here
```

Note: Do *not* call `purify_stop_here` directly from your program. Instead, set a breakpoint on it. Use a call to `purify_stop_here_internal` to force a call to `purify_stop_here`.

Calling Purify API functions from your program

To call Purify functions from ANSI C and C++ programs, include the file `purify.h`:

```
# include <purify.h>
```

This header file is located in the same directory as Purify. You might need to add the compiler option `-I<purifyhome>` in your makefile to locate it.

`IRIX` On IRIX, you must also link with the Purify API stubs library. See "Linking with the Purify stubs library on IRIX" below.

`Sun OS4` `Solaris` `HPUX` ### Linking with the Purify stubs library

If you call Purify functions in your program, you should link with the Purify API stub library. This is a small library that stubs out all the Purify API functions when you are *not* using Purify. When you *are* using Purify, the stubs are ignored.

Add the library `<purifyhome>/libpurify_stubs.a` to your link line.

Linking with the Purify stubs library on IRIX

`IRIX` If you call Purify functions in your program, you should link with the Purify API stub library. This is a small library that stubs out all the Purify API functions. When you are *not* using Purify, these stubs satisfy the linker; when you *are* using Purify, the stubs are overridden by Purify but still required. Purify on IRIX includes two versions of the stub library:

- `libpurify_stubs.so`
- `libpurify_stubs.a`

Note: If you are using the N32 Application Binary Interface (ABI), link with these versions of the stub library:

- `libpurify_stubs_n32.so`
- `libpurify_stubs_n32.a`

Installing libpurify_stubs.so

In the examples below, replace `<purifyhome>` with the path to your Purify installation.

If `ld` is available when you install Purify, the full pathname is automatically encoded in `libpurify_stubs.so`.

If `ld` is not available, Purify uses the default path `/usr/pure/purify`.

If you do not install Purify in `/usr/pure/purify` and `ld` is not available, `libpurify_stubs.so` is installed without a built-in path. You can specify the pathname by typing:

```
% ld -shared -all -soname \
<purifyhome>/libpurify_stubs.so -o \
<purifyhome>/libpurify_stubs.so \
<purifyhome>/libpurify_stubs.so.std
```

You can also use `libpurify_stubs.so` without a path, then specify it at run time by typing:

```
% cc <program>.c `purify -printhomedir`/libpurify_stubs.so
% setenv LD_LIBRARY_PATH `purify -printhomedir`
% a.out
```

Linking with libpurify_stubs.so

During development, link your program with the `libpurify_stubs.so` library by typing:

```
cc <program>.c <purifyhome>/libpurify_stubs.so
```

This resolves any API references in your code and lets you use the API when you Purify the program.

Warning: Do not ship a program linked with the `libpurify_stubs.so` library. It will cause a fatal error when the library is not found at run time.

Linking with libpurify_stubs.a

To produce a program for non-Purify users, link with the
`libpurify_stubs.a` library by typing:

```
cc <program>.c <purifyhome>/libpurify_stubs.a
```

This disables Purify API functions. If you Purify a program linked
with this library, Purify API functions are ignored.

12

Purify Options and API Reference

This chapter describes Purify options and API functions.

Build-time options quick reference

Build-time options let you control how your program is Purify'd
and linked.

Build-time option	Default	Page
-always-use-cache-dir	no	12-6
-auto-mount-prefix	/tmp_mnt	12-6
-cache-dir	<purifyhome>/cache	12-6
-collector *	not set	12-7
-forbidden-directories	system dependent	12-6
-ignore-runtime-environment	no	12-8
-force-rebuild	not set	12-6
-help		12-8
-linker *	system dependent	12-7
-print-home-dir		12-7
-static-checking	yes	12-24
-static-checking-guardzone	16	12-24
-static-checking-default	safe	12-25
-usage		12-8
-version		12-8

IRIX * The -linker and -collector options are not supported on IRIX.

Run-time options quick reference

Run-time options let you specify which errors Purify reports, the information contained in the error messages, the appearance of messages, and where they are printed.

Run-time option	Default	Page
-append-logfile	no	12-21
-auto-mount-prefix	/tmp_mnt	12-6
-chain-length	6	12-18
-copy-fd-output-to-logfile	not set	12-9
-exit-status	no	12-10
-fds	26	12-12
-fds-inuse-at-exit	yes	12-12
-follow-child-processes	no	12-31
-free-queue-length	100	12-14
-free-queue-threshold	10000	12-14
-freeze-on-error	no	12-31
-g++	no	12-7
-handle-signals	not set	12-32
-ignore-signals	not set	12-32
-inuse-at-exit	no	12-16
-jit-debug	not set	12-31
-leaks-at-exit	yes	12-16
-log-file	not set	12-21
-mail-to-user	not set	12-13
-max-threads	20	12-27
-messages	first	12-19
-output-limit	1000000	12-22
-pointer-mask	0xffffffff	12-16
-pointer-offset	0	12-16
-program-name	argv[0]	12-8
-run-at-exit	not set	12-10

Run-time option	Default	Page
-search-mmaps	no	12-16
-show-directory	no	12-18
-show-pc	no	12-18
-show-pc-offset	no	12-18
-suppression-filenames	system dependent	12-26
-threads	no	12-27
-thread-report-at-exit	no	12-27
-thread-stack-change	0x1000	12-27
-user-path	not set	12-22
-view	not set	12-21
-view-file	not set	12-22
-watchpoints-file	./<program-name>.watchpoints	12-29
-windows	not set	12-21

API functions quick reference

Unless otherwise specified, Purify functions return 0, indicating success.

Function	Page
`purify_all_fds_inuse (void)`	12-12
`purify_all_inuse (void)`	12-17
`purify_all_leaks (void)`	12-17
`purify_all_messages (void)`	12-20
`purify_assert_is_readable (const char *addr, int size)`	12-15
`purify_assert_is_writable (const char *addr, int size)`	12-15
`purify_clear_fds_inuse (void)`	12-12
`purify_clear_inuse (void)`	12-17
`purify_clear_leaks (void)`	12-17
`purify_clear_messages (void)`	12-20
`purify_describe (char *addr)`	12-15
`purify_exit (int status)`	12-11
`purify_get_pool_id (char  *mem)`	12-23
`purify_get_user_data (char  *mem)`	12-23
`purify_is_running (void)`	12-33
`purify_logfile_printf (char *fmt, ...)`	12-9
`purify_map_pool (int id,  void (*fn) (char *mem, int size, void *data))`	12-23
`purify_map_pool_id (void (*fn) (int id))`	12-23
`purify_name_thread (const char * name)`	12-28
`purify_new_fds_inuse (void)`	12-12
`purify_new_inuse (void)`	12-17
`purify_new_leaks (void)`	12-17
`purify_new_messages (void)`	12-20
`purify_printf (char *fmt, ...)`	12-9
`purify_printf_with_call_chain (char *fmt, ...)`	12-9
`purify_set_pool_id (char  *mem,  int id)`	12-23

Function	Page
`purify_set_user_data (char  *mem, void *data)`	12-23
`purify_start_batch (void)`	12-20
`purify_stop_batch (void)`	12-20
`purify_stop_here (void)`	12-33
`purify_stop_here_internal (void)`	12-33
`purify_watch (char *addr)`	12-29
`purify_watch_info (void)`	12-30
`purify_watch_n (char *addr, unsigned int size, char *type)`	12-30
`purify_watch_<num> (char *addr) <num>=1,2,4,8`	12-29
`purify_watch_r_<num> (char *addr) <num>=1,2,4,8`	12-29
`purify_watch_remove (int watchno)`	12-30
`purify_watch_remove_all (void)`	12-30
`purify_watch_rw_<num> (char *addr) <num>=1,2,4,8`	12-30
`purify_watch_w_<num> (char *addr) <num>=1,2,4,8`	12-29
`purify_what_colors (char *addr, unsigned int  size)`	12-15

Build-time options

Build-time options	Default
`-cache-dir`	`<purifyhome>/cache`

Sets the global directory where Purify caches instrumented versions of object files and libraries. See also, "Deleting cached object files" on page 6-19.

`-always-use-cache-dir`	**no**

Forces all Purify'd libraries and object files to be written to the global cache directory, even if they reside in writable directories.

`-forbidden-directories`	**system dependent**

Use this option to specify a colon-separated list of directories into which Purify cannot write files, even if the directories listed are writable. All the subdirectories of forbidden directories are also forbidden. The default values are:

Sun OS4   `/lib:/opt:/usr/lib:/usr/5lib:/usr/ucb/lib:/usr/lang:/usr/local`

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

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

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

`-auto-mount-prefix`	`/tmp_mnt`

Specifies the directory prefix used by the file system auto-mounter, usually `/tmp_mnt`, to mount remote file systems in NFS environments. Use this option to strip the prefix, if present, in order to improve the readability of source filenames in Purify reports.

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

If this option is not set correctly, Purify might be unable to access files on auto-mounted filesystems. The auto-mounter might not recognize their names.

`-force-rebuild`	**not set**

Forces your entire program to be reinstrumented (irrespective of whether object files and libraries have been updated since they were last instrumented). See `-static-checking-guardzone` option on page 12-24 for details of when this is useful.

Build-time options	Default
-linker	**system dependent**

Specifies the name of the linker that Purify should invoke to produce the executable. Use this option only if you need to bypass the default linker. The default linkers are:

`/bin/ld`

`/usr/ccs/bin/ld`

Purify does not support the `-linker` option on IRIX.

Note: Do not use this option to specify PureLink. For instructions on using PureLink with Purify, see page 1-8.

-g++	**no**

Purify sets this option automatically if you call the `g++` compiler. Purify knows that the `g++` compiler is being used and invokes special processing to avoid spurious ABR errors with the `delete` operator. It also sets the default demangling mode so that `g++` mangled function names are properly resolved.

-collector	**not set**

Specifies the name of the collect program to be used to sequence and collect static constructors in `C++` code. You must set this option to the name of the collect program used by the `g++` compiler.

To find the name of the collect program used by the `g++` compiler, use:

```
% g++ -v myprogram.c
```

For example, if the collect program is:

```
/usr/local/lib/gcc-lib/sun-sparc-sunos4/4.0/ld
```

use the command:

```
% purify -g++=yes \
-collector=/usr/local/lib/gcc-lib/sun-sparc-sunos4/4.0/ld \
g++ myprogram.c
```

Note: `g++` on Solaris 2 does not use a collector for `C++` programs. Purify on Solaris ignores this option.

Note: Purify does not support the `-collector` option on IRIX.

Build-time options	Default

`-help`

Prints a short help message about how to use the command line options.

`-ignore-runtime-environment`   **no**

Prevents the run-time Purify environment, including suppressions, from overriding the option values used in building the program.

This is useful if you are building a Purify'd program for someone else to run, and you want to make sure that the options and suppressions you specify are in effect at run time.

Use this option when you use the option `-mail-to-user`.

`-print-home-dir`

Prints the name of the directory where Purify is installed, then exits. For example, you can use this option to build the compiler command when including the `purify.h` file from the installation directory:

```
$CC -c $CFLAGS -I`purify -print-home-dir` myprogram.c
```

`-program-name`   **argv[0]**

Specifies the full pathname of the Purify'd program if `argv[0]` contains an undesirable or incorrect value. For example, when your program is invoked by an `exec` call whose path differs from the argument that it passes as `argv[0]` to your program. In such cases, Purify cannot find the program file and therefore cannot interpret addresses as function names.

You might need to use this option if you find little or no symbolic information in the messages from your Purify'd program.

`-usage`

Prints a short help message about how to use the command line options.

`-version`

Purify prints its version number string to `stdout` and then exits. For example, you can identify which version of Purify is in use while running a test suite by incorporating these lines in your test harness scripts:

```
#!/bin/sh
...
echo "Run monitored by Purify: `purify -version`"
...
```

Annotation options

Annotation options	Default
-copy-fd-output-to-logfile	**not set**

This option appends file descriptor output to the log file. Specify a list of file descriptors separated by commas. Purify copies output written to these file descriptors into the log file. This can help you reconstruct what the user did.

For example, to copy output written to stdout and stderr into the log file interspersed with Purify output, use:

```
% purify -copy-fd-output-to-logfile=1,2 cc myprogram.c
```

Annotation API

Note: Purify does not support the full `%` conversion-character syntax of `printf`. You can use the simple conversion characters `%d`, `%u`, `%n`, `%s`, `%c`, `%e`, `%f`, or `%g`. No field width or precision specifiers are allowed, and the `%e`, `%f`, `%g` characters are equivalent to `%10.2f`.

Annotation functions

int purify_printf (char *fmt, ...)

Prints formatted output from the program to the Viewer, stderr or log file if set.

int purify_printf_with_call_chain (char *fmt, ...)

Prints formatted output and the current call chain to the Viewer, stderr, or log file if set. For example:

```
if (detect_error) {
    purify_printf_with_call_chain(
        "Found bad input value %d\n", in_val);
}
```

This example displays the specified string and the function-call sequence to this point. This might help track errant function-call requests without stepping through the debugger. In this manner, the function purify_printf_with_call_chain extends the power of debugging using printf.

int purify_logfile_printf (char *fmt, ...)

Prints formatted output from the program to the log file if the -log-file option is set. If -log-file is not set, this function does nothing.

Exit processing options

Exit processing options

`-exit-status`   **no**

Enables you to control the exit status of your Purify'd program, based upon the
Purify results. If Purify detects unsuppressed access errors, leaks, or potential
leaks, the additional bits are OR'd into the exit status of the program.

Unsuppressed error	Bit OR'd in exit status
Memory access errors	0x40
Memory leaks	0x20
Potential memory leaks	0x10

`-run-at-exit`   **not set**

Specifies an arbitrary shell command to be run when your program exits or
otherwise terminates. In addition to the `%V`, `%v`, and `%p` conversion characters
described on page 11-2**,** Purify recognizes these conversion characters:

`%z`   String value "true" or "false" indicating whether any call chains were printed
       (for example, in error or leak reports)
`%x`   Program's exit status (0 if the program did not call exit)
`%e`   Number of distinct access errors printed
`%E`   Total number of errors printed
`%l`   Number of bytes of memory leaked
`%L`   Number of bytes of memory potentially leaked

For example, if you set the option:
```
setenv PURIFYOPTIONS '-run-at-exit="if %z ; then \
echo \"%v: %e errors, %l+%L bytes leaked.\" ; fi"'
```

When your program exits, you might see on `stdout`:
```
testprog: 2 errors, 1+10 bytes leaked.
```

See also:

- `-leaks-at-exit` **on page 12**-**16**
- `-inuse-at-exit` **on page 12**-**16**
- `-fds-inuse-at-exit` **on page 12**-**12**
- `-thread-report-at-exit` **on page 12**-**27**
- "**Running shell scripts at exit**" **on page 6**-**14**

Exit processing API

Exit processing functions

`int purify_exit (int status)`

This function behaves like the function `exit`, unless Purify detects any unsuppressed errors, leaks, or potential leaks, in which case it ORs special flag bits into the status value you supply. These are:

Unsuppressed error	Bit OR'ed in exit status
Memory access errors	0x40
Memory leaks	0x20
Potential memory leaks	0x10

You can replace the call to `exit(status)` or the `return status` in `main` with this function. If Purify is *not* running, `purify_exit` behaves like the regular `exit` function.

File descriptor options

File descriptor options	Default
`-fds`	`26`

Changes the default set of file descriptors used by Purify in case they clash with the ones used by your program. For example, to use file descriptors 57 and 58 instead of the default 26 and 27, use: `-fds=57`

`-fds-inuse-at-exit`	`yes`

Specifies whether file descriptors in use should be reported at program exit. Use `-fds-inuse-at-exit=no` to suppress printing file descriptor messages.

File descriptor API

File descriptor functions

`int purify_all_fds_inuse (void)`

Generates a list of all file descriptors currently open. Returns the number of currently open file descriptors.

`int purify_new_fds_inuse (void)`

Generates a list of new file descriptors found since the last call to a file descriptor API function. Returns the number of new file descriptors.

`int purify_clear_fds_inuse (void)`

Marks the file descriptors that have been opened since the last call to a file descriptor API function so that the function `purify_new_fds_inuse` does not report them. Returns the number of new file descriptors.

See also:

- `-copy-fd-output-to-logfile` on page 12-9
- Chapter 5, "Analyzing File Descriptors"

Mail mode option

Mail mode options	Default
`-mail-to-user`	**not set**

This option specifies the e-mail addresses to which the Purify reports are mailed if the program reports errors. For example, to send the reports to the user named "Chris" upon completion of the Purify'd application, use:

```
% purify -mail-to-user=chris gcc ...
```

Or specify a list of addresses separated by commas (,).

When mail mode is turned on, by default Purify does *not* send output to the Viewer or to `stderr` unless you specify the option `-windows=yes` or `-log-file=stderr`.

When you use `-mail-to-user`, you might also want to use the `-ignore-runtime-environment` option. See page 12-8.

See also:

- "Mailing Purify output to developers" on page 6-6

Memory access options

Memory access options	Default
`-free-queue-length`	`100`

Sets the number of entries in the free queue maintained by Purify. When you free a block, Purify stores it in an FIFO queue. When the queue length is exceeded, Purify frees the first block queued making it available for reuse. This helps Purify detect free-memory accesses (FMW, FMR). If you have plenty of swap space, you can use this option to increase the number of entries queued, thereby increasing the probability of detecting free-memory accesses.

`-free-queue-threshold`	`10000`

Sets the maximum size of freed blocks to be appended to the Purify free queue. Purify immediately frees blocks larger than this.

See also:

- "Memory access API" on page 12-15
- Chapter 3, "Memory Access Errors"

Memory access API

Memory access functions

---

int   purify_assert_is_readable (const char *addr, int size)

Simulates a read, generating any ABR, BSR, FMR, IPR, MSE, NPR, SBR, UMR, WPR or ZPR errors detected and calling `purify_stop_here`. Returns 0 if errors are detected, and returns 1 if no errors are detected.

---

int   purify_assert_is_writable (const char *addr, int size)

Simulates a write, generating any ABW, BSW, FMW, IPW, MSE, NPW, SBW, WPW, or ZPW errors detected and calling `purify_stop_here`. Returns 0 if errors are detected, and returns 1 if no errors are detected.

---

int   purify_describe (char *addr)

Prints specific details about the memory pointed to by `addr`, including its location (stack, heap, text) and, for heap memory, the call chains of its allocation and free history.

Returns the pointer passed to it.

---

int   purify_what_colors (char *addr, unsigned int size)

Prints out the memory state of `size` bytes starting at memory address `addr`. The memory state of each byte of memory is represented by one of the letters "R," "G," "B," or "Y," corresponding to the colors red, green, blue, and yellow respectively.

Unallocated, uninitialized memory is red. When it is allocated but not yet initialized, it is yellow. Once written or initialized, it is green. When freed, uninitialized memory turns from yellow to red, while initialized memory turns from green to blue.

```
(gdb) print purify_what_colors(buf, sizeof(buf))
color codes of 8 bytes at 0xefffee1c:GGGGYYYY
```

For more information about color of memory, see "How Purify finds memory access errors" on page 3-2.

See also:

- "Memory access options" on page 12-14
- Chapter 3, "Memory Access Errors"

Memory leak options

Memory leak options	Default
-inuse-at-exit	**no**

Specifies whether memory in use is reported at program exit. Memory in use is memory that has been allocated and to which there are still pointers.

Use `-inuse-at-exit=yes` to print in-use messages at exit.

-leaks-at-exit	**yes**

Specifies whether memory leaked is reported at program exit. Memory leaked is memory that has been allocated and to which there are no pointers.

Use `-leaks-at-exit=no` to suppress printing memory leak messages at exit.

-pointer-mask	**0xffffffff**

Specifies a mask that lets Purify extract the correct value from custom pointers.

Normally, if your application ORs flags into the upper bits of heap pointers, Purify cannot follow them to the memory blocks they refer to and might incorrectly report leaks. For example, if you use the upper 4 bits of pointers for flags, you should use:

```
-pointer-mask=0x0fffffff
```

-pointer-offset	**0**

Tells Purify the size of the extra memory allocated by a custom `malloc` wrapper.

If you use a `malloc` wrapper that allocates extra memory for bookkeeping purposes over and above the size requested and returns an adjusted pointer past the extra memory allocated, Purify might incorrectly report memory inuse as potential leaks (PLKs). This option ensures that Purify does not report false PLKs.

-search-mmaps	**no**

Purify automatically sets this option to yes if you use ObjectStore (OSCC).

Tells Purify to search for heap pointers in memory obtained from mmap. Use this option when using an object-oriented database such as ObjectStore, where mmap'd databases anchor many blocks of memory that would otherwise be reported as leaks.

See also:

- "Memory leak API" on page 12-17
- Chapter 4, "Memory Leaks"
- Chapter 9, "Custom Memory Managers"

Memory leak API

Memory leak functions

`int purify_all_inuse (void)`

Prints a summary message on all heap memory currently allocated.

`int purify_all_leaks (void)`

Prints a summary message on all current memory leaks.

`int purify_new_inuse (void)`

Prints an incremental message on all new heap memory allocations. This is memory allocated since the last call to the functions `purify_new_inuse` or `purify_clear_inuse`.

`int purify_new_leaks (void)`

Prints an incremental message on all new leaks, that is, leaks introduced since the last call to the functions `purify_new_leaks` or `purify_clear_leaks`.

All Purify leak-detection functions return the totals of the unsuppressed leaks or memory in use. This simplifies usage both programmatically and in the debugger. For example, from the debugger you can use:

`(gdb) break event_loop if (purify_new_leaks())`

Or from your program, you can use:

```
event_loop(){
  while(1){
      do stuff
      if (purify_new_leaks()) {
          purify_stop_here_internal();
      }
  }
}
```

`int purify_clear_inuse (void)`

Finds memory in use and notes it as found so that future calls to the function `purify_new_inuse` do not show the memory in use. This function does not print a message.

`int purify_clear_leaks (void)`

Finds leaks and marks them cleared so that the function `purify_new_leaks` does not report them. This function does not print a message. It is useful for ignoring all leaks from a certain portion of code, such as a start-up sequence.

Message appearance options

Message appearance options	Default
`-chain-length`	`6`

Specifies the number of stack frames to be recorded and printed in the function call chain in a Purify message. This option also affects the extent of the red zone around memory blocks used by Purify to detect array-bound errors. Since Purify stores the call chain of the function allocating the memory in the red zone at the ends of the block, you should set this to a larger value to increase the red zone and to increase the extent of the area where incorrect array boundary accesses can be detected. This also changes the memory and swap space used by the program.

`-show-directory`	`no`

Shows the directory path for each file in the call chain if that information is available. This information might not be available if the file is not compiled with the compiler debugging option `-g`. The directory, if displayed, is stripped of the auto-mounter prefix. The format of the call chain looks similar to:

```
func1[/home/myprogram/file.o]
func2[/home/myprogram/file.o]
```

`-show-pc`	`no`

Facilitates debugging by showing the full program counter (pc) value in each frame of the call chain. The format of the call chain is similar to:

```
func1[file.o pc=0x5678]
func2[file.o pc=0xd2e0]
```

`-show-pc-offset`	`no`

Facilitates debugging by appending a pc-offset from the start of the function to each function name in the call chain. The format of the call chain looks similar to:

```
func1+0x1234[file.o]
func2+0x4000[file.o]
```

See also:

- `-leaks-at-exit` on page 12-16
- `-inuse-at-exit` on page 12-16
- `-thread-report-at-exit` on page 12-27
- `-fds-inuse-at-exit` on page 12-12

Message batching options

Message batching options	Default
-messages	**first**

Controls how Purify handles repeated messages that have the same call chain.

-messages=first: Purify displays *only* the first occurrence of each repeated message. If the same message is generated again and the output is being sent to the Purify Viewer, the message is not displayed but the repeat count on the first message is updated. Purify discards repeated messages when saving output to a log file.

-messages=batch: Purify batches all error messages and displays them along with repeat counts when the program exits. This mode is useful when you send output to a log file or mail-mode report and you want to note the number of occurrences of each message.

-messages=all: Purify displays each error message in the order generated. This is useful for some types of interactive debugging, for example, where you need to correlate repeated occurrences of errors with other program actions.

See also:

- "Message batching API" on page 12-20
- "Controlling message batching" on page 6-9

Message batching API

Message batching functions

`int purify_start_batch (void)`

Enables batch mode, if not already set. Batch mode postpones error reporting and consolidates identical messages until batch mode is turned off, or the program exits.

The summarized batch message includes the number of occurrences of each error, the function call chain, and other details of the first occurrence of the error.

`int purify_stop_batch (void)`

Disables batch mode. Prints all new messages in the batch and resumes automatic and immediate reporting.

`int purify_new_messages (void)`

Prints new messages consolidated in the batch since the last call to `purify_all_messages`, `purify_new_messages`, or `purify_clear_messages`.

`int purify_clear_messages (void)`

Marks new messages in the batch so that `purify_new_messages` does not print them.

`int purify_all_messages (void)`

Prints all messages in the batch.

See also:

- "Message batching options" on page 12-19
- "Controlling message batching" on page 6-9

Output mode options

Output mode options	Default
-windows	**not set**

Use to control whether Purify opens the Viewer.

If this option is not specified, Purify opens the Viewer unless -log-file or -mail-to-user is set.

-windows=yes: Purify opens the Viewer, in addition to any other output formats specifically requested.

-windows=no: Purify does not open the Viewer, but outputs ASCII text to stderr, unless -log-file, -view-file or -mail-to-user is set.

-log-file	**not set**

If this option is not specified, Purify opens the Viewer unless -windows=no or -view-file is set.

-log-file=stderr: Purify outputs ASCII text to the program's stderr stream, in addition to any other output formats specifically requested.

-log-file=<filename>: Purify saves ASCII output to the named file, in addition to any other output formats specifically requested.

You can use conversion characters in <filename>. See "Using conversion characters in filenames" on page 11-2.

-append-logfile	**no**

Appends Purify output to the current log file rather than replacing it.

-view	**not set**

To open an empty Viewer, use
% purify -view <program-name>

To open an empty Viewer on a different screen, use:
% purify -view -display=<myscreen>.0 <program-name>

To open a view file in the Viewer, use:
% purify -view <program-name>.pv

Purify opens the specified file in the Viewer, displaying the same output as when you ran the Purify'd program interactively. You do not need access to the original program.

Output mode options	Default
`-view-file`	**not set**

If this option is not set, Purify opens the Viewer unless `-windows=no`, `-log-file`, or `-mail-to-user` is set.

`-view-file=<filename>`: Purify writes compact binary data to the specified file, in addition to any other output formats specifically requested.
Use `purify -view <filename>` to view the resulting file.

You can use conversion characters in `<filename>`. See "Using conversion characters in filenames" on page 11-2.

`-output-limit`	**1000000**

Use with the option `-log-file` to restrict the size of the log file and to conserve disk space. The value of this option specifies the maximum size of the Purify message in bytes. Purify truncates all output beyond this size.

`-user-path`	**not set**

Specifies a list of directories in which to search for programs and source code. You can specify full pathnames separated by spaces or colons (:). For example:

```
setenv PURIFYOPTIONS -user-path=/usr/home/prog1:/usr/home/prog2
```

Purify searches for the executable file from which to read the symbol table in your `$PATH`, then in `-user-path`. See also `-program-name`.

When searching for source code, Purify looks for the file in the full pathname specified in the debugging data, then in directories listed in `-user-path`, and finally in the current directory.

See also:

- "Controlling Purify output" on page 6-2

Pool allocation API

Pool allocation functions

`void  purify_set_pool_id  (char  *mem,  int id)`

Sets the pool id for the specified memory `mem` to `id`.

`int  purify_get_pool_id  (char  *mem)`

Returns the pool id associated with the pool of memory `mem`.

`void  purify_set_user_data  (char  *mem, void *data)`

Sets the auxiliary user data associated with the pool of memory `mem` to `data`.

`void*  purify_get_user_data  (char  *mem)`

Returns a pointer to the auxiliary user data associated with the pool of memory `mem`.

`void  purify_map_pool`
`(int id,  void (*fn) (char *mem, int size, void *data))`

Applies the function `fn` to all members of the pool of memory identified by pool-id `id.` The arguments to the function `fn` are: the memory pointer (`mem`), the size of the memory(`size`), and the auxiliary user data associated with the pool (`data`).

`void  purify_map_pool_id  (void (*fn) (int id))`

Applies the function `fn` to each pool id known to the system.

See also:

- "Modifying pool allocators" on page 9-5

Static checking options

Static checking options	Default
`-static-checking`	`yes`

Enables instrumentation for array bounds checking of static data.

In the default mode, `-static-checking=yes`, Purify looks first in the `.purify` directive files for static-checking directives for a given object file, then it checks the command-line options. If Purify does not find any static checking options, it defaults to instrumenting in `safe` mode with a guard zone of 16 bytes.

If you specify `-static-checking=no`, static checking is completely disabled, regardless of whether you specify Purify directives for a given object file.

`-static-checking-guardzone`	16

Sets the size of the guard zone inserted between variables in the data section. The default value is 16 bytes; however, for arrays of large structures, this value might be too small to catch a reference beyond the last element.

To set the guard zone for specific files, add the following directive to a `.purify` file:

`static_checking_guardzone <integer value> <filename>`

You can include wildcard characters in `<filename>`. For example: `program*.o` matches `/dira/dirb/program.o`, `/dira/dirb/program1.o`, and `program4`.o.

For more information about how to specify directives in a `.purify` file, see page 7-4.

Note: The size of the guard zone affects the way a file is instrumented. To change the guard zone size of a file that is already instrumented, you need to relink your program to cause it to be reinstrumented with the new guard zone size. To cause your entire program to be reinstrumented with the new guard zone size, use the `-force-rebuild` option. See page 12-6.

Static checking options	Default
`-static-checking-default`	`safe`

Controls the default behavior of static checking in the absence of specific entries in directive files.

If `never` is specified, static checking is disabled.

If `minimal` is specified, Purify inserts guard zones only at the beginning and end of the data section for a given object file.

If the default `safe` is specified, Purify inserts guard zones only between data variables if there are no data section relative relocations. If an object file contains a data section relative relocation, Purify instruments that file in `minimal` mode.

If `aggressive` is specified, Purify inserts guard zones between data variables even if it finds data section relative relocations. However, the relocations must be to addresses that correspond to a known data variable.

To control the behavior of static checking for specific object files, add the following directive to a `.purify` file:

`static_checking_default [never|minimal|safe|aggressive] <filename>`

You can include wildcard characters in `<filename>`.

For more information about how to specify directives in a `.purify` file, see page 7-4.

See also:

- "How Purify checks statically allocated memory" on page 3-4

Suppression options

Suppression options	Default
`-suppression-filenames`	**system dependent**

Specifies the directive files for specific programs or specific operating systems. The default suppression files are:

.purify,.purify.sunos4

.purify,.purify.solaris2

.purify,.purify.hpux

.purify,.purify.irix

See also:

- "Static checking options" on page 12-24
- "Using the -suppression-filenames option" on page 7-10

Threads options

Threads options	Default
`-threads`	`no`

Enables thread support. Purify enables this automatically when your program is linked with a supported thread package.

Use `-threads=no` to disable thread support.

`-max-threads`	`20`

Specifies the maximum number of threads in a program. If you expect to use more than 20 threads, set this option to avoid overflowing the tables that record thread data.

`-thread-report-at-exit`	`no`

Specifies whether or not the threads summary is printed when the program exits. Use `-thread-report-at-exit=no` to suppress printing a thread summary at exit.

Purify enables this automatically when your program is linked with a supported thread package.

`-thread-stack-change`	`0x1000`

Specifies the minimum size of a change to the stack pointer that signals a thread context switch.

Programs that allocate large data structures on the stack might need to increase this value. Programs that create threads whose stacks are very close to one another might need to decrease it.

See also:

- "Threads API" on page 12-28
- "Customizing the thread summary message" on page 6-10

Threads API

Threads functions

int pure_name_thread(const char * name)

Associates the specified name with the id of the current thread. Returns 0.

Purify uses this name in all messages that mention this thread. For example:

```
UMR: Uninitialized memory read
   This is occurring while in thread 6 "Consumer":
        consumer_loop  [test.c:213]
        do_consumer_loop [test.c:236]
        writer_to_stdio [test.c:244]
        _thread_start  [libthread.so.1]
Reading 4 bytes from 0xeee03d5c on the stack of
             thread 5 "Producer".
```

See also:

- "Threads options" on page 12-27
- "Customizing the thread summary message" on page 6-10

Watchpoint options

Watchpoint options	Default
`-watchpoints-file`	`./<program-name>.watchpoints`

Specifies the filename where Purify saves watchpoint settings. To disable saving watchpoints in a file, set this option to an empty filename, using:

```
% setenv PURIFYOPTIONS -watchpoints-file=
```

You can use conversion characters in the filename. See "Using conversion characters in filenames" on page 11-2.

Watchpoint API

Watchpoint functions

```
int   purify_watch (char *addr)
int   purify_watch_<num> (char *addr) <num>=1,2,4,8
int   purify_watch_w_<num> (char *addr) <num>=1,2,4,8
```

These functions specify watchpoints that detect writes, allocations, frees, and entry and exit of memory addresses. The `purify_watch_<num>` and `purify_watch_w_<num>` functions are identical. You can specify watchpoints for variables of 1, 2, 4, and 8 bytes. The `purify_watch` function watches 4 bytes.

```
(gdb)     print purify_watch_1(&my_char)
(dbx)     print purify_watch_w_8(my_double_ptr)
(xdb)     p purify_watch_1(&my_char)
```

These functions return the number assigned to the watchpoint.

```
int   purify_watch_r_<num> (char *addr) <num>=1,2,4,8
```

Specifies a watchpoint that detects reads, allocations, frees, and entry and exit of memory addresses. Use to specify watchpoints for variables of 1, 2, 4, and 8 bytes.

```
(gdb) print purify_watch_r_1(&read_only_char)
```

This function returns the number assigned to the watchpoint.

Watchpoint functions

`int   purify_watch_rw_<num> (char *addr) <num>=1,2,4,8`

Specifies a watchpoint that detects reads, writes, allocations, frees, and entry and exit of memory addresses. `<num>`=1, 2, 4, or 8. You can specify watchpoints for variables of 1, 2, 4, and 8 bytes.

```
(gdb) print purify_watch_rw_1(&rw_char)
```

This function returns the number assigned to the watchpoint.

`int   purify_watch_n (char  *addr, unsigned int size, char *type)`

Sets a watchpoint on an arbitrary-sized buffer.
`addr` specifies the address of the beginning of the buffer.
`size` specifies the number of bytes to watch.
`type` specifies whether to watch for writes (`"w"`), reads (`"r"`), or both (`"rw"`).

```
(gdb) print purify_watch_n(buf, sizeof(buf), "rw")
(dbx) print purify_watch_n(write_only_buf,100,"w")
```

This function returns the number assigned to the watchpoint.

For interactive use, it is sometimes easier to call `purify_watch_n` with `type = 1, 2, or 3` instead of `r`, `w`, or `rw` respectively.

`int   purify_watch_info (void)`

Lists all the active watchpoints and returns `0`.

`int   purify_watch_remove (int watchno)`
`int   purify_watch_remove_all (void)`

The function `purify_watch_remove` removes the watchpoint specified by `watchno`. The function `purify_watch_remove_all` removes all watchpoints. Both functions return `0`.

See also:

▪ **Chapter 8, "Setting Watchpoints"**

Miscellaneous options

Miscellaneous options	Default
`-follow-child-processes`	**no**

Controls whether Purify monitors child processes created when a Purify'd program forks. If you do not specify `-follow-child-processes`, Purify does not follow child processes.

If this option is set to `yes`, Purify invokes a new Viewer (if appropriate) and monitors the progress of the child process separately from the parent, reporting access errors and memory leaks.

`-freeze-on-error`	**no**

If this option is set, when an error is reported to the Purify Viewer it delays sending the response back to the application, which usually causes the application to freeze. This is useful when you want to explore the relationship between error reports and program activity.

While the application is frozen, two new buttons appear on the message display. Press **Continue** to continue to the next error message. Press **Reset freeze-on-error then Continue** to continue uninterrupted.

Note: Do *not* use this feature with a debugger. Instead, set a debugger breakpoint in the function `purify_stop_here`. When the application is frozen by the Viewer, the debugger freezes also, so you cannot examine variables or obtain stack traces.

`-jit-debug`	**not set**

Enables just-in-time debugging, instructing Purify to automatically start a debugger when it reports a message of the type you specify. You can use your debugger to investigate errors even when you run your application from outside the debugger.

Specify a list of keywords separated by commas. For example:

```
-jit-debug="error, warning, ask, watchpoint"
```

`ask`	Purify asks you if you want to start the debugger when it encounters the specified type of message
`error`	Purify starts a debugger for fatal or corrupting messages
`warning`	Purify starts a debugger for warning messages
`watchpoint`	Purify starts a debugger for watchpoint messages

For a description of message severities, see "Message severity" on page 10-2.

Miscellaneous options	Default

`-jit-debug` option continued:

You can change the list of available debuggers, and Purify's interface to them, using your ~/.purify.Xdefaults file and the pure_jit_debug script which is located in the Purify installation directory. See "Customizing Purify scripts" on page 6-16.

JIT debugging can also be enabled from the Viewer. See "Enabling JIT debugging" on page 6-11.

`-handle-signals`	**not set**
`-ignore-signals`	**not set**

Purify installs a signal handler for many of the possible software signals that can be delivered to a Purify'd process. The signal handler outputs a SIG or COR message to the Viewer or log file before passing control to the user or to the default signal handler. The initial default signals handled by Purify are:

Sun OS4

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

Solaris

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

HPUX

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

IRIX

```
SIGHUP, SIGINT, SIGQUIT, SIGILL, SIGABRT, SIGEMT, SIGFPE,
SIGBUS, SIGCPU, SIGSEGV, SIGSYS, SIGPIPE, SIGALRM, SIGTERM,
SIGUSR1, SIGUSR2, SIGTSTP, SIGTTIN, SIGTTOU, SIGVTALRM,
SIGPROF, SIGXCPU, SIGXFSZ
```

To ignore signals in this list, set `-ignore-signals` to a comma-delimited list of the signals to be ignored. For example: `-ignore-signals=SIGSEGV,SIGBUS`

To handle additional signals, set `-handle-signals` to a comma-delimited list of the additional signals. For example: `-handle-signals=SIGALRM,SIGCHLD`

Miscellaneous options	Default

`-handle-signals`, `-ignore-signals` continued:

Purify does *not* handle SIGKILL, SIGSTOP, or SIGTRAP signals, since doing so interferes with normal program operation. If you specify these signals in `-handle-signals`, Purify silently ignores them.

Note: The default action on delivery of SIGALRM terminates the process. Purify does not handle this signal by default, since it is used internally by functions such as `sleep`. However, if you see a process terminated with a message like "Alarm clock," you can set `-handle-signals=SIGALRM` to get a report when the program terminates. You can also add the following suppression directive to a `.purify` file to silence the signal message when used in the sleep function:

```
suppress SIG sleep
```

See "Specifying suppressions in a .purify file" on page 7-4.

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

Miscellaneous API

Miscellaneous functions

int   purify_is_running (void)

Returns 1 if the executable is Purify'd, 0 otherwise. You can use this function to enclose special purpose application code to execute in the Purify'd environment. For example:

```
if (purify_is_running()) {
    install_gui_leaks_button();
}
```

int purify_stop_here (void)

Sets a breakpoint on `purify_stop_here` to cause your debugger to stop on every Purify error message just before the error actually occurs. Do not call `purify_stop_here` directly from your program. Instead, set a breakpoint on it.

int purify_stop_here_internal (void)

Manually triggers a breakpoint you've set on `purify_stop_here`.

13

Common Questions

This chapter contains answers to common questions about:

- Building Purify'd programs, this page
- Running Purify'd programs, page 13-5
- General questions, page 13-9

Questions about building Purify'd programs

How much swap space does a Purify'd program use at build time?

At build time, a Purify'd program uses swap space equal to approximately two to ten times the size of the program's largest uninstrumented object file or library.

Can I Purify just part of my program?

No. Purify needs to keep track of the state of the entire program's memory as all of your program modifies it. If you tried to Purify only a part of your program, initialization in the non-Purify'd portion would not be noted, causing many spurious Purify reports.

Can I tell Purify to ignore certain libraries or object files? I don't care about errors in them.

No. However, while you cannot tell Purify to skip checking in code that you are not working on, you can suppress error reports from this code. Note that errors that show up in library functions are often caused by your program's misuse of those functions, or misinterpretation of the function's programming interface. See Chapter 7, "Suppressing Purify Messages," for more information.

Can I make Purify put everything in the cache when my project directory is full?

Yes. Use:

```
% setenv PURIFYOPTIONS -always-use-cache-dir=yes
```

Can I delete all those <filename>_pure_*.o files?

Yes. See "Deleting cached object files" on page 6-19.

Why do I get this ld warning?

```
ld: /usr/purify/cache/lib/libc_pure.300.sa.1.7: warning: table
of contents for archive is out of date; rerun ranlib(1)
```

Your workstation's clock is out of sync with respect to your file server. To get them back in sync, become root and use the command:

```
# rdate <file-server>
```

Why do I get this message?

```
malloc failed with request for 12345678 bytes?
```

You have run out of swap space. You can build on a machine that has more swap space, or add additional swap space. Purify provides instructions in the full text of the message. On SunOS 4, the message might look like the one below; however, a similar message appears on other platforms.

```
malloc failed with request for 12345678 bytes. Your machine is
out of swap space. Use '/usr/etc/pstat -s' to see how much swap
is available. You can increase available swap space by quitting
other programs, or by using 'swapon' and 'mkfile' (see their man
pages). Note that the first time you Purify your application,
Purify needs to build all the libraries, and will use more swap
space than it will subsequently.
```

For SunOS 4, use the command /usr/sbin/swap -a or /usr/etc/pstat -s to see how much memory you have. Use swapon and mkfile to add swap space.

Solaris For Solaris, use the command `/usr/sbin/swap -a` or `/usr/sbin/swap -s` to see how much memory you have. Use `mkfile` to add swap space.

HPUX For HP-UX, use the command `/etc/swapinfo` to see how much memory you have. Use `swapon` to add swap space.

IRIX For IRIX, use the command `/sbin/swap -s` to see how much memory you have. Use `mkfile` to add swap space.

For example, when you use `/usr/etc/pstat -s` on SunOS 4, you might get:

```
24592k allocated + 7472k reserved = 32064k used,
11692k available
```

In this case, the current swap space totals `32064 + 11692K` ≈ 44M. You need to add an additional 13 megabytes for a total of 57 megabytes of swap space.

Why do I get a linker error using Purify, while without Purify there is no linker error?

If you are using `/bin/ld` and your link line is long it might return with a message like:

```
ld: libfoo.a: No such file or directory
```

Purify might increase the link line sufficiently to expose this shortcoming in `/bin/ld`. You should be able to work around the problem with:

```
% unsetenv LD_LIBRARY_PATH
```

or by linking statically (using the `-Bstatic` option):

```
% purify cc -Bstatic -o prog prog.c
```

What should I do if I get this message?

```
ld: libw_ui_pure_300.a: warning: archive has no table of
contents?
```

Run the `ranlib` program on the library. This happens if the `ranlib` command issued by Purify failed, for example due to a lack of swap space.

What should I do if I get this message?

```
ld.so can't find file -lc_pure_NNNN
(or another filename).
```

You might have a dynamic linker deficiency. Try removing the directory within the Purify cache that contains the library that `ld.so` complains it cannot find. Then, rebuild with Purify to rebuild the shared directory and the shared libraries.

What should I do if I get this message?

```
ranlib: warning: libutil.a(util.o): no symbol table
```

You can ignore this message. It indicates that the object file `util.o` is empty. The source file probably contained `#ifdefs` that were all false.

Why do I get several warning messages?

```
While processing file
```

```
/dir/libfoo.a: Warning: Reloc of type 6 at 0x330 references
unknown segment 5. Ignored.
```

These messages are followed by the message:

```
PureLink1.1: Bad File: bad symbol (unknown type 0x5) in
modulell_pure_210.a
```

There is an incompatibility with GNU's assembler (GAS). To verify which assembler is being used, pass the option `-v` to the compiler. Then pass the `-v` to the specific assembler, for example:

```
% /bin/gnu/tools/as -v
```

To fix the problem use `/bin/as` instead. You can create a symbolic link if necessary.

What should I do if I get the warning?

```
ranlib: can't create __.SYMDEF: Permission denied?
```

You are trying to write to the current directory that does not have write permissions. You should change to a writable directory or modify the permissions of the current directory.

Questions about running Purify'd programs

How much swap space does a Purify'd program use at run time?

At run time, a Purify'd program uses approximately 1.5 times the swap space required by the non-instrumented program. Purify also uses swap space for an error processing process it creates.

How does a Purify'd program perform at run time?

On average, your Purify'd program will run two to five times slower than your non-Purify'd program. The exact speed depends on how many errors Purify finds in your program, how your program uses memory, and the size of your program's virtual memory relative to your machine's real memory (RAM).

If your Purify'd program runs more than five times slower, it might be thrashing, that is, spending an excessive amount of time paging to disk. Use the `vmstat` command to see how much time your program is spending in kernel mode as opposed to user mode.

Typically, a program should be in user mode more than 80 percent of the time. If user mode drops to less than 50 percent, run your program on a machine with more real memory, try to increase the locality of your program's memory references to reduce paging, or increase the real memory of your machine. You can also reduce the call chain length that Purify uses. See the `-chain-length` option on page 12-18.

I am getting a lot of UMR errors from Purify. Can I suppress errors?

Yes. `<purifyhome>/.purify` is the default suppressions file. See Chapter 7, "Suppressing Purify messages" for more information.

My program runs fine without Purify. What should I do when my Purify'd program exits prematurely and I get:

```
system error 24 - too many open files
```

By default, only 64 file descriptors can be in use at once by a program. When more than 64 files are used you will get the system error 24. Purify adds 2 file descriptors to your program, and if that exceeds the 64 file limit, your program will get this error.

You can increase the number of file descriptors allowed to work around this problem. This is a kernel modification that can be done by your system administrator. You can also use the `limit` command to add more file descriptors without having to change the kernel.

How do I tell which stack variable is uninitialized from this message?

```
Reading 4 uninitialized bytes from 0xff7ffaac
```

Look at the offending source line, and use a debugger to print the addresses of the variables used. Find the variable whose address matches the address in the report. Purify will always print the name of the variable if it can.

Why does Purify report a memory leak in this example?

```
int main(){
      int foo = malloc(10);
}
```

Since your program ends without calling `exit`, it returns to its caller, function `start`. When it returns, variable `foo` goes out of scope, causing a memory leak. The same thing happens if your program ends by calling `return`.

If you add a call to `exit` to the end of this program, no leak will be reported. When `main` calls `exit`, `foo` is still in scope and anchors the 10-byte memory block.

Why are the line numbers listed by Purify occasionally a line or two off?

Different compilers build their debugging information regarding program source code in different ways. Purify uses whatever information is provided to indicate line numbers when necessary. You can sometimes see similar behavior in debuggers as well; it is not a bug in Purify. Some C++ compilers tend to put wrong line-numbers in the debugging information in the code.

What should I do if I get this message?

```
ld.so: text enable failed
```

You need to use a machine with more swap space, close down other programs, or add additional swap space. See your system administrator.

For the commands to add swap space, see the answer to "Why do I get this message?" on page 13-2.

Why do my suppression directives not work?

If an item in the suppression database has a call chain with more function names in it than are recorded in the reports it is comparing against, an exact match is not possible. In such cases, the reports are not suppressed and a warning is issued for the first such instance.

You can control the number of functions in a call chain reported by Purify with the option `-chain-length`. Set this option to a larger number to ensure your suppressions are recognized.

Can I use my own malloc, or does Purify have its own?

You can use your own `malloc`. Purify intercepts calls to `malloc` but does not implement them. Your `malloc` package must implement

the standard `malloc` interface. See Chapter 8, "Setting Watchpoints," for more information.

How do I get a timestamp/user-name/etc. into the log file?

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/param.h>
#include "purify.h"

int
main(int argc, char **argv)
{
/* Add environment information to Purify logfile. */
if (purify_is_running()) {
    int i;
    time_t now;
    char cwd[MAXPATHLEN];
    char *user;

    /* Print timestamp. */
    time(&now);
    purify_printf("Run at: %s", ctime(&now));  /* ctime adds '\n' */

    /* Print user name. */
    user = getenv("USER");
    purify_printf("Run by: %s (%d)\n",
            user? user : "<$USER not set>",
            getuid());

    /* Print working directory */
#ifdef SUNOS4
    if (getwd(cwd) != NULL)
#else
    if (getcwd(cwd, sizeof(cwd)) != NULL)
#endif
    {
      purify_printf("Current directory: %s\n", cwd);
    }
}
...
}
```

Call `purify_printf` to include timestamp and user information

General questions

Why does this program show the UMR error on line 6 instead of line 5?

```
1      int main {}
2          int worse = 0;
3          int bad;
4
5          worse = bad;
6          worse++;
7          exit(0);
8      }
```

The variable `bad` is uninitialized. The uninitialized random value is copied from `bad` to `worse`. Then `worse`, which now contains the random uninitialized value, is incremented. Purify reports that uninitialized memory is used on line 6, but by default does not report that it is copied on line 5.

Many programs copy uninitialized values, but do not use them. An example is a program that uses `bcopy` to copy structures that contain padding due to data alignment restrictions. Since the program does not use these values it is not in error. Only when an uninitialized value is used in a computation or passed to a function does Purify signal that an error has occurred. For more details, see "A UMR example" on page 3-11.

Why does the leak show up on a line that seems unrelated to the memory block when I step through my program calling purify_new_leaks?

A leak occurs when the last reference to the memory is overwritten. After your program is done with a pointer, it is not always immediately overwritten. The pointer variable can be "dead" with respect to your source code, while the memory location or register containing it stays around for a while. This is why `purify_new_leaks` sometimes shows a leak occurring a few lines after the pointer variable becomes 'dead'. Of course, this does not affect the `malloc` location listed in the Purify report.

Does Purify detect array bounds errors for static and stack variables?

Purify now detects array bounds read and write errors on variables in statically-allocated memory. On SPARC systems, Purify also detects accesses across stack frames. See page 10-26 and page 10-27 for details on the SBR and SBW messages.

Does Purify support shared libraries?

Yes, including the use of `dlopen` on SunOS 4.1, Solaris 2, and IRIX, and `shl_load` on HP-UX. If you dynamically open a library that Purify has not seen before, or if you delete a cached Purify'd shared library, Purify will build it, if required at run-time.

Does Purify work with shared memory?

Yes.

What does "below the frame pointer" mean?

Automatic local variables in a function are stored on the stack, which also contains space to store register values and other function instance data. On the SPARC architecture, in most functions, two registers point to the top and bottom of the stack frame, or section of the stack used to hold that function's data. These are the frame pointer which points to the high-address end of the stack frame, and the stack pointer which points to the low-address end of the stack frame.

Where suitable symbolic data is available, Purify translates addresses on the stack into local variable names. However, if debugging data is not available, Purify will tell you the offset below the frame pointer, or above the stack pointer where an address lies.

Typically, local variables are allocated at addresses below the frame pointer, with the first named variable in the function at the highest address, and so on. If the code has been optimized, some local variables will be in registers and not on the stack. The

distance below the frame pointer might give some hint as to which variable is being referenced.

Registers can be spilled into a special section at the bottom 64 bytes of the stack frame, typically by the operating system. If you see references to addresses in this region, it's typically due to a wild pointer happening to point to such a region.

The stack grows downwards on the SPARC architecture, so as one function calls the next, the caller stack pointer becomes the called frame pointer, and the stack pointer is set to a new lower address.

Why does my application run fine without Purify but core dump when I use Purify? I get this message:

```
Purify (cor): Received signal 11 SIGSEGV (segmentation
violation):
```

Purify tends to magnify the existence of a fatal problem and as a result core dumps. Although the application doesn't normally core dump, this type of problem is very likely to core dump in the field, on a different system, or even sporadically on the current system.

In most cases the core dump is a result of a fatal error detected by Purify, for example NPR, NPW, ZPR, ZPW. The fatal error is usually reported just before the core message. Fixing this fatal error will fix the core dump.

When using Purify, I get this message:

```
Purifying: libgcc.a ...... execlp("ar", ...) failure in
add_symbol_table: No such file or directory.
```

This error occurs if you are using the GNU binutils version of `ar` because it doesn't put the symbol table in the library. To avoid this error use `/usr/ccs/bin/ar`. Specify `/usr/ccs/bin` as the first entry of the `PATH` variable. If you need to use the GNU version of `ar`, run binutils' `ranlib` on the library.

What are those funny function names like 'ReAd' and 'MaLlOc'?

Purify intercepts a number of functions like `read` and `malloc`. It changes the name of real definitions to be mixed-case, and provides wrapper definitions with the normal names, that do the checking and then call the real definitions.

What is the .pure file? Is it safe to remove it?

Purify uses the `.pure` file as part of its file-locking mechanism when creating Purify'd object files and libraries. The file will be created as needed, so you can safely delete it. However, since the file is zero-length, it does not impact disk space.

Using the Purify Viewer

To build a Purify'd program: `% purify cc -g <filename>.o`
Purify opens the Viewer by default when you run a Purify'd program: `% a.out`
To also open a saved view file (`.pv` file) in a Viewer:
`% setenv PURIFYOPTIONS '-view-file=./%v.pv'; a.out; purify -view ./a.out.pv`

Next	Previous	Expand	Collapse	Edit	Suppress	Explain	New Leaks	PureCoverage	PureDDTS

To display, select **Toolbar** from the View menu

Click to expand or collapse a message

Click to open an editor

Source code insert showing exact location of the error

```
Purify: a.out

File    View    Actions    Options                          Help

Finished  a.out                    (  1 error,   12 leaked bytes)
  Purify instrumented a.out (pid 2615 at Thu Jul 18 20:33:04 1996)
  ABR: Array bounds read
  This is occurring while in:
      _doprnt        [libc.so.1.9]
      printf         [libc.so.1.9]
      main           [hello_world.c:15]
        main()
        {
            char *mystr = malloc(strlen(helloWorld)
        );

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

      start          [crt0.o]
  Reading 1 byte from 0x4414c in the heap.
  Address 0x4414c is 1 byte past end of a malloc'd block at 0x44140 of 12
  This block was allocated from:
      malloc         [rtlib.o]
      main           [hello_world.c:12]
      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.

 Make …    Run …    Debug …    Kill …    Edit …
```

To display, select **Program controls** from the View menu

Running a make-run-debug-edit cycle

You can run an entire debugging cycle from the Viewer using the program controls: start a make, run an executable, or launch the debugger or editor.

Keyboard accelerators

Key	Action	Menu equivalent
Control-n, or Down arrow	Move to the next message in the outline hierarchy	Next in the Actions menu
Control-p, or Up arrow	Move to the previous message in the outline hierarchy	Previous in the Actions menu
Return	Expand the selected message	Expand in the Actions menu
DEL	Collapse the selected message	Collapse in the Actions menu
Space	Expand if selected message is currently collapsed; Collapse if selected message is expanded	

Purify 4.0 Quick Reference

Purify messages

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

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

Suppressing messages

Message suppression using a `.purify` file

Suppressing messages from the Viewer: Click the message, then select **Suppress** from the Options menu. This suppresses messages for the current session. To make the suppression permanent, click **Make permanent** or add the directive shown at the bottom of the suppression dialog to a `.purify` file in one of these standard directories:
- The program directory, to suppress messages from programs in that directory
- Your home directory, to suppress messages from all programs that you run
- The `<purifyhome>` directory, to suppress messages from all programs run by all users at your site

You can also use the `-suppression-filenames` option to specify the filenames of your choice.

Message suppression directive syntax and examples

Suppression syntax in a .purify file: `suppress <message-type> <function-call-chain>`
For `<message-type>`, specify the acronym for the message to be suppressed, wildcard "*" is permitted.
For `<function-call-chain>`, specify a semi-colon delimited chain of call-site specifications each of which may be either a function name or a filename (enclosed in double quotes). Wildcards "*" and "?" are permitted. "..." matches any series of functions.

For example:
- To suppress UMRs from the function `sqrt` add: `suppress umr sqrt`
- To suppress ABRs in any method of class `color` with prefix `test` add: `suppress abr color::test*`
- To suppress all messages from the static and shared versions of `libc` add: `suppress * "libc*"`
- To suppress array bounds messages in all functions called from `main` add: `suppress ab* ...; main`

Purify 4.0 Quick Reference

API functions

Include `<purifyhome>/purify.h` in your code and always link with `<purifyhome>/purify_stubs.a`
Useful compile/link options include: `-I`purify -print-home-dir` -L`purify -print-home-dir``

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 Purify'd
`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/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 color coding of memory

Build-time options

Set build-time options on the link line to build Purify'd programs:
`% purify -cache-dir=$HOME/cache -always-use-cache-dir cc ...`

Commonly used build-time options	Default
`-always-use-cache-dir`	**no**
Forces all Purify'd object files to be written to the global cache directory	
`-cache-dir`	**`<purifyhome>/cache`**
Specifies the global directory where Purify caches instrumented object files	
`-collector`	**not set**
Specifies the collect program to handle static constructors (for use with `gcc`, `g++`)	
`-ignore-runtime-environment`	**no**
Prevents the run-time Purify environment from overriding the option values used in building the program	
`-linker`	**system-dependent**
Sets the alternative linker to build the executables instead of the system default	
`-print-home-dir`	
Prints the name of the directory where Purify is installed, then exits	

Using Purify with other Pure Software products

Product	Command line syntax
PureCoverage	`% purify <purifyoptions> purecov <purecovoptions> cc ...`
PureLink	`% purelink <purelinkoptions> purify <purifyoptions> cc ...`
Quantify	Cannot instrument for Purify and Quantify simultaneously

Purify 4.0 Quick Reference

Run-time options

Set run-time options using the `PURIFYOPTIONS` environment variable:

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

Commonly used run-time options	Default
`-auto-mount-prefix`	**`/tmp_mnt`**
Removes the prefix used by file system auto-mounters	
`-chain-length`	**`6`**
Sets the maximum number of stack frames to print in a report	
`-fds-in-use-at-exit`	**`yes`**
Specifies that the file descriptor in use message be displayed at program exit	
`-follow-child-processes`	**`no`**
Controls whether Purify monitors child processes in a Purify'd program	
`-jit-debug`	**`not set`**
Enables just-in-time debugging	
`-leaks-at-exit`	**`yes`**
Reports all leaked memory at program exit	
`-log-file` †	**`stderr`**
Writes Purify output to a log file instead of the Viewer window	
`-messages`	**`first`**
Controls display of repeated messages: `"first"`, `"all"` or in a `"batch"` at program exit	
`-program-name`	**`argv[0]`**
Specifies the full pathname of the Purify'd program if `argv[0]` contains an undesirable or incorrect value	
`-show-directory`	**`no`**
Shows the directory path for each file in the call chain, if the information is available	
`-show-pc`	**`no`**
Shows the full pc value in each frame of the call chain	
`-show-pc-offset`	**`no`**
Appends a pc-offset to each function name in the call chain	
`-view-file` †	**`not set`**
Saves Purify output to a view file (`.pv` file) instead of the Viewer. To examine a view file, use `purify -view <filename>.pv`	
`-user-path`	**`not set`**
Specifies a list of directories in which to search for programs and source code	
`-windows`	**`not set`**
Redirects Purify output to `stderr` instead of the Viewer if `-windows=no`	

† *Can use conversion characters listed below.*

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

Index

Symbols