Debugging Dynamic Memory Usage Errors Using HP WDB

HP Part Number: 5014-0301
Published: July 2007
Edition: 2.0

Table of Contents

List of Tables

List of Examples

Introduction

HP Wildebeest Debugger (WDB) is an HP-supported implementation of the open source debugger GDB. Apart from the normal debugging functions, it also enables you to debug memory-related errors in a program.

HP WDB supports memory-debugging (using Run Time Checking (RTC)) of source-level programs written in HP C, HP aC++, and Fortran 90 on Itanium®-based systems running HP-UX 11i v2, or HP-UX 11i v3, and PA-RISC systems running HP-UX 11.0, HP-UX 11i v1, HP-UX 11i v2, or HP-UX 11i v3 operating systems.

WDB offers the following memory-debugging capabilities:

- Reports memory leaks
- Reports heap allocation profile
- Stops program execution if bad writes occur with string operations such as `strcpy()`, and `memcpy()`
- Stops program execution when freeing unallocated or deallocated blocks
- Stops program execution when freeing a block if bad writes occur outside block boundary
- Stops program execution conditionally based on whether a specified block address is allocated or de-allocated
- Scrambles previous memory contents at `malloc()`, and `free()` calls
- Simulates and detects out-of-memory event errors
- Monitors changes in data segment space allocation

Intended Audience

This document is intended for C, C++, and Fortran programmers who use WDB to detect and debug memory-related errors in HP C, HP aC++ and Fortran 90 applications. Reader of this document must be familiar with the basic commands supported by WDB.

Typographic Conventions

This document uses the following typographical conventions:

`$`, `$`or `#`	A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells. A number sign represents the superuser prompt.
gdb(5)	A manpage. The manpage name is *gdb*.
`Command`	A command name or qualified command phrase.
`Computer output`	Text displayed by the computer.
`ENVIRONMENT VARIABLE`	The name of an environment variable, for example, `PATH`.
[ERROR NAME]	The name of an error, usually returned in the `errno` variable.
`Variable`	The name of a placeholder in a command, function, or other syntax display that you replace with an actual value.
< >	The contents are optional in syntax. If the contents are a list separated by |, you must choose one of the items.
[ ]	The contents are optional in syntax. If the contents are a list separated by |, you must choose one of the items.
|	Separates items in a list of choices.
IMPORTANT	This alert provides essential information to explain a concept or to complete a task
NOTE	A note contains additional information to emphasize or supplement important points of the main text.

Related Information

The HP WDB documentation is available at the following location:

`/opt/langtools/wdb/doc/`

Table 1 lists the documentation available for WDB.

Table 1 Documentation for HP WDB

Document	Location
Debugging with GDB	`/opt/langtools/wdb/doc/gdb.pdf`

GDB Quick Reference Card	`/opt/langtools/wdb/doc/refcard_a4.pdf`
	`/opt/langtools/wdb/doc/refcard_a3.df`
	`/opt/langtools/wdb/doc/refcard.pdf` (Letter Format)

Getting Started with WDB	`/opt/langtools/wdb/doc/html/wdb/C/GDBtutorial.html`
WDB Online Help	`/opt/langtools/wdb/doc/index.html`
HP WDB GUI Documentation	`/opt/langtools/wdb/doc/html/wdbgui/C/`
GDB manpage	`gdb(1)`

For the most current WDB documentation, see the *HP WDB technical resources* website at:

http://www.hp.com/go/wdb

Prerequisites

Following are the prerequisites for debugging memory-related problems in WDB:

- The memory-debugging feature in WDB is dependent on the availability of the dynamic Linker Version `B.11.19`.
- WDB uses the heap debugging library, `librtc.[sl|so]`, to enable memory-debugging support. The `librtc.[sl|so]` library is a part of the HP WDB product. If the debugger is installed in a directory other than the default `/opt/langtools/bin` directory, you must use the environment variable, `LIBRTC_SERVER`, to set the path of the appropriate version of `librtc.[sl|so]`.

  From HP WDB 5.7 onwards, the archive version of the run time check library, `librtc.a`, is not available. You must use the shared version of the library, `librtc.[sl|so]`, instead.
- WDB does not support debugging of programs that link with the archive version of the standard C library, `libc.a`, or the core library, `libcl.a`. The programs must be linked with `libc.[sl|so]`.
- The memory-debugging feature is supported only for programs that directly or indirectly call `malloc()`, `realloc()`, `free()`, `mmap()`, or `munmap()` from the standard C library, `libc.[sl|so]`, or a third party (custom allocator) implementation of these functions.
- The memory debugging feature is not supported for `CMA` threaded programs.
- The memory debugging feature cannot be used with applications that redefine or override the default system-supplied versions of the standard library routines (under `libc.so` and `libdl.so`), such as `abort()`, `strcat()`, `ctime()`, and `dlclose()`. Before enabling the memory debugging feature in WDB, use the `nm(1)` command to determine if the application or the dependent libraries in the application redefine or substitute the standard library routines. For more information on the dependent standard library routines, see the HP WDB release notes, available at the *HP WDB Documentation* website at:

  http://www.hp.com/go/wdb

Memory-Related Errors

This section discusses the following memory-related errors that can occur in an application:

- Heap corruption
- Memory leaks
- Access errors

Heap Corruption

A heap corruption occurs when an application erroneously overwrites some of the data in the heap. Heap corruption can result in data corruption, memory corruption, or both.

When an application inadvertently uses the erroneously overwritten data in the heap, it results in **data corruption** in the application. Data corruption can lead to unpredictable program behavior.

The data corruption in the heap can lead to **memory corruption** if the corrupted data in the heap is used by memory management functions in the application to allocate, access, or deallocate memory blocks. In other words, memory corruption occurs when the corrupted datum in the heap is accessed as a pointer. Memory corruptions compromise the data integrity of the application and can result in segmentation violations if the erroneously allocated or accessed memory blocks are out of the bounds of the virtual memory of the application.

Causes for Heap Corruption

Following are some of the typical causes for heap corruption:

Double-Free

A double-free error occurs when a program attempts to free a memory block that is already freed. (Example 16 (page 67) illustrates how WDB detects double-frees.)

Freeing Unallocated/Uninitialized Memory

Heap corruption occurs when a program tries to free memory that is not allocated to the program. Such instances include freeing uninitialized pointers where the pointer addresses memory outside the allocated memory. (Example 17 (page 68) illustrates how WDB detects such errors.)

Accessing freed memory

Accessing freed memory results in heap corruption. The scramble feature is a minimal aid to detect such errors. See "Scrambling a Heap Block" (page 50) for more information.

Memory Leaks

A memory leak occurs when an application fails to free allocated memory. As a result, the kernel frees the memory that is allocated by a process only when the process terminates. If the program leaks memory on a continual basis, the virtual memory requirement for the process continues to increase and this can result in serious consequences for long-running applications and memory intensive applications.

Memory leaks can also cause fragmentation of the heap. This slows down the allocation, de-allocation, and access of memory blocks and can eventually cause the application to fail with out-of-memory errors.

When to Suspect a Memory Leak?

You must suspect a memory leak in an application if the system runs out of swap space, runs slower, or both. Memory leaks in an application increase the memory consumption in an application. When the memory consumed by the application exceeds the resource limits set by the kernel, the application fails with out-of-memory errors.

WDB enables you to detect out-of-memory conditions through runtime memory checking. It also enables you to simulate out-of-memory conditions in an application to understand application behavior under such conditions.

For information on how you can use WDB to simulate and detect out-of-memory conditions in an application, see "Error Injection " (page 28)

Types of Memory Leaks

Following are the types of memory leaks:

- **Physical Leaks**

  A physical leak is a definite memory leak that occurs when an application loses all handles, or all pointers to the allocated memory. If a valid pointer to a memory block is absent, the elusive block of memory cannot be accessed or freed.

  The handles to a memory block are typically lost under the following conditions:

  — When an application overwrites a pointer that addresses a block of memory with another address or data
  — When a pointer variable goes out of scope
  — When you free a structure or an array that has pointers which are not freed

  When all handles to a block of memory are lost, it causes the block to be leaked. Example 19 (page 70), Example 20 (page 71), and Example 21 (page 72) illustrate how WDB detects memory leaks.

- **Logical Leaks**

  A logical leak occurs when an application fails to optimally utilize the allocated memory. In this case the allocated block of memory can still be accessed through a pointer variable in the application.

The typical causes for logical leaks are listed below:

— Leaks caused by premature allocation of memory

  The application allocates the memory much ahead of the actual use of the allocated memory.

— Leaks caused by delayed de-allocation

  The application delays the freeing the allocated block beyond the actual use of the allocated memory.

— Leaks caused by failure to utilize allocated memory

  The application allocates memory, but fails to use the allocated memory.

NOTE: WDB supports the debugging of physical memory leaks only. It does not detect logical memory leaks.

Access Errors

Memory access errors can occur under the following conditions:

- When reading uninitialized local, or heap data
- When reading or writing to nonexistent, unallocated, or unmapped memory
- When a stray pointer overflows the bounds of a heap block, or tries to access a heap block that is already freed to cause buffer overruns and under-runs
- When reading or writing to memory locations that are already freed in the program

NOTE: WDB provides minimal support for debugging some of the memory access errors. The scrambling feature and detection for out-of-bounds writes are supported by WDB.

Using WDB to Debug Memory Problems

WDB supports the memory-debugging of applications involving dynamic allocations and de-allocations of virtual memory blocks, or during the calls to `libc` string routines like `strcpy()`, and `memcpy()`. It debugs memory-related problems at the time of allocation or de-allocation of memory blocks. It supports the detection of outstanding memory-related problems at specific user-defined probe-points (breakpoints) during the use of the memory blocks. Memory-related problems that appear after the specified probe points are not detected. It does not support the debugging of access errors that are caused when reading from or writing to unallocated, uninitialized, or de-allocated memory.

WDB does not support the memory-debugging of the stack, static memory, and register memory.

WDB provides the interactive, batch, and attach modes for debugging memory-related problems. See “Supported Modes of Memory-debugging in WDB” (page 51) for more information on the supported modes for debugging.

HP aC++/ HP C Compiler Runtime Checking Options

The HP aC++/HP C compiler also provides options for enabling memory debugging using WDB in Integrity Systems. This feature is supported only on Integrity Systems.

Table 2 list the runtime checking options are available in HP aC++/HP C compilers for memory debugging.

Table 2 Compiler Options for Memory Debugging

Compiler Option	Description
`+check=[all|none|bounds|malloc|stack|uninit]`	The `+check` compiler options provide runtime checks to detect out-of-bounds array references `(+check=bounds)`, memory leaks and heap corruption `(+check=malloc)`, writing outside the stack frame `(+check=stack)`, and uninitialized variables `(+check=uninit)`. The `+check=all` option enables all the available runtime checks for the `+check` compiler option. A failed check results in the program abort at runtime. The error message and the stack trace is printed to `stderr` before the program terminates. The environment variable `RTC_NO_ABORT` must be set to `1` to continue the program execution after a failed runtime check. This enables you to collect the diagnostics for all the failed checks in a single execution run.

NOTE: The `+check` options must be specified at compile time and link time. If different `+check` options are specified while compiling different source files, all the specified `+check` options are needed at link time. Multiple `+check` options are interpreted from left to right with the options on the right overriding earlier `+check` options.

For more information on the HP aC++ compiler options for memory debugging, see the *HP aC++ World Wide Webpage* at:

http://www.hp.com/go/cpp

For more information on the HP aC++ compiler options for memory debugging, see the *HP C World Wide Webpage* at:

http://www.hp.com/go/c

Memory-Debugging Features of WDB

WDB supports the following memory-debugging features:

- Heap Profiling features
- Leak Profiling feature
- Error Injection features
- Event Monitoring features

In addition to these features, HP WDB provides the following generic commands for memory debugging:

Table 3 Generic Commands for Memory Checking

Command	Description
`set heap-check <on/off>`	Toggles the setting of commands for detecting leaks, bounds, double frees, and heap profiling.
`show heap-check`	Displays the current settings for memory checking

NOTE: GDB reports an incorrect stack trace after `dlclose` or `shl_unload`, and a subsequent `dlopen` or `shl_load`. The leaks are displayed erroneously when the memory address range overlaps between the newly loaded shared library, and the recently unloaded shared library.

Workaround: Place a breakpoint at `dlclose` or `shl_unload`, and enter the `info leaks` command to view the leaks accurately when a shared library is unloaded.

Heap Profiling

You can profile the heap usage in an application by using WDB. The heap-profiling feature enables you to analyze the influence of algorithms and data structures on heap usage and tune the memory requirements of an application.

WDB supports the following heap-analysis profiles:

- Snapshot Profile
- Incremental Heap Profile
- Arena Profile

NOTE: Heap profiling must be enabled to view heap reports. The `set heap-check on` command enables heap profiling also.

Snapshot Profile

The snapshot profile displays the outstanding heap allocations at a specific instant (probe point) at runtime. It does not display the blocks that are already freed before the probe point.

Table 4 lists the basic commands used for heap profiling.

Table 4 Commands for Heap Profiling

Command	Description
`info heap`	Displays the heap report that includes the current heap allocations, the sizes of the blocks allocated, and number of allocation instances.
`info heap <filename>`	Writes the heap report output to the specified file.
`info heap <idnumber>`	Displays detailed information about the specified heap allocation including the allocation call stack.
`set heap-check min-heap-size <num>`	Reports the heap allocations that exceed the specified number, `<num>`, of bytes based on the cumulative number of bytes that are allocated at each call-site inclusive of multiple calls to `malloc()` at a particular call site. See Example 1 (page 17) for more information.

To obtain a snapshot heap profile, complete the following steps:

1. Run the debugger and load the program by entering the following command at command prompt:

```
$ gdb <executable> <arguments>
(gdb) set heap-check on
```

NOTE: The `set heap-check on` command enables the memory-debugging feature in WDB. This enables the detection of leaks, heap profiles, bounds checking, checking for double free

2. Set a breakpoint by entering the following command:

```
(gdb) b <probepoint>
```

3. Run the program by entering the following command:

   ```
   (gdb)run
   ```

4. When the program is stopped at a breakpoint, enter the following `info heap` command:

   ```
   (gdb) info heap
   ```

   The following output is displayed:

   ```
   Analyzing heap ...done

   Actual Heap Usage:
   Heap Start = 0x40408000
   Heap End = 0x4041a900
   Heap Size = 76288 bytes

   Outstanding Allocations:
   41558 bytes allocated in 28 blocks

   No. Total bytes Blocks Address      Function

   0   34567        1     0x40411000    foo()
   1    4096        1     0x7bd63000    bar()
   2    1234        1     0x40419710    baz()
   3     245        8     0x404108b0    boo()
   [...]
   ```

5. To view a specific allocation, specify the allocation number as an argument to the `info heap` command.

   For example:

   ```
   (gdb) info heap 1
   4096 bytes at 0x7bd63000 (9.86% of all bytes allocated)
   in bar () at test.c:108
   in main () at test.c:17
   in _start ()
   in $START$ ()
   ```

   When multiple blocks are allocated from the same call stack, WDB displays additional information similar to the following:

   ```
   (gdb) info heap 3
   245 bytes in 8 blocks (0.59% of all bytes allocated)
   These range in size from 26 to 36 bytes and are allocated
   in boo ()
   in link_the_list () at test.c:55
   in main () at test.c:13
   in _start ()
   ```

   You can control the stack frames that are collected for reporting at any allocation point. For more information on this feature, see "Settings to Manage Performance Degradation. " (page 51)

Example 1 (page 17) illustrates the use of the `info heap` command with the `min-heap-size` filter setting.

Example 1 Filtered Heap Reporting for Allocations Exceeding <num> at a Particular Call-Site

Sample Program

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int i, *arr[1000];
for (i=0; i < 1000; i++)
arr[i] = malloc (49);
malloc (30);
set_brkpt_here(0)
exit(0);

}
```

Sample Debugging Session

```
$ gdb minheap
(gdb) b set_brkpt_here
(gdb) set heap-check min-heap-size 31
(gdb) run
(gdb) info heap
Analyzing heap ...


49000 bytes allocated in 1000 blocks

No.   Total bytes     Blocks     Address     Function
0        49000          1000    0x4044eff0    main()
```

Incremental Heap Profile

The incremental profile displays the outstanding allocations at multiple probe points in an application at runtime. This profile is analogous to processing multiple snapshot profiles. Example 2 (page 19) illustrates this feature.

Table 5 lists the commands for incremental heap-profiling.

Table 5 Commands for Incremental Heap-Profiling

Command	Description
`set heap-check interval <nn>`	Starts the incremental heap growth profile. All allocations prior to the execution of this command are ignored. If incremental heap growth profile is already on, executing this command resets the counters and starts a fresh collection. The interval is specified in seconds.
`set heap-check repeat <nn>`	Enables you to specify the number of intervals for which WDB must collect the incremental heap growth. The default value is 100. Every repeat of the interval tracks heap allocation during that interval.
`info heap-interval <filename>`	Creates a detailed report of the heap growth. The data for each interval has the start and end time of the interval. If a filename is specified, the detailed report is written in the specified file.
`set heap-check reset`	When incremental heap profile is used the heap growth data is internally stored by WDB in a temporary file. The heap growth data gathered during each interval is appended to this file . If the session is very long, this file may become very large. This command discards the data existing in the file and creates a new data file. If the command is executed, the user cannot see the old data in the file.

Example 2 Incremental Heap Profile

Sample Program

```
$ cat testincremental.c
#include <stdio.h>
#include <malloc.h>

char *f1_p;
char *f2_p;
char *f3_p;

void marker1()
{}

int
func1()
{
    int i;
    for (i = 0; i< 2; i++)
      f1_p = (char *)malloc(10);
      return 1;
}


int
func2()
{
     int i;
     for (i = 0; i< 2; i++)
        f2_p = (char *)malloc(20);
     return 1;
 }


 int
 func3()
 {
     int i;
     for (i = 0; i< 2; i++)
        f3_p = (char *)malloc(30);
     return 1;
 }


 main()
 {
  int i;
   int repeat;

   for (repeat = 0; repeat < 2; repeat++)
        for (i = 0; i < 2; i++) {
           func1();
            sleep (1);
         }

     /* 2 interval records */
     marker1();

    for (repeat = 0; repeat < 2; repeat++)
        for (i = 0; i < 2; i++) {
            func2();
           sleep (1);
        }

  /* 4 (3 old and 1 new) interval records;
    as the repeat count of 100 has been exceeded.
   */
  marker1();


  /* set repeat count to 500 */
  for (repeat = 0; repeat < 2; repeat++)
      for (i = 0; i < 2; i++) {
          func3();
          sleep (1);
      }

 marker1();
  exit(0);
}
```

Sample Debugging Session

```
(gdb) file testincremental
Reading symbols from testincremental.. done.
(gdb) set heap-check interval 1
(gdb) set heap-check repeat 2
(gdb) b marker1
Breakpoint 1 at 0x4000000000000e50:0: file testincremental.c,
line 9 from testincremental
(gdb) r
Starting program: testincremental
Breakpoint 1, marker1 () at testincremental.c:9    {}
(gdb) info heap-interval
Analyzing heap ...
==========================================================
Start Time: Mon Oct 30 01:38:11 2006
End Time: Mon Oct 30 01:38:12 2006
Interval: 1
40 bytes allocated in 4 blocks
No.   Total bytes     Blocks     Address     Function
0       40             4       0x6000000000004840 func1()
(gdb) c
Continuing.
Breakpoint 1, marker1 () at /testincremental.c:99   {}
(gdb) info heap-interval
Analyzing heap ...
==========================================================
Start Time: Mon Oct 30 01:38:11 2006
End Time: Mon Oct 30 01:38:12 2006
Interval: 1
40 bytes allocated in 4 blocks
No.   Total bytes     Blocks     Address     Function
0       40             4       0x6000000000004840  func1()
===========================================================
Start Time: Mon Oct 30 01:38:13 2006
End Time: Mon Oct 30 01:38:14 2006
Interval: 2
40 bytes allocated in 4 blocks
No.   Total bytes     Blocks     Address           Function
0       40             4       0x60000000000048c0  func1()
===========================================================
```

Arena Profile

WDB enables you to view the high-level memory-usage statistics of a running application. You can analyze the memory-usage statistics to understand the memory consumption, the allocation pattern, and the heap-fragmentation of the application.

WDB enables you to view the following memory-usage statistics:

- High level memory-usage statistics of a process
- High level memory-usage statistics of each arena
- Block level and overall memory-usage statistics of each arena
- Block level and overall memory-usage statistics of each arena along with the allocated stack trace for each allocated block.

NOTE: For more information on arenas, see the *malloc*(3c) manpages.

Table 6 lists the commands for monitoring memory-usage in an arena.

Table 6 Memory-Usage in an Arena

Command	Description
`info heap process`	Displays the high level memory-usage of a process. Lists the number of free blocks, used blocks, small blocks, holding blocks, node blocks and regular blocks.
`info heap arenas <arena_id>`	Displays the high level memory-usage details of the specified arena `<arena_id>`. It also lists the number of free blocks, used blocks, small blocks, holding blocks and regular blocks. If the `<arena_id>` is not specified, it displays the memory-usage statistics for the current arena.
`info heap arenas <arena_id> blocks`	Displays the memory-usage statistics of all the blocks in the given arena, in the increasing order of block-addresses.
`info heap arenas <arena_id> blocks <block-id>`	Displays the memory-usage statistics of a specific block in the arena with the stack trace for the specified arena and block.
`info heap arena <arena_id> blocks stacks`	Displays the overall and block level memory-usage statistics, with stack trace wherever applicable.

The `info heap process` and `info heap arenas` commands do not require re-linking or rebuilding of the application. You can attach a running process to the debugger and get a snapshot of the heap-profile of the process. Example 3 (page 24) illustrates the use of the `info heap process` and `info heap arenas` commands.

NOTE:

- The `info heap arenas` and `info heap process` commands are not supported in batch mode.
- The `info heap process` and `info heap arenas` are available only on HP-UX 11i v3.
- The stack trace is displayed only if memory debugging is enabled. (Enable the `set heap-check on` command to enable memory debugging). If the stack trace is not required, the memory-usage statistics can be viewed without enabling memory checking.

Analyzing the `info heap process` output

The `info heap process` command displays the number of used, free, small, holding, node and regular (ordinary) blocks.

If there are a larger number of free small blocks, you can suspect heap-fragmentation. The application does not differentiate between a small block and an ordinary block. However, you can tune `malloc()` to use a specific ratio of small and ordinary blocks and reduce heap-fragmentation.

The holding block headers and node blocks are used for the internal data-structure and bookkeeping in `malloc()`. The sum of the total bytes in holding block headers and node blocks determines the efficiency of `malloc()`. The memory allocator is more efficient when it uses less memory for the internal data-structure and bookkeeping.

Following is a sample output of the `info heap process` command:

```
(gdb) info heap process
Total space in arenas: 4657088
Number of bytes in free small blocks: 69216
Number of bytes in used small blocks: 199584
Number of bytes in free ordinary blocks: 2480
Number of bytes in used ordinary blocks: 4375600
Number of bytes in holding block header:  912
Number of small blocks: 3500
Number of ordinary blocks: 9
Number of holding blocks: 0
Number of free ordinary blocks: 1
Number of free small blocks: 388
Small block allocator parameters
        enabled: 1
        maxfast: 512
        numblks: 100
        grain: 16

cache
        enabled: 0
        miss: 0
        bucketsize: 0
        buckets: 0
        retirement: 0
Exec type: SHARE_MAGIC
```

Analyzing the `info heap arenas` output

The `info heap arenas` command displays the memory-usage statistics for the specified arena. You can analyze the memory-usage statistics of all the arenas to determine if there is an imbalance in memory-usage across the arenas. For example, if there are many free blocks in an arena and these blocks are not used by threads from another arena, you can tune the memory-usage to optimize the performance.

The `info heap arenas <arena_id> blocks` displays the details of all the blocks in the given arena in an increasing order of addresses. You can analyze the size of the blocks in the increasing order of block-addresses to detect heap-fragmentation. For example, you can suspect heap fragmentation if two large free blocks are separated by a small used block.

Following is a sample output of the `info heap arenas` command:

```
(gdb) info heap arenas
num_arenas: 1
expansion: 4096

Arena ID: 0

Total number of blocks in arena: 47
Start address: 0x4001003c
Ending address: 0x40480ffc
Total space: 4657088
Number of bytes in free small blocks: 69216
Number of bytes in used small blocks: 199584
Number of bytes in free ordinary blocks: 2480
Number of bytes in used ordinary blocks: 4375600
Number of bytes in holding block header: 912
Number of small blocks: 3500
Number of ordinary blocks: 9
Number of holding blocks: 35
Number of free ordinary blocks: 1
Number of free small blocks: 388
```

Example 3 Monitoring memory usage in an arena

Sample Program

```
$ cat malloc_1.c
/* test large malloc.
* corruption.
*/
#include <stdio.h>
#include <stdlib.h>
void f1()
{
char * cp;

cp = malloc (5000);
}
void f2()
{
char * cp;

 cp = malloc (7000);
 }
void f3()
{
char * cp;

cp = malloc (3000);
}
void f4()
{
char * cp;

 cp = malloc (6000);
}
void f1_small()
{
char * cp;

cp = malloc (50);
}
void f2_small()
{
char * cp;

cp = malloc (70);

void f3_small()
{
char * cp;

cp = malloc (30);
}
void f4_small()
{
char * cp;

cp = malloc (60);
}

void set_brkpt_here() {
}
int main()
{

 for (int i=0; i<777; i++)
{
  f4_small();
  f1_small();
  f2_small();
  f3_small();
}
set_brkpt_here();

for (int i=0; i<1000; i++)
{
  f4();
  f1();
  f2();
  f3();

}
set_brkpt_here();
}
```

Sample Debugging Session

```
(gdb) -leaks  malloc_1.32
(gdb) b set_brkpt_here
(gdb) run
(gdb) info heap process
Total space in arenas: 4657088
Number of bytes in free small blocks: 69216
Number of bytes in used small blocks: 199584
Number of bytes in free ordinary blocks: 2480
Number of bytes in used ordinary blocks: 4375600
Number of bytes in holding block header:  912
Number of small blocks: 3500
Number of ordinary blocks: 9
Number of holding blocks: 0
Number of free ordinary blocks: 1
Number of free small blocks: 388
Small block allocator parameters
        enabled: 1
        maxfast: 512
        numblks: 100
        grain: 16
cache
        enabled: 0
        miss: 0
        bucketsize: 0
        buckets: 0
        retirement: 0
Exec type: SHARE_MAGIC
```

```
(gdb) info heap arenas
num_arenas: 1
expansion: 4096

Arena ID: 0

Total number of blocks in arena: 47
Start address: 0x4001003c
Ending address: 0x40480ffc
Total space: 4657088
Number of bytes in free small blocks: 69216
Number of bytes in used small blocks: 199584
Number of bytes in free ordinary blocks: 2480
Number of bytes in used ordinary blocks: 4375600
Number of bytes in holding block header: 912
Number of small blocks: 3500
Number of ordinary blocks: 9
Number of holding blocks: 35
Number of free ordinary blocks: 1
Number of free small blocks: 388
```

Leak Profiling

The leak profile feature in WDB conservatively identifies the blocks of memory that are leaked in an application, and displays the stack trace that shows when the block was allocated. All the leaks detected by WDB are definite physical leaks.

WDB uses a garbage collection algorithm to identify the blocks that are leaked. It identifies the root-set of memory that are possible pointers to the heap. The initial root-set includes the shared library data, the program/thread stacks, the registers, the thread specific private data, the `mmap` regions, and the shared memory regions. The initial root-set includes all data except the heap blocks.

The debugger considers suitably aligned words in the root-set as possible pointers to the heap. The debugger performs a reachability analysis based on the root-set, and determines the memory blocks that are reachable through possible pointers from the root-set. The heap blocks that are not reachable through possible pointers from the root-set are reported as leaks.

WDB is conservative in detecting the memory leaks. The memory leaks can be masked if a datum in the root-set inadvertently holds a possible pointer to a heap block.

Table 7 lists the basic commands for leak profiling in WDB.

Table 7 Commands for Leak Profiling

Command	Description
`set heap-check leaks <on/off>`	Controls WDB memory leak checking
`info leaks`	Displays a leak report. It also lists information such as the leaks, size of blocks, and number of instances.
`info leaks <filename>`	Writes the complete leak report output to the specified file
`info leak <leaknumber>`	Displays detailed information on the specified leak including the allocation call stack
`set heap-check min-leak-size <num>`	Specifies the minimum leak size for stack trace collection. The debugger continues to report leaks that are smaller than `<num>` bytes, but it does not provide the stack trace for the same. By default, `num` is set to 0. This command also enables you to reduce performance degradation. See "Settings to Manage Performance Degradation. " (page 51)

To view the leak profile, complete the following steps:

1. Run the debugger and load the program by entering the following command:

   ```
   $ gdb <executable> <arguments>
   ```

   or

   ```
   $ gdb -leaks <executable> <arguments>
   ```

2. Enable leak checking by entering the following command:

   ```
   (gdb) set heap-check leaks on
   ```

   (if the `-leaks` option is not used in Step 1)

NOTE: Alternatively, you can use the `set heap-check on` command to automatically enable the detection of leaks by toggling the `set heap-check leaks on` command. This command enables the detection of leaks, heap profiles, bounds checking, and checking for double frees.

3. Set breakpoints in the code at probe-points where you want to examine cumulative leaks by entering the following command:

   ```
   (gdb) b <probe-points>
   ```

4. Run the program in the debugger by entering the following command:

   ```
   (gdb) run
   ```

5. When the breakpoint triggers, enter the following info leaks command to display the list of memory leaks:

   ```
   (gdb) info leaks
   ```

   The following output is displayed:

   ```
   Scanning for memory leaks...done

   2439 bytes leaked in 25 blocks

   No. Total bytes   Blocks    Address    Function
   0     1234           1      0x40419710    foo()
   1      333           1      0x40410bf8    main()
   2      245           8      0x40410838    strdup()

   [...]
   ```

   The debugger assigns a numeric identifier for each leak. To view a stack trace for a specific leak, specify the leak number from the list of leaks, as follows:

   ```
   (gdb) info leak 2
   245 bytes leaked in 8 blocks (10.05% of all bytes leaked)
   These range in size from 26 to 36 bytes and are allocated in strdup ()
   in link_the_list () at test.c:55
   in main () at test.c:13
   in _start ()
   ```

Error Injection

WDB supports error injection features to debug out-of-memory events in an application. It enables you to simulate out-of-memory conditions in an application and analyze the behavior of the applications under such conditions. In addition, it enables you to gain control over program execution when an out-of-memory event occurs.

To simulate an out-of-memory condition, you must use the `set heap-check null-check` command to force `malloc()` to return `NULL` after `<N>` or a random number of allocations. After simulating the out-of-memory error, you can use the `catch nomem` command to gain control over the execution when an out-of-memory error occurs.

Table 8 lists the commands available for error injection.

Table 8 Commands for Error Injection

Command	Description
`set heap-check null-check <N>`	Forces `malloc()` to return `NULL` after `<N>` invocations of `malloc()`. Example 4 (page 29) illustrates the use of this command.
`set heap-check null-check-size <N>`	Forces `malloc()` to return `NULL` after `<N>` bytes are allocated by the program. Example 5 (page 31) illustrates the use of this command.
`set heap-check null-check random`	Forces `malloc()` to return `NULL` after random number of invocations of `malloc()`. Example 6 (page 33) illustrates the use of this command.
`set heap-check random-range <N>`	Defines the range for random number calculation for the `set heap-check null-check random` command
`set heap-check seed-value <N>`	Defines the seed-value for random number calculation for the `set heap-check null-check random` command
`catch nomem`	Enables the user to gain control over an out-of-memory event. The user can step through program execution after the `nomem` event is detected.

Example 4 Simulating out-of-memory conditions after <N> allocations

Sample Program

```
bash-2.05b$ cat null-check.c
1 #include <stdio.h>
2 #include <malloc.h>
3
4 int cnt;
5
6 void break_here()
7 {
8 }
9
10 void test_null_check()
11 {
12     int i;
13     char *a;
14
15     cnt = 0;
16     for(i = 0; i <= 10; i++) {
17      a = malloc(100);
18       if (a == NULL)
19       printf("Out of memory scenario simulated\n");
20     }
21 }
22
23 int main()
24 {
25         test_null_check();
26         exit (0);
27 }
```

Sample Debugging Session

```
$ /opt/langtools/bin/gdb
HP gdb 5.5.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00
and target hppa1.1-hp-hpux11.00.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.5.0 (based on GDB) is covered
by the GNU General Public License. Type "show copying" to
see the conditions to change it and/or distribute copies.
Type "show warranty" for warranty/support.

(gdb) file null
Reading symbols from null...done.
(gdb) set heap-check null-check 6
(gdb) b main
Breakpoint 1 at 0x2c64: file null-check.c, line 25 from null.
(gdb) b 19
Breakpoint 2 at 0x2c10: file null-check.c, line 19 from null.
(gdb) r
Starting program: null

Breakpoint 1, main () at null-check.c:25
25              test_null_check();
(gdb) catch nomem
Catchpoint 3 (nomem)
(gdb) c
Continuing.
warning: Malloc is returning simulated 0x00000000 value
0x70e78e8c in __rtc_nomem_event+0x4 ()
from /opt/langtools/lib/librtc.sl
(gdb) c
Continuing.
```

```
Breakpoint 2, test_null_check () at null-check.c:19
19       printf("Out of memory scenario simulated\n");
(gdb) bt
#0  test_null_check () at null-check.c:19
#1  0x2c70 in main () at null-check.c:25
#2  0x70ee3478 in _start+0xc0 () from /usr/lib/libc.2
(gdb) c
Continuing.
Out of memory scenario simulated

Program exited normally.
```

Example 5 Simulating out-of-memory conditions after <N> bytes are allocated

Sample Program

```
$ cat null-check.c
1 #include <stdio.h>
2 #include <malloc.h>
3
4 int cnt;
5
6 void break_here()
7 {
8 }
9
10 void test_null_check()
11 {
12     int i;
13     char *a;
14
15     cnt = 0;
16     for(i = 0; i <= 10; i++) {
17      a = malloc(100);
18       if (a == NULL)
19        printf("Out of memory scenario simulated\n");
20     }
21 }
22
23 int main()
24 {
25         test_null_check();
26         exit (0);
27 }
```

Sample Debugging Session

```
$ /opt/langtools/bin/gdb
HP gdb 5.5.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00
and target hppa1.1-hp-hpux11.00.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.5.0 (based on GDB) is covered
by the GNU General Public License. Type "show copying" to
see the conditions to change it and/or distribute copies.
Type "show warranty" for warranty/support.

(gdb) set heap-check on
(gdb) file null
Reading symbols from null...done.
(gdb) b test_null_check
Breakpoint 1 at 0x2be0: file null-check.c, line 15 from null.
(gdb) b 19
Breakpoint 2 at 0x2c10: file null-check.c, line 19 from null.
(gdb) run
Starting program: null

Breakpoint 1, test_null_check () at null-check.c:15
15          cnt = 0;
(gdb) set heap-check null-check-size 400
(gdb) catch nomem
Catchpoint 3 (nomem)
(gdb) c
Continuing.
warning: Malloc is returning simulated 0x00000000 value
0x70e78e8c in __rtc_nomem_event+0x4() from /opt/langtools/lib/librtc.sl
(gdb) c
Continuing.

Breakpoint 2, test_null_check () at null-check.c:19
19          printf("Out of memory scenario simulated\n");
(gdb) p i
$1 = 4
```

```
(gdb) c
Continuing.
warning: Malloc is returning simulated 0x00000000 value
0x70e78e8c in __rtc_nomem_event+0x4 () from /opt/langtools/lib/librtc.sl

(gdb) bt
#0  0x70e78e8c in __rtc_nomem_event+0x4 () from /opt/langtools/lib/librtc.sl
#1  0x70e7b554 in handle_null_check+0x134 () from /opt/langtools/lib/librtc.sl
#2  0x70e7b614 in malloc+0xb4 () from /opt/langtools/lib/librtc.sl
#3  0x2c04 in test_null_check () at null-check.c:17
#4  0x2c70 in main () at null-check.c:25
#5  0x70ee3478 in _start+0xc0 () from /usr/lib/libc.2
(gdb) c
Continuing.

Breakpoint 2, test_null_check () at null-check.c:19
19        printf("Out of memory scenario simulated\n");
(gdb) p i
$2 = 9
(gdb) c
Continuing.
Out of memory scenario simulated

Program exited normally.
```

Example 6 Simulating out-of-memory conditions after a random number of allocations

Sample Program

```
 bash-2.05b$ cat null-random.c
1 #include <stdio.h>
2 #include <malloc.h>
3
4 int main()
5 {
6 int i;
7 char *a;
8    for (i=0; i <= 500; i++) {
9          a = malloc(100);
10         if (a == NULL)
11           {
12             printf("Out of memory simulated\n");
13           }
14     }
15         exit (0);
16 }
```

Sample Debugging Session

```
$ /opt/langtools/bin/gdb
HP gdb 5.5.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00
and target hppa1.1-hp-hpux11.00.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.5.0 (based on GDB) is covered by
the GNU General Public License. Type "show copying" to see the
conditions to change it and/or distribute copies.
Type "show warranty" for warranty/support.
(gdb) file null-random
Reading symbols from null-random...done.
(gdb) set heap-check on
(gdb) b main
Breakpoint 1 at 0x2be4: file null-random.c, line 8 from null-random.
(gdb) b 12
Breakpoint 2 at 0x2c10: file null-random.c, line 12 from null-random.
(gdb) r
Starting program: null-random

Breakpoint 1, main () at null-random.c:8
8          for (i=0; i <= 500; i++) {
(gdb) set heap-check null-check random
(gdb) catch nomem
Catchpoint 3 (nomem)
(gdb) c
Continuing.
warning: Malloc is returning simulated 0x00000000 value
0x70e78e8c in __rtc_nomem_event+0x4 () from /opt/langtools/lib/librtc.sl
(gdb) c
Continuing.

Breakpoint 2, main () at null-random.c:12
12                  printf("Out of memory simulated\n");
(gdb) p i
$1 = 55
(gdb) c
Continuing.
Out of memory simulated
warning: Malloc is returning simulated 0x00000000 value
0x70e78e8c in __rtc_nomem_event+0x4 () from /opt/langtools/lib/librtc.sl
(gdb) c
Continuing.
```

```
Breakpoint 2, main () at null-random.c:12
12                printf("Out of memory simulated\n");
(gdb) p i
$2 = 110
(gdb) c
Continuing.
Out of memory simulated
warning: Malloc is returning simulated 0x00000000 value
0x70e78e8c in __rtc_nomem_event+0x4 () from /opt/langtools/lib/librtc.sl
```

Event Monitoring

The event monitoring commands in WDB enable you to monitor specific heap events and heap-corruption problems in an application.

Monitoring Heap Events

WDB enables you to monitor specific events such as the size of memory allocations, the high water mark.

Table 9 lists the commands for monitoring heap events.

Table 9 Monitoring Heap Events

Command	Description
`set heap-check watch <address>`	Stops program execution when the block at the given address is allocated or de-allocated
`set heap-check block-size <num-bytes>`	Stops program execution when a program tries to allocate a block larger than `num-bytes` in size
`set heap-check heap-size <num-bytes>`	Stops program execution when the program tries to increase the program-heap by at least `num-bytes`
`info heap high-mem`	Displays the highest `brk()` value and the number of `brk()` value changes for a given run. This number signifies the number of times that the heap grows.
`set heap-check high-mem-count <X_number>`	Stops program execution when break value has moved `<X_number>` times
`set heap-check free <on|off>`	Toggles the detection of double-frees and frees with improper arguments

Monitoring a Specific Address

The `set heap-check watch` command enables you to monitor a specific address. It instructs the debugger to stop the program execution and transfer execution control to the user when the specified block at `<address>` is allocated, or de-allocated.

Following is the syntax for the `set heap check watch` command:

```
(gdb) set heap-check watch <address>
```

Example 7 (page 36) illustrates the use of the `set heap-check watch <address>` command.

Example 7 Monitoring a specific address

Sample Program

```
bash-2.05b$ cat watch-addr.c
1  #include<stdio.h>
2
3  void enable_watch(char *cp)
4  {
5
6  }
7
8  int main()
9  {
10   char *cp = (char*)malloc(100);
11  enable_watch(cp);
12   free(cp);
13  exit(0);
14  }
```

Sample Debugging Session

```
$ /opt/langtools/bin/gdb
HP gdb 5.5.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00
and target hppa1.1-hp-hpux11.00.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.5.0 (based on GDB) is covered
by the GNU General Public License. Type "show copying" to see
the conditions to change it and/or distribute copies.
Type "show warranty" for warranty/support.

(gdb) file watch-addr
Reading symbols from watch-addr...done.
(gdb) set heap-check on
(gdb) b main
Breakpoint 1 at 0x2bf0: file watch-addr.c, line 10 from watch-addr.
(gdb) b 13
Breakpoint 2 at 0x2c24: file watch-addr.c, line 13 from watch-addr.
(gdb) r
Starting program: watch-addr

Breakpoint 1, main () at watch-addr.c:10
10          char *cp = (char*)malloc(100);
(gdb) n
11          enable_watch(cp);
(gdb) p/x cp
$1 = 0x4042a3b8
(gdb) set heap-check watch 0x4042a3b8
(gdb) c
Continuing.
warning: Watch address 0x4042a3b8 deallocated
0x70e78d7c in __rtc_event+0 () from /opt/langtools/lib/librtc.sl
(gdb) bt
#0  0x70e78d7c in __rtc_event+0 () from /opt/langtools/lib/librtc.sl
#1  0x70e7ada0 in rtc_record_free+0xb8 () from /opt/langtools/lib/librtc.sl
#2  0x70e7b9a0 in free+0xc8 () from /opt/langtools/lib/librtc.sl
#3  0x2c24 in main () at watch-addr.c:12
#4  0x70ee3478 in _start+0xc0 () from /usr/lib/libc.2
(gdb) c
Continuing.

Breakpoint 2, main () at watch-addr.c:13
13          exit(0);
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

Monitoring Allocations Greater Than a Specified size

The `set heap-check block-size` command instructs WDB to stop the program and transfer the execution control to the user when the program allocates a heap block whose size is greater than or equal to `<num-bytes>`.

Following is the syntax for the `set heap-check block-size` command:

```
set heap-check block-size <num-bytes>
```

Example 8 (page 37) illustrates the use of the `set heap-check block-size` command.

Example 8 Monitoring allocations greater than a specified size

Sample Program

```
bash-2.05b$ cat block-size.c
1  #include<stdio.h>
2
3  int main()
4  {
5    char * cp;
6    printf("Start of the program\n");
7    cp = (char *)malloc(1024 *1024*10);
8    free (cp);
9    exit(0);
10  }
```

Sample Debugging Session

```
bash-2.05b$ /opt/langtools/bin/gdb
HP gdb 5.5.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00
and target hppa1.1-hp-hpux11.00.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.5.0 (based on GDB) is covered
by the GNU General Public License. Type "show copying" to see
the conditions to change it and/or distribute copies.
Type "show warranty" for warranty/support.
(gdb) file block-size
Reading symbols from block-size...done.
(gdb)  set heap-check on
(gdb) set heap-check block-size 900000
(gdb) b main
Breakpoint 1 at 0x2c04: file block-size.c, line 6 from block-size.
(gdb) b 9
Breakpoint 2 at 0x2c3c: file block-size.c, line 9 from block-size.
(gdb) r
Starting program: block-size

Breakpoint 1, main () at block-size.c:6
6         printf("Start of the program\n");
(gdb) c
Continuing.
Start of the program
warning: Attempt to allocate a large object at 0x4042c3e8
0x70e78d7c in __rtc_event+0 () from /opt/langtools/lib/librtc.sl
(gdb) bt
#0  0x70e78d7c in __rtc_event+0 () from /opt/langtools/lib/librtc.sl
#1  0x70e7a918 in rtc_record_malloc+0xf0 () from /opt/langtools/lib/librtc.sl
#2  0x70e7b7e0 in malloc+0x280 () from /opt/langtools/lib/librtc.sl
#3  0x2c28 in main () at block-size.c:7
#4  0x70ee3478 in _start+0xc0 () from /usr/lib/libc.2
(gdb) c
Continuing.

Breakpoint 2, main () at block-size.c:9
9         exit(0);
```

Monitoring the Program Heap Growth.

The `set heap-check heap-size` command instructs WDB to stop the program and transfer execution control to the user when the program attempts to increase the heap size of the program by `<num-bytes>` or more. Example 9 (page 38) illustrates the use of this command.

Following is the syntax for the `set heap-check heap-size` command:

```
set heap-check heap-size <num-bytes>
```

Example 9 Monitoring program heap growth

Sample Program

```
bash-2.05b$ cat heap-size.c
1  #include<stdio.h>
2
3 int main()
4  {
5    char * cp;
6    printf("Start of the program\n");
7    cp = (char *)malloc(1024 *1024*10);
8    free (cp);
9    cp = (char *)malloc(1024 *1024*8);
10   free (cp);
11   exit(0);
12  }
```

Sample Debugging Session

```
bash-2.05b$ /opt/langtools/bin/gdb
HP gdb 5.5.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00
and target hppa1.1-hp-hpux11.00.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.5.0 (based on GDB) is covered
by the GNU General Public License. Type "show copying" to
see the conditions to change it and/or distribute copies.
Type "show warranty" for warranty/support.
(gdb) file heap-size
Reading symbols from heap-size...done.
(gdb) set heap-check on
(gdb) set heap-check heap-size 8000000
(gdb) b main
Breakpoint 1 at 0x2c04: file heap-size.c, line 6 from heap-size.
(gdb) b 11
Breakpoint 2 at 0x2c60: file heap-size.c, line 11 from heap-size.
(gdb) r
Starting program: heap-size

Breakpoint 1, main () at heap-size.c:6
6         printf("Start of the program\n");
(gdb) c
Continuing.
Start of the program
warning: Attempt to grow the heap at 0x4042c3e0
0x70e78d7c in __rtc_event+0 () from /opt/langtools/lib/librtc.sl
(gdb) bt
#0  0x70e78d7c in __rtc_event+0 () from /opt/langtools/lib/librtc.sl
#1  0x70e7aff4 in malloc_padded+0xa8 () from /opt/langtools/lib/librtc.sl
#2  0x70e7b634 in malloc+0xd4 () from /opt/langtools/lib/librtc.sl
#3  0x2c28 in main () at heap-size.c:7
#4  0x70ee3478 in _start+0xc0 () from /usr/lib/libc.2
(gdb) c
Continuing.

Breakpoint 2, main () at heap-size.c:11
11        exit(0);
```

Monitoring Changes in Data Segment Space Allocation (High Water Mark Feature)

The high water mark feature records the number of times the break value changes which is the number of times the heap grows.

The high water mark feature monitors changes in the program break value. This value points to the end of the heap (which is also the end of the data segment). When memory is allocated using `malloc()` in excess of the available heap memory, the `brk()` call extends the heap. This changes the break value. Most implementations of `malloc()` do not decrease the heap size when the memory is freed. The break value is indicative of the memory consumption by the program.

Table 10 lists the commands that support the high water mark feature.

Table 10 Commands Supporting High Water-Mark Feature

Command	Description
`info heap high-mem`	Displays the number of times that the break value has been changed for the current run at the instant, the command is issued
`set heap-check high-mem-count <X_number>`	Stops when break value has moved the specified number, `<X_number>`, of times

NOTE: This feature assumes that an application has a deterministic memory allocation pattern from one run to another.

The `info heap high-mem` command displays the maximum number of times the break value changes for a given run. The `set heap-check high-mem-count <X_number>` stops program execution when the break value moves a specified number `<X_number>` of times, and transfers execution control to the user. Both these commands display the size and call site of the last memory allocation that extended the high water mark.

This feature also enables you to analyze the memory-usage in an application and check if the memory-usage is critical or close to triggering an out-of-memory error.Example 10 (page 40) illustrates the high water mark feature.

Example 10 High Water-Mark Feature

Sample Program

```
$cat high.c
1  #include <stdio.h>
2  #include <malloc.h>
3
4  char * func1()
5  {
6    char *ptr;
7    ptr = malloc(1000);
8    return ptr;
9
10 }
11
12 char * func2()
13 {
14  char *ptr;
15  ptr = malloc (100);
16  return ptr;
17
18 }
19
20 void  func4()
21 {
22  char *ptr;
23  ptr = malloc (9000);
24  free (ptr);
25
26 }
27
28 void main()
29 {
30    char* ptr;
31    int i;
32
33    for (i=0; i<100; i++)
34     {
35       ptr = func1();
36       free(ptr);
37     }
38    ptr = func2();
39    func4();
40 }
```

Sample Debugging Session

Case 1: The `info heap high-mem` command displays the number of times that the break value changes for a given run and to display the highest break value in the current run.

```
$ gdb high
HP gdb 5.6 for HP Itanium (32 or 64 bit) and target HP-UX 11.2x.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.6 (based on GDB) is covered
by the GNU General Public License.
Type "show copying" to see the conditions to change it and/or
distribute copies. Type "show warranty" for warranty/support.
..
(gdb) b 40 /*set a breakpoint at the end of your application to view
 the highest break value for the current run*/
 Breakpoint 1 at 0x4000b90:2: file high.c, line 40 from high.
(gdb) set heap-check on
(gdb) r
Starting program: high

Breakpoint 1, main () at high.c:40
```

```
40          };
(gdb) info heap high-mem
Analyzing heap ...

High memory mark stat
High water mark updated count: 2

No.   Total bytes     Blocks     Address     Function
0       100            1       0x4044ff20   func2()
(gdb)
```

Case 2: The `set heap-check high-mem-count <X_number>` command stops execution when the break value has moved `<X_number>` of times.

```
$ gdb high
HP gdb 5.6 for HP Itanium (32 or 64 bit) and target HP-UX 11.2x.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.5.8 (based on GDB) is covered
by the GNU General Public License. Type "show copying" to
see the conditions to change it and/or distribute copies.
Type "show warranty" for warranty/support.
..
(gdb) set heap-check on
(gdb) set heap-check high-mem-count 2
(gdb) r
Starting program: high
warning: High water mark
(address = 0x4044ff00 total memory per call site = 100)
#1  func2() at high.c:15
#2  main() at high.c:40
#3  main_opd_entry() from
warning: Use command backtrace (bt) to see the current context.

Ignore top 4 frames belonging to leak detection library of gdb.

__rtc_event (ecode=RTC_NO_ERROR, pointer=0x0, pclist=0x0, size=0)
    at ../../../Src/gnu/gdb/infrtc.c:1236
1236    {
```

Monitoring De-allocations to Detect Double-Frees

The `set heap-check free <on/off>` command enables you to detect double-frees and frees with improper arguments.

When this command is enabled, the `free()` calls are monitored to verify whether the parameters address valid heap blocks. If an erroneous `free()` is detected, the debugger stops execution and reports the error. You can analyze the stack trace to analyze where and how the error occurred. Example 16 (page 67) illustrates the use of the `set heap-check free` command.

Monitoring Heap Corruption

WDB enables you to detect the presence of heap-corruption in your application. Table 11 lists the commands for monitoring heap corruption

Table 11 Commands for Monitoring Heap Corruption

Command	Description
`set heap-check string <on |off>`	Toggles validation of calls to `strcpy()`, `strncpy()`, `memcpy()`, `memccpy()`,`memset()`, `memmove()`, `bzero()`, and, `bcopy()`
`set heap-check bounds <on|off>`	Toggles the bounds-checking feature for detection of heap-corruption in WDB
`info corruption`	Checks for corruption in the currently allocated heap blocks
`set heap-check scramble <on | off>`	Scrambles a memory block and overwrites it with a specific pattern when it is allocated and de-allocated

Monitoring String Corruption

The `set heap-check string <on/off>` command toggles the string corruption detection feature. It enables you to detect string corruption if functions of the `strcpy()` family write out-of-bounds of the allocated memory. Example 11 (page 43)illustrates the use of the `set heap-check string` command.

This command currently detects string corruption when writing out-of-bounds for `strcpy()`, `strncpy()`,`memcpy()`,`memccpy()`,`memset()`,`memmove()`,`bzero()`,and, `bcopy()` functions.

Example 11 Monitoring heap-corruption caused by erroneous handling of string functions

Sample Program

```
bash-2.05b$ cat string.c
1  #include<stdio.h>
2
3  int main()
4  {
5   char *ptr, *ptr1;
6
7   ptr = (char*)malloc(10);
8   ptr1 = (char *)malloc(20);
9
10  strcpy(ptr, "Hello");
11  strcpy(ptr1, "Welcome to HP WDB");
12
13  memcpy(ptr+5,ptr1,10);
14 }
```

Sample Debugging Session

```
bash-2.05b$ /opt/langtools/bin/gdb string
HP gdb 5.5.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00
and target hppa1.1-hp-hpux11.00.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.5.0 (based on GDB) is
covered by the GNU General Public License.
Type "show copying" to see the conditions to
change it and/or distribute copies.
Type "show warranty" for warranty/support.
..
(gdb) set heap-check on
(gdb) set heap-check string on
(gdb) b main
Breakpoint 1 at 0x2bdc: file string.c, line 7 from string.
(gdb) r
Starting program: string

Breakpoint 1, main () at string.c:7
7          ptr = (char*)malloc(10);
(gdb) c
Continuing.
warning: memcpy corrupted (address = 0x4042a3dd size = 10)
#1  main() at string.c:7
#2  _start() from /usr/lib/libc.2
#3  _start() from /opt/langtools/lib/librtc.sl
#4  $START$() from
warning: Use command backtrace (bt) to see the current context.

Ignore top 4 frames belonging to leak detection library of gdb.

0x70e78d7c in __rtc_event+0 () from /opt/langtools/lib/librtc.sl
(gdb) bt
#0  0x70e78d7c in __rtc_event+0 () from /opt/langtools/lib/librtc.sl
#1  0x70e7cc44 in search_addr+0x64 () from /opt/langtools/lib/librtc.sl
#2  0x70e7cd88 in libc_mem_common+0x130 () from /opt/langtools/lib/librtc.sl
#3  0x70e7ceb8 in memcpy+0x58 () from /opt/langtools/lib/librtc.sl
#4  0x2c54 in main () at string.c:13
#5  0x70ee3478 in _start+0xc0 () from /usr/lib/libc.2
(gdb) c
Continuing.

Program exited normally.
```

Detecting Out-of-Bounds Writes with the Bounds-Checking Feature

The `set heap-check bounds <on/off>` command toggles the bounds-checking feature in WDB. When bounds-checking is enabled, WDB allocates extra space (guard bytes) at the beginning and end of a block during allocation and fills this space with a specific pattern. When the blocks are freed, the debugger verifies if the patterns are intact. If the patterns are corrupted, the debugger detects underflow or overflow errors and reports the corruption. Example 12 (page 45) illustrates the bounds-checking feature.

The bounds checking feature detects overflow and underflow errors only when the write operation occurs within the guard bytes.

Example 12 Bounds-checking to detect out-of-bounds writes

Sample Program

```
bash-2.05b$ cat bounds.c
1  #include<stdio.h>
2
3  int main()
4  {
5     char *cp = (char*)malloc(100);
6     cp[-1] = 100;
7     strcpy(cp,"Hello");
8     cp[100] = 100;
9     free(cp);
10    exit(0);
11 }
```

Sample Debugging Session

```
bash-2.05b$ gdb bounds
HP gdb 5.5.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00
and target hppa1.1-hp-hpux11.00.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.5.0 (based on GDB) is covered
by the GNU General Public License. Type "show copying" to
see the conditions to change it and/or distribute copies.
Type "show warranty" for warranty/support.
..
(gdb) set heap-check bounds on
(gdb) b main
Breakpoint 1 at 0x2c04: file bounds.c, line 5 from bounds.
(gdb) b 10
Breakpoint 2 at 0x2c58: file bounds.c, line 10 from bounds.
(gdb) r
Starting program: bounds

Breakpoint 1, main () at bounds.c:5
5          char *cp = (char*)malloc(100);
(gdb) c
Continuing.
warning: Memory block (size = 100 address = 0x4042a3c8)
appears to be corrupted at the beginning.
Allocation context not found

#1  main() at bounds.c:5
#2  _start() from /usr/lib/libc.2
#3  _start() from /opt/langtools/lib/librtc.sl
#4  $START$() from
warning: Use command backtrace (bt) to see the current context.

Ignore top 4 frames belonging to leak detection library of gdb.

0x70e78d7c in __rtc_event+0 () from /opt/langtools/lib/librtc.sl
(gdb) c
Continuing.
warning: Memory block (size = 100 address = 0x4042a3c8)
appears to be corrupted at the end.
Allocation context not found

#1  main() at bounds.c:5
#2  _start() from /usr/lib/libc.2
#3  _start() from /opt/langtools/lib/librtc.sl
#4  $START$() from
warning: Use command backtrace (bt) to see the current context.

Ignore top 4 frames belonging to leak detection library of gdb.
```

```
0x70e78d7c in __rtc_event+0 () from /opt/langtools/lib/librtc.sl
(gdb) f 4
#4  0x4000960:0 in main () at bounds.c:9
9          free(cp);
(gdb)
```

Detecting Heap Corruption

The `info corruption <filename>` command enables you to view the corruption profile of all the allocations that are corrupted at a specified probe-point in the program. Ensure that the bounds checking is enabled before using the `info corruption` command. The corruption information is written to a specified file if the `<file name>` is provided. Otherwise, it is written to `stdout`.

NOTE: The `info corruption` command is not supported in batch mode debugging

Example 13 (page 48) illustrates the use of the `info corruption` command.

Example 13 Detecting heap corruption using the info corruption command

Sample Program

```
$cat infobounds.c
1  #include <stdio.h>
2  #include <malloc.h>
3
4  char *t;
5  char *t1;
6  char *t2;
7  char *t3;
8
9  char * sm_malloc(sz)
10 int sz;
11  {
12   return malloc(sz); /* line number 12 */
13  }
14
15  main()
16  {
17    t = (char *)sm_malloc(10);
18    strcpy(t, "123456789123");
19    t1 = (char *)sm_malloc(10);
20    strcpy(t1, "12345678912");
21    t2 = (char *)sm_malloc(10);
22    strcpy(t2, "1234567891");
23    t3 = (char *)sm_malloc(10);
24    strcpy(t3, "123456789");
25    printf("Hello\n");
26    free (t);
27    free (t1);
28    free (t2);
29    free (t3);
30    free (t);
31    free (t1);
32    exit(1);
33  }
```

Sample Debugging Session

```
$ gdb infobounds
HP gdb 5.6 for HP Itanium (32 or 64 bit) and target HP-UX 11.2x.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.6 (based on GDB) is covered
by the GNU General Public License. Type "show copying" to
see the conditions to change it and/or distribute copies.
Type "show warranty" for warranty/support.
..
(gdb) b 25
Breakpoint 1 at 0x4000b70:1: file infobounds.c,
line 25 from infobounds.
(gdb) set heap-check on
(gdb) run
Starting program: infobounds

Breakpoint 1, main ()
    at .infobounds.c:25
25          printf("Hello\n");
(gdb) info corruption
Analyzing heap ...

Following blocks appear to be corrupted
No.   Total bytes     Blocks     Address     Function
0        10              1       0x400124e0   sm_malloc()
1        10              1       0x40012500   sm_malloc()
```

```
2        10             1       0x40012520   sm_malloc()
(gdb) info corruption 2
10 bytes at 0x40012520 (33.33% of all bytes allocated)
#0  sm_malloc() at infobounds.c:12
#1  main() at infobounds.c:21
#2  main_opd_entry() from /usr/lib/hpux32/dld.so
(gdb)
```

Scrambling a Heap Block

The `set heap-check scramble <on/off>` command enables you to scramble a heap block and overwrite it with a specific pattern ("`0xfeedface`") when it is allocated or de-allocated.

If the application continues to use (read) a freed block (incorrect memory usage), the application fails to find the expected data in the block. (This means that the data in the block is different from the initial data that was written in the block.)

This increases the chances of the application to crash or result in unpredictable program behavior sooner with the unexpected data that is read from the block. Additionally, you can detect this condition with assertion checks in the code to validate the read data during the further run of the application.

This command does not detect the corruption. It is only a minimal aid to detect corruption.

Example 14 (page 50) illustrates the scramble feature in WDB.

Example 14 Scrambling a memory block on de-allocation

Sample Program

```
$ cat scramble.c
1  #include <stdio.h>
2  #include <malloc.h>
3
4  int
5  main ()
6  {
7     char **tp;
8     tp = malloc (100);
9     printf ("Batch RTC test over, *tp=%p.\n", *tp);
10    fflush(stdout);
11    free(tp);
12    exit (0);
13 }
```

Sample Debugging Session

```
$ gdb scramble
HP gdb 5.5 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00
and target hppa1.1-hp-hpux11.00.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest 5.5.8 (based on GDB) is covered
by the GNU General Public License. Type "show copying" to
see the conditions to change it and/or distribute copies.
Type "show warranty" for warranty/support.
..
(gdb) set heap-check scramble on
(gdb) b main
Breakpoint 1 at 0x295c: file scramble.c, line 8 from scramble.
(gdb) r
Starting program: scramble

Breakpoint 1, main () at scramble.c:8
8           tp = malloc (100);
(gdb) n
9           printf ("Batch RTC test over, *tp=%p.\n", *tp);
(gdb) p *tp
$1 = 0xfeedface <Error reading address 0xfeedface: Bad address>
```

Settings to Manage Performance Degradation.

Memory-debugging slows down the performance of an application by 20-40% because of stack unwinding. Reducing the number of stack frames the debugger collects for each allocation reduces the performance degradation.

Table 12 lists the options for reducing the performance degradation.

Table 12 Options for Performance Improvement

Setting	Command	Description
Stack Depth	`set heap-check frame-count <num>`	Controls the depth of the call stack. By default, `num` is set to 4.
Minimum Leak Size	`set heap-check min-leak-size <num>`	Specifies the minimum leak size for stack trace collection. The debugger continues to report leaks that are smaller than `<num>` bytes, but it does not provide the stack trace for the same. By default, `num` is set to 0.

Supported Modes of Memory-debugging in WDB

WDB supports the following modes of memory-debugging:

- Interactive Mode
- Batch Mode
- Attach Mode

Debugging in the Interactive Mode

The interactive mode of memory-debugging is typically useful during the development and defect fixing phase, where you need the flexibility to control the flow of program execution while debugging memory related problems.

To debug your program in the interactive mode, complete the following steps:

1. Compile the source files with the –g option. No special compilation of link options are required.

   The program must be linked with shared `libc.[so|sl]`. Memory-checking features do not work on the programs linked with archived `libc.a`

   ```
   $ aCC –g <source filename> –o <executable>
   ```

2. To activate the memory debugging, perform either of the following:
   - Invoke WDB with the `-leaks` option as follows:

     ```
      $ gdb -leaks <executable>
     ```

     This enables leak checking. To enable other memory debugging features you must use the appropriate set of commands.

   - Alternatively, enter the following command at the `gdb` prompt:

     ```
     $ gdb <executable>
     (gdb)set heap-check on
     ```

     This enables leaks checking, bounds checking, and check for double-frees.

3. Place breakpoints at probe points by entering the following command:

   ```
   (gdb)b <probe_point>
   ```

4. To generate a leak profile at the breakpoint, enter the following command:

   ```
   (gdb)info leaks <filename>
   ```

5. To generate a snapshot heap profile at the breakpoint, enter the following command:

   ```
   (gdb) info heap <filename>
   ```

Debugging in Batch Mode

In this mode, the user does not interactively issue commands in a debugger session. Instead, the memory-debugging commands are stored in a user-specified configuration file. The configuration file gets processed during the run of the application and at the end of the program the debugger creates output data files for that run. It creates three separate output files for leak profile, heap profile, and the memory corruption reports.

Batch mode memory-debugging stops the application at the end of the program when `exit()` is called or when all the statically linked libraries (including `librtc.[sl|so]`) are unloaded. After the application is stopped, it invokes the debugger to print the leak or heap data.

Following is the naming convention for the output files:

```
<file_name>.<pid>.<suffix>
```

Where:

`<pid>` is the process id and `<suffix>` can be either `leaks`, `heap`, or `mem` based on the type of report.

For example: `memtest.8494.mem`

Environment Variables for Batch Memory-Debugging

This section discusses the environment variables that must be set for using the batch mode of memory debugging.

Enabling and Disabling Batch Mode Memory-Debugging

The environment variable, `BATCH_RTC`, must be configured to enable and disable batch mode memory-debugging.

Following is the syntax for enabling and disabling batch mode debugging:

```
export BATCH_RTC=<on/off>
```

Pre-loading the Appropriate Version of librtc.[sl|so] Along With the Application

The appropriate version of the `librtc.[sl|so]` runtime library must be preloaded to enable batch mode and attach mode memory debugging of an application.

You can explicitly preload `librtc.[sl|so]` from the appropriate path by using the `LD_PRELOAD` environment variable.

Alternately you can use the `+mem_check <enable|disable>` option for the `chatr` command to automatically preload `librtc.[sl|so]`. Both of these methods are illustrated in this section.

NOTE: The `+mem_check <enable>` option for the `chatr` command is available for dynamic linker versions `B.11.61` and later on HP 9000 systems, and dynamic linker versions `B.12.46` and later on Integrity systems.

Using chatr +mem_check to Automatically Preload librtc.[sl|so]

To automatically preload `librtc.[sl|so]` by using the `+mem_check <enable|disable>` for the `chatr` command, enter the following command at the HP-UX prompt:

```
$ chatr +mem_check <enable> <executable>
```

In addition to automatically loading `librtc.[sl|so]`, the `+mem_check` option for the `chatr` command also maps the shared libraries as private. The `+mem_check` option preloads `librtc.[sl|so]` from the following default paths for `librtc.[sl|so]` :

- - For 32 bit IPF applications

`/opt/langtools/lib/hpux32/librtc.so`

- For 64 bit IPF applications,

  `/opt/langtools/lib/hpux64/librtc.so`

- For 32 bit PA applications,

  `opt/langtools/lib/librtc.sl`

- For 64-bit PA applications,

  `/opt/langtools/lib/pa20_64/librtc.sl`

NOTE: To preload from a path that is different from the default paths, you must use the `LD_PRELOAD` environment variable.

Using LD_PRELOAD TO Preload librtc.[sl|so]

To explicitly preload an appropriate version of `librtc.[sl|so]` with the application, set the environment variable, `LD_PRELOAD` as follows:

- For 32-bit applications running on Itanium,

  `LD_PRELOAD=/opt/langtools/lib/hpux32/librtc.so <executable> <arguments>`

- For 64-bit applications running on Itanium,

  `LD_PRELOAD=/opt/langtools/lib/hpux64/librtc.so <executable> <arguments>`

- For 32-bit applications running on PA-RISC,

  `LD_PRELOAD=/opt/langtools/lib/librtc.sl <executable> <arguments>`

- For 64-bit applications running on PA-RISC,

  `LD_PRELOAD=/opt/langtools/lib/pa20_64/librtc.sl <executable> <arguments>`

NOTE: If `LD_PRELOAD` and `chatr +mem_check` are used to preload the `librtc.[sl|so]` runtime library, `librtc[sl|so]` is loaded from the path specified by `LD_PRELOAD`.

Overriding the Default Location for librtc.[sl|so]

By default WDB uses the `librtc.[sl|so]`, available at the location `/opt/langtools/lib`. If the application requires the use of `librtc.[sl|so]` at a different location, you must set the environment variable, `LIBRTC_SERVER`, to point to the location where the library is located.

Following is the syntax for setting an alternate location for `librtc.[sl|so]`:

```
export LIBRTC_SERVER=<path>
```

NOTE: This environmental variable is applicable for attach and interactive mode also.

Overriding the Default Path for Searching the GDB Executable

The `GDB_SERVER` variable enables you to override the default path from where the `gdb` executable is used to debug memory problems. The default path for the `gdb` executable is `/opt/langtools/bin/gdb`.

Following is the syntax to override the default path for the `gdb` executable:

```
export GDB_SERVER=<path>
```

NOTE: Batch Mode RTC displays one of the following errors and causes the program to temporarily hang if the version of GDB and `librtc.[sl|so]` do not match, or if GDB is not available on the system:

```
/opt/langtools/bin/gdb: unrecognized option '-brtc' Use '/opt/langtools/bin/gdb --help' for a complete list of
 options.
```

(OR)

```
execl failed. Cannot print RTC info: No such file or directory
```

This error does not occur under normal usage where GDB or `librtc.[sl|so]` is used from the default location at `/opt/langtools/...`. However, this error occurs if `GDB_SERVER` ,`LIBRTC_SERVER`, or both are set to a mismatched version of GDB or `librtc.[sl|so]` respectively.

Configuration File for Batch Mode Debugging

You can set your preferences for batch mode memory-debugging by setting the parameters in the configuration file. The following sections discuss the location of the configuration file and the supported variables.

Location of the Configuration File for Batch Mode Debugging

The configuration file `rtcconfig` for batch mode debugging is user-defined. You must either create the `rtcconfig` file in the current directory or specify the location of the configuration file by exporting the environment variable `GDBRTC_CONFIG` to contain the pathname of the configuration file (including the filename.)

The following example illustrates how to specify the path of the configuration file:

```
$ export GDBRTC_CONFIG=/tmp/rtcconfig
```

If the path to the configuration file is not specified, the debugger assumes that the `rtcconfig` configuration file, by default, is located in the current working directory.

Supported Variables for Memory-Debugging in the Batch Mode Configuration File

You must specify the variables in the configuration file based on the commands that are required to debug the application. Table 13 lists the variables that are supported in the configuration file.

Table 13 Supported Variables in the Batch Mode Configuration File

Command	Description
`set heap-check free <on/off>`	Enables you to detect double-frees and frees with improper arguments
`set heap-check <on/off>`	Enables heap profiling
`set heap-check leaks <on/off>`	Enables you to detect leaks
`set heap-check string <on/off>`	Enables validation of calls to `strcpy()`, `strncpy()`, `memcpy()`, `memccpy()`,`memset()`, `memmove()`, `bzero()`, and, `bcopy()`
`set heap-check bounds <on/off>`	Enables you to check for out-of-bounds corruption when the block is freed
`files=<file1:file2:..fileN>`	Enables you to specify the executables for which memory leak detection is enabled. If the files option is not specified (after setting `BATCH_RTC=on`) , the debugger checks all the executables.
`set heap-check frame-count <num>`	Enables you to set the number of frames to be collected for leak or heap profiles

Table 13 Supported Variables in the Batch Mode Configuration File *(continued)*

Command	Description
`set heap-check min-heap-size <num>`	Enables you to set the minimum block size to use for heap reporting
`set heap-check min-block-size <num>`	Enables you to set the minimum block size to use for leak reporting
`output_dir= <output_data_dir_path>`	Enables you to specify the name of the output data directory
`set heap-check scramble <on/off>`	Enables you to scramble the blocks

NOTE: It is incorrect to use spaces before or after the '=' symbol in the batch mode configuration options in the configuration file, rtcconfig. Additionally, it is incorrect to use spaces before the batch mode configuration options.

For example:

Correct Usage:

```
$ cat rtcconfig
check_leaks=on
check_heap=on
files=batchrtc4
```

Incorrect Usage:

```
$ cat rtcconfig
 check_leaks=on
check_heap = on
files=batchrtc4
```

Overriding the Configuration File Settings

The `RTC_MALLOC_CONFIG` variable enables you to override the default `rtcconfig` file settings.

Following is the syntax for exporting the configuration to `RTC_MALLOC_CONFIG`:

```
export RTC_MALLOC_CONFIG=config_string1[;config_strings]
```

The overriding settings of `RTC_MALLOC_CONFIG` are dependent on the global environment variable `RTC_NO_ABORT` setting. `RTC_NO_ABORT` must not be set if the configuration strings must abort the execution of the program on detection of the first occurrence of bounds, double-free, or out-of-memory conditions.

If `RTC_NO_ABORT` is set to `1`, the program does not abort for failed checks and you can view the logfiles for all the failed checks in a single execution run.

Table 14 lists the `config_strings` options that are available for `RTC_MALLOC_CONFIG`. The `config_strings` are separated by semicolon (;).

Table 14 The config_strings Options for RTC_MALLOC_CONFIG

config_string Options	Description
`abort_on_bounds=[01]`	`RTC_NO_ABORT` must not be set. If `abort_on_bounds` is set to `1`, the batch mode aborts execution of the program and reports the bounds condition, when bound checking fails.
`abort_on_bad_free=[01]`	`RTC_NO_ABORT` must not be set. If `abort_on_bad_free` is set to `1`, the batch mode aborts execution when a `free()`, or a `realloc()` call attempts to free a heap object that is not valid.
`abort_on_nomem=[01]`	`RTC_NO_ABORT` must not be set. If `abort_on_nomem` is set to `1`, the batch mode aborts execution when an out-of-memory condition is detected.

`mem_logfile=stderr[+]filename`
`heap_logfile=stderr[+]filename`
`leak_logfile=stderr[+]filename`
	The appropriate logfiles for the heap, leak, and corruption detection are displayed on `stderr`. The logfiles are directed to the specified file `<filename>`. Output is appended to the file if the + option is used.

Debugging in Batch Mode

To debug an application in the batch mode, complete the following steps:

1. Compile the source files.

NOTE: On HP 9000 systems, you must map the shared libraries as private, by using the `chatr` command if you are using `LD_PRELOAD` to preload the `librtc.[sl|so]` instead of the `+mem_check <enable|disable>` option for the `chatr` command .

```
chatr +dbg enable ./<executable>
```

2. Set the required variables in the `rtcconfig` configuration file, as follows:

```
$ cat rtcconfig
"rtcconfig" 5 lines, 76 characters
set heap-check on
set heap-check free on
files=executable_name
output_dir= ./
```

3. Set the required environment variables as follows:

```
export BATCH_RTC=on
```

4. You can use the `+mem_check <enable|disable>` option for the `chatr` command to automatically preload `librtc.[sl|so]` or you can explicitly preload `librtc.[sl|so]` from the appropriate path by using the `LD_PRELOAD` environment variable.

NOTE: The `+mem_check <enable>` option for the `chatr` command is available for dynamic linker versions `B.11.61` and later on HP 9000 systems, and dynamic linker versions `B.12.46` and later on Integrity systems.

To preload the `librtc.[sl|so]` runtime library, complete one of the following steps:

- To automatically preload `librtc.[sl|so]` by using the `+mem_check <enable|disable>` for the `chatr` command, enter the following command at the HP-UX prompt:

  ```
  $ chatr +mem_check <enable> <executable>
  ```

  NOTE: To preload from a path , which is different from the default path, you must use the `LD_PRELOAD` environment variable.

- Set the environment variable, `LD_PRELOAD` as follows:
  - For 32-bit applications running on Itanium,
    ```
    LD_PRELOAD=/opt/langtools/lib/hpux32/librtc.so <executable> <arguments>
    ```
  - For 64-bit applications running on Itanium,
    ```
    LD_PRELOAD=/opt/langtools/lib/hpux64/librtc.so <executable> <arguments>
    ```
  - For 32-bit applications running on PA-RISC,
    ```
    LD_PRELOAD=/opt/langtools/lib/librtc.sl <executable> <arguments>
    ```
  - For 64-bit applications running on PA-RISC,
    ```
    LD_PRELOAD=/opt/langtools/lib/pa20_64/librtc.sl <executable> <arguments>
    ```
    If `LD_PRELOAD` and `chatr +mem_check` are used to preload the `librtc.[sl|so]` runtime library, `librtc[sl|so]` is loaded from the path specified by `LD_PRELOAD`.

NOTE: If the application invokes calls such as `system(3s)`, and `popen()`, which invoke a new shell, `librtc.[sl|so]` must not be loaded to the invoked shell. You must use `LD_PRELOAD_ONCE`, instead of `LD_PRELOAD`, to exclusively load the`librtc.[sl|so]` file to the calling process only.

Following is the syntax for using `LD_PRELOAD_ONCE`:

```
LD_PRELOAD_ONCE= /opt/langtools/lib/librtc.sl
```

Example 15 (page 58) illustrates the batch mode debugging of the `memtest.c` program. The debugging results are stored in `memtest.8494.mem`, `memtest.8494.heap`, and `memtest.8494.leaks`.

Example 15 Batch Mode Debugging for a 32-bit Application running on Itanium

Sample Program

```
$ cat memtest.c
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>

#include <signal.h>
void myhandler(void ) {
exit(0);
}

main(int argc)
{

  // signal(11, myhandler);

       char *p[20], buffer[]="0123456789012345", *temp;
       int i;

       for (i=0; i<20; i++) {
               p[i]=(char *)malloc(4);
        };
        memcpy(p[15], buffer, 12);
        for (i=0; i<10; i++) {
                free(p[i]);
        };
        free(p[9]);
}

```

Sample Configuration File

```
$ cat rtcconfig
"rtcconfig" 7 lines, 86 characters
set heap-check on
set heap-check leaks on
set heap-check free on
set heap-check string on
files=memtest
output_dir=./
```

Sample Debugging Session

```
$ cc -g -o memtest memtest.c

$ export BATCH_RTC=on

$ chatr +mem_check enable  memtest
warning: Memory corruption info is written to "memtest.8494.mem".
warning: Memory leak info is written to "memtest.8494.leaks".
         Memory heap info is written to "memtest.8494.heap


$ cat memtest.8494.mem
-----------------------------------------
Attempt to free unallocated or already freed object at 0x4006e7b0
(0)  0x60000000cac1bdc0  print_stack_trace_to_log_file + 0x1d0 at
     ../../../Src/gnu/gdb/infrtc.c:996
    [/opt/langtools/lib/hpux32/librtc.sl]
(1)  0x60000000cac1d3e0  __rtc_event + 0x160 at
      ../../../Src/gnu/gdb/infrtc.c:1296
     [/opt/langtools/lib/hpux32/librtc.sl]
(2)  0x60000000cac22da0  rtc_record_free + 0x380 at
     ../../../Src/gnu/gdb/infrtc.c:2651
     [/opt/langtools/lib/hpux32/librtc.sl]
(3)  0x60000000cac17200  __rtc_free + 0x160 at
     ../../../Src/gnu/gdb/infrtc.c:2977
     [/opt/langtools/lib/hpux32/librtc.sl]
(4)  0x0000000004000bc0  main + 0x230 at memtest.c:25[memtest]
(5)  0x60000000c0029000  main_opd_entry + 0x50[/usr/lib/hpux32/dld.so]



$ cat .//memtest.8494.leaks

40 bytes leaked in 10 blocks

No.   Total bytes     Blocks     Address     Function
0        40             10      0x4006e7d0   main()


------------------------------------------------------------
                              Detailed Report

------------------------------------------------------------
40 bytes leaked in 10 blocks (100.00% of all bytes leaked)
These range in size from 4 to 4 bytes and are allocated
#0  main() at memtest.c:19
#1  main_opd_entry() from /usr/lib/hpux32/dld.so

------------------------------------------------------------


$ cat memtest.8494.heap

40 bytes allocated in 10 blocks

No.   Total bytes     Blocks     Address     Function
0        40             10      0x4006e8f0   main()


------------------------------------------------------------
                              Detailed Report

------------------------------------------------------------
40 bytes in 10 blocks (100.00% of all bytes allocated)
These range in size from 4 to 4 bytes and are allocated
#0  main() at memtest.c:19
#1  main_opd_entry() from /usr/lib/hpux32/dld.so

------------------------------------------------------------
```

Debugging Multiple Applications in Batch Mode

To debug multiple applications in the batch mode, complete the following steps:

1. Compile the source files.
2. Set the required variables in the `rtcconfig` configuration file, as follows:

   ```
   $ cat rtcconfig
   "rtcconfig" 5 lines, 83 characters
   set heap-check on
   set heap-check free on
   files=exec1:exec2:exec3
   output_dir= ./
   ```

3. Set the required environment variables as follows:

   ```
   export BATCH_RTC=on
   ```

4. Complete one of the following steps to preload `librtc.[sl|so]`:
   - Use the `+mem_check` option for the `chatr` command on each of the required executable files that must be instrumented, as follows:

     ```
     $ chatr +mem_check enable exec1 exec2 exec3
     ```

     The `+mem_check <enable>` option for the `chatr` command is available for dynamic linker versions `B.11.61` and later on HP 9000 systems, and dynamic linker versions `B.12.46` and later on Integrity systems.

     (Or)
   - Preload `librtc.[sl|so]` for all the executables, as follows:
     — For 32-bit applications running on Itanium,

     ```
     LD_PRELOAD=/opt/langtools/lib/hpux32/librtc.so exec1 exec2 exec3
     ```

     — For 64-bit applications running on Itanium,

     ```
     LD_PRELOAD=/opt/langtools/lib/hpux64/librtc.so exec1 exec2 exec3
     ```

     — For 32-bit applications running on PA-RISC,

     ```
     LD_PRELOAD=/opt/langtools/lib/librtc.sl exec1 exec2 exec3
     ```

     — For 64-bit applications running on PA-RISC,

     ```
     LD_PRELOAD=/opt/langtools/lib/pa20_64/librtc.sl exec1 exec2 exec3
     ```

NOTE: If `exec1` eventually spawns `exec2`, `exec3`, `exec4`, and `exec5`, only `exec1`, `exec2` and `exec3` are debugged based on the settings in the `rtcconfig` file.

Debugging in Attach Mode

WDB can attach to a running process and debug memory problems. However to use the debugger in this mode, the application must be launched after preloading the `librtc.[sl|so]` runtime library.

To debug memory on attaching GDB to a running process, complete the following steps:

1. You can use the `+mem_check <enable|disable>` option for the `chatr` command to automatically preload `librtc.[sl|so]` or you can explicitly preload `librtc.[sl|so]` from the appropriate path by using the `LD_PRELOAD` environment variable.

   To preload the `librtc.[sl|so]` runtime library, complete one of the following steps:

   - To automatically preload `librtc.[sl|so]` by using the `+mem_check <enable|disable>` for the `chatr` command, enter the following command at the HP-UX prompt:

     ```
     $ chatr +mem_check <enable> <executable>
     ```

     NOTE: To preload from a path that is different from the default paths, you must use the `LD_PRELOAD` environment variable.

   - Set the environment variable, `LD_PRELOAD` as follows:
     - For 32-bit applications running on Itanium,

       ```
       LD_PRELOAD=/opt/langtools/lib/hpux32/librtc.so <executable> <arguments>
       ```
     - For 64-bit applications running on Itanium,

       ```
       LD_PRELOAD=/opt/langtools/lib/hpux64/librtc.so <executable> <arguments>
       ```
     - For 32-bit applications running on PA-RISC,

       ```
       LD_PRELOAD=/opt/langtools/lib/librtc.sl <executable> <arguments>
       ```
     - For 64-bit applications running on PA-RISC,

       ```
       LD_PRELOAD=/opt/langtools/lib/pa20_64/librtc.sl <executable> <arguments>
       ```

     If `LD_PRELOAD` and `chatr +mem_check` are used to preload the `librtc.[sl|so]` runtime library, `librtc[sl|so]` is loaded from the path specified by `LD_PRELOAD`.

2. Identify the required process (using the `ps` command) and attach the debugger to the process as follows.

   ```
   gdb -leaks <executable-name> <process-id>
   ```

3. Insert breakpoints at suitable probe-points. When the breakpoints trigger, use the `info heap` and `info leaks` commands to display the heap and leak profile.

NOTE: To attach and find leaks for PA-32 applications from the startup, the environment variable `RTC_INIT` must be set to `on` in addition to preloading the `librtc.[sl|so]` library before starting the application, as follows:

```
$ LD_PRELOAD=/opt/langtools/lib/librtc.sl RTC_INIT=on <executable>
```

If `RTC_INIT` is enabled, `librtc.[sl|so]` starts recording heap information for PA32 process by default. Hence, you must set this environment variable only when it is required. You must not export the `RTC_INIT` environment variable for shell.

Summary of Memory Debugging Commands

Most of the commands available in the interactive and the attach modes are also available in the batch mode. Table 15 "Commonly Used Commands for Memory Debugging" lists the commands that are available in the batch mode and the equivalent commands in the interactive mode. It also lists the commands that are not supported in the batch mode.

Table 15 Commonly Used Commands for Memory Debugging

Description	Interactive Mode/Attach Mode	Batch Mode
Enables heap profiling	`set heap-check <on/off>`	`set heap-check <on/off>`
Enables you to detect leaks.	`set heap-check leaks <on/off>`	`set heap-check leaks <on/off>`
Enables you to detect double-frees and frees with improper arguments	`set heap-check free <on/off>`	`set heap-check free <on/off>`
Enables you to scramble blocks.	`set heap-check scramble <on/off>`	`set heap-check scramble <on/off>`
Enables you to check for out-of-bounds corruption when the block is freed.	`set heap-check bounds <on/off>`	`set heap-check bounds <on/off>`
Enables validation of calls to `strcpy()`,`strncpy()`,`memcpy()`, `memccpy()`,`memset()`,`memmove()`, `bzero()`, and, `bcopy()`	`set heap-check string <on/off>`	`set heap-check string <on/off>`
Enables you to set the number of frames to be printed for leak and heap profiles.	`set heap-check frame-count <num>`	`set heap-check frame-count <num>`
Enables you to set the minimum block size to report in heap profiles.	`set heap-check min-heap-size <num>`	`set heap-check min-heap-size <num>`
Enables you to set the minimum block size to use for leak detection.	`set heap-check min-leak-size <num>`	`set heap-check min-leak-size <num>`
Stops program execution when the block at the given address is allocated or de-allocated	`set heap-check watch address`	Not supported in Batch Mode
Checks for corruption in the currently allocated heap blocks	`info corruption`	Not supported in Batch Mode
Forces `malloc()` to return `NULL` after `<N>` invocations of `malloc()`	`set heap-check null-check <num>`	Not supported in Batch Mode
Forces `malloc()` to return `NULL` after `<N>` bytes are allocated by the program	`set heap-check null-check-size <size>`	Not supported in Batch Mode
Enables the user to gain control over an out-of-memory event. The user can step through program execution after the `nomem` event is detected.	`catch nomem`	Not supported in Batch Mode
Defines the seed-value for random number calculation for the `set heap-check null-check random` command	`set heap-check seed-value <num>`	Not supported in Batch Mode
Forces `malloc()` to return `NULL` after random number of invocations of `malloc()`	`set heap-check random-range <num>`	Not supported in Batch Mode
This command resets the data existing in the file for incremental profiling and creates a new data file. The old data in the file is erased.	`set heap-check reset`	Not supported in Batch Mode

Table 15 Commonly Used Commands for Memory Debugging *(continued)*

Description	Interactive Mode/Attach Mode	Batch Mode
Starts the incremental heap growth profile. All allocations prior to the execution of this command are ignored. If incremental heap growth profile is already on, executing this command resets the counters and starts a fresh collection. The interval is specified in seconds.	`set heap-check interval <num>`	Not supported in Batch Mode
Enables you to specify the number of intervals for which WDB must collect the incremental heap growth. The default value is 100. Every repeat of the interval tracks heap allocation during that interval.	`set heap-check repeat <num>`	Not supported in Batch Mode
Creates a detailed report of the heap growth. The data for each interval has the start and end time of the interval.	`info heap interval`	Not supported in Batch Mode
Stops when break value has moved `<X_number>` times	`set heap-check high-mem-count <X_number>`	Not supported in Batch Mode
Displays the number of times break value changes for a given run	`info heap high-mem`	Not supported in Batch Mode
Displays the high level memory-usage of a process or an arena. Lists the number of free blocks, used blocks, small blocks, holding blocks, node blocks and regular blocks.	`info heap <process|arenas>`	Not supported in Batch Mode

Debugging Memory Using WDB GUI

The WDB GUI is a Graphical User Interface (GUI) designed by Hewlett-Packard for WDB. It can be used to debug native-compiled HP C, HP aC++, and Fortran programs on Itanium-based systems running HP-UX 11i v2 or HP-UX 11i v3, and PA-RISC systems running HP-UX 11.0, HP-UX 11i v1, HP-UX 11i v2, or HP-UX 11i v3.

The WDB GUI offers the following capabilities to debug memory–related errors in an application

- Detects corruption caused by calls to `strcpy()`, `memset()`, and `memcopy()`
- Stops program-execution at free of an unallocated or de-allocated block address
- Stops program-execution when block is freed if bad writes occur before or after block bounds
- Scrambles previous memory contents on `malloc()` or `free()` calls
- Stops if the following block address is allocated or de-allocated
- Stops program execution when an allocation causes heap growth exceeding `<num>` bytes
- Collects memory leak data (equivalent to info leak) and memory-usage (equivalent to `info heap`) data

WDB GUI supports both interactive and attach mode of memory-debugging. It does not support the batch mode debugging of applications.

The source window in the WDB GUI displays the source code. The command window displays the output after debugging the application. It also enables you to use the command-line interface in WDB. The command window can be used to take advantage of memory-debugging features that are not directly supported in the GUI.

Using WDB GUI to Debug Memory-Related Problems

To debug an application for memory problems using WDB GUI, complete the following steps:

1. Load the program to WDB as follows:
   - Select **File —> Load Program** in the WDB GUI window.
   - In the **Load Program** dialog-box, enter the executable name to load the executable or use the PID to attach a process for debugging.
2. After you load the application, you can set the memory checking preferences by setting the preferences in the **Memory Checking** window. Select **Tools —>Memory Checking** to activate the **Memory Checking** window.
3. To set a break-point using WDB GUI, click the rectangular selection strip adjacent to the specific program line-number in the source window. When the breakpoint is set, a red octagonal button appears at the specified probe-point. Alternatively, you can set breakpoints by selecting **Edit —>Breakpoints** and specifying the breakpoints in the **Breakpoints** window.
4. Run the application. WDB GUI now gives you the leaks usage, memory-usage and the results of memory checking at the specified break-points.

Heap and Leak Profiling Using WDB GUI

In order to view the heap report and leak reports while debugging the application, select the **Memory Usage** tab in the command window. On selecting the **Memory Usage** tab, the **Memory Leaks** and **Memory Usage** options are displayed.

To view the leak report, select the **Memory Leaks** option. The stack-unwind information for each leak can be obtained by expanding the enhanced array browser for each leak.

To see a heap report, select the **Memory Usage** option. The stack-unwind information can be obtained by expanding the enhanced array browser for each block of the heap.

Incremental Heap Profiling Using WDB GUI

HP WDB GUI provides support to view the incremental heap profile for a program.

To view the incremental heap profile for a program, complete the following steps:

1. Load the program to WDB GUI.
2. Select **Tools->Memory Check**
3. Select the **Incremental Heap Check Settings** option while setting the memory debugging preferences in the **Memory Check** window.
4. Enter **Heap Check Interval** and **Heap Check Repeat Count** in the **Memory Check** window.
5. Run the program after setting the required breakpoints.
6. Select **View->Memory Usage->Incremental Heap** to view the incremental heap profile.
7. The **Incremental Heap View** window displays the incremental heap profile graph for the program

   The incremental heap profile graph can be plotted based on the outstanding allocations in the program or the actual heap profile, as follows:

   - To view the incremental heap profile graph based on the outstanding allocations, select the **Allocation** option in the **Plot Graph** frame. The **Allocation Profile** displays the outstanding allocations (in KB) in the program with a unique color coding for each interval.
   - To view the incremental heap profile graph based on the actual heap profile, select the **Actual Heap** option in the **Plot Graph** frame. The **Heap Space Profile** displays the heap size in KB for the program.

   To specify the time interval for displaying the incremental heap profile, you must select the **Select Time** option and specify the start time and the end time from the **Start Time** list menu and the **End Time** list menu. The listings for time in the **Start Time** list menu and the **End Time** list menu are calculated by dividing the total program execution time into five equal intervals. Additionally, you can enter a custom start time, or a custom end time for displaying the incremental heap profile.

To view the incremental heap profile summary, click **Summary Table**. The summary table displays the record ID, the start time, the end time, the heap interval, the heap start, the heap end, the heap size in bytes, the number of allocated bytes, and the number of blocks used for all the collected incremental heap profile records. Click on the required incremental heap profile record to view the block allocation details for the corresponding record.

Arena Profiling Using WDB GUI

HP WDB GUI 5.7 and later versions provide support to view the arena information for a program running on HP-UX 11i v3.

To view the arena information, complete the following steps:

1. Load the program to HP WDB GUI.
2. Stop the program execution at the required breakpoints.
3. Select **View->Memory Usage->Heap Arena** to view the arena information. The arena information is displayed in the **View Heap Arena** window.

The following information is displayed in the **View Heap Arena** window:

- **Arenas**

  The Arena IDs are listed in the **Arenas** list menu. Select the required Arena ID from the **Arenas** list menu to view **Arena ID Summary**, or **Block Details** for the selected arena.

- **Arena ID Summary**

  The summary information for the selected Arena ID is displayed in the **Arena ID Summary** frame.

- **Block Details**

  To view the block level details in an arena, select Block Details after selecting the required Arena ID in the **Arenas** list menu. The block distribution in the arena is displayed in the **Arenas Block Distribution** window.

  The **Arenas Block Distribution** window displays the block level space distribution graph for an arena. The graph displays the space occupied by the user blocks, the free blocks, the unclaimed space, and the `malloc` metadata (which includes the node blocks, the cached blocks, the holding header blocks, and the holding SBA blocks). The virtual address of the blocks is used to arrange the blocks in the graph. The Block ID of the block is also displayed within the block if the scale of the graph supports the display. The start of the heap, the end of the heap, the total heap size, and the total number of blocks are also listed.

  WDB GUI displays the block distribution graph in the default window. If the complete block distribution graph cannot be displayed in the default window, you must select the **Expand Block Distribution Graph** toggle option to view the magnified block distribution graph.

  To view the Block ID, the block type, the block size, and the virtual address for each block in the arena, you must click on the required block in the block distribution graph. The number of used blocks in each block-count range is also displayed graphically for the selected arena. This information is also displayed in a tabular format.

- **Heap Arena Space Usage**

  The **Heap Arena Space Usage** frame displays the Arena ID, the space usage (in KB) in the arena, and the percentage space usage in each arena in comparison to the total space occupied by all the arenas. The comparative space usage across arenas is also displayed in a pie chart. The Arena ID is also displayed in the pie chart if the scale of the graph supports it.

- **Full Summary**

  To view the summary for all the arenas, click **Full Summary**. The summary information for all the arenas is displayed in the Arena Summary window.

- **Heap Arena Detailed Graphs**

  The **Heap Arena Detailed Graphs** display the following information:

  The byte distribution (in KB) across the used ordinary blocks, the used small blocks, the free ordinary blocks, and the free small blocks is displayed for each arena. This information is displayed as a bar graph for each arena.

  The byte distribution for the used ordinary blocks, the used small blocks, the free ordinary blocks, and the free small blocks across the arenas is displayed in a pie chart. This information is also displayed in a table.

  The number of blocks that are distributed across the used ordinary blocks, the used small blocks, the free ordinary blocks, and the free small blocks are displayed for each arena. This information is displayed as a bar graph for each arena.

  The number of blocks occupied by the used ordinary blocks, the used small blocks, the free ordinary blocks, and the free small blocks across the arenas are displayed in a pie chart. This information is also displayed in a table.

Conclusion

Memory-related errors are some of the most difficult programming errors to detect and debug. Debugging memory-related errors is difficult without the help of an effective memory analysis tool. WDB enables you to debug memory leaks and heap-related errors in an application

In addition to plugging memory leaks in your application, it is also important to track the memory utilization in your application. WDB provides capabilities such as heap profiling and error injection to analyze the memory-usage of your application. The heap profile displays information about the allocated memory, the calling function, and it also displays the allocating call stack.

Additional Examples

Example 16 to Example 21 illustrate how WDB detects memory leaks and heap-errors caused by different types of programming errors.

Example 16 Detecting a double free error

Sample Program

```
$ cat double-free.c
#include<stdio.h>
1
2 int main()
3 {
4
5   printf("Starting program\n");
6   char* han = (char*)malloc(sizeof(char));
7   free(han);
8   printf("Now freeing a pointer twice...\n");
9   free(han);
10 }
```

Sample Debugging Session

```
(gdb) set heap-check free on
(gdb) file double-free
Reading symbols from double-free...done.
(gdb) b main
Breakpoint 1 at 0x2be4: file double-free.c,
                line 5 from double-free.
(gdb) r
Starting program: /double-free

Breakpoint 1, main () at double-free.c:5
5         printf("Starting program\n");
(gdb) n
Starting program
6         char* han = (char*)malloc(sizeof(char));
(gdb)
7         free(han);
(gdb)
8         printf("Now freeing a pointer twice...\n");
(gdb)
Now freeing a pointer twice...
9         free(han);
(gdb)
warning: Attempt to free unallocated or already freed
         object at 0x4042c3e0
0x70e78d7c in __rtc_event+0 ()
            from /opt/langtools/lib/librtc.sl
```

Example 17 Detecting de-allocation of memory that has not been initialized

Sample Program

```
$ cat unalloc.c
1 #include<stdio.h>
2
3 int main() {
4
5  printf("Starting program\n");
6  char* han;
7  free(han);
8 }
```

Sample Debugging Session

```
gdb) set heap-check on
(gdb) file unalloc
Reading symbols from unalloc...done.
(gdb) b main
Breakpoint 1 at 0x2bb4: file unalloc.c, line 5 from unalloc.
(gdb) r
Starting program: unalloc

Breakpoint 1, main () at unalloc.c:5
5         printf("Starting program\n");
(gdb) n
Starting program
7         free(han);
(gdb)
warning: Attempt to free unallocated or already freed
         object at 0x70fee070
0x70e78d7c in __rtc_event+0 ()
            from /opt/langtools/lib/librtc.sl
```

Example 18 Detecting de-allocation of un-allocated blocks

Sample Program

```
$ cat unit.c
1 #include<stdio.h>
2
3 int main() {
4
5  printf("Starting program\n");
6  int *han = (int*)malloc(sizeof(int));
7  han++;
8  free(han);
9 }
```

Sample Debugging Session

```
(gdb) set heap-check on
(gdb) file uninit
Reading symbols from uninit...done.
(gdb) b main
Breakpoint 1 at 0x2bdc: file uninit.c, line 5 from uninit.
(gdb) r
Starting program: uninit

Breakpoint 1, main () at uninit.c:5
5         printf("Starting program\n");
(gdb) n
Starting program
6         int *han = (int*)malloc(sizeof(int));
(gdb)
7         han++;
(gdb)
8         free(han);
(gdb)
warning: Attempt to free unallocated or already freed
         object at 0x4042c3e4
0x70e78d7c in __rtc_event+0 () from /opt/langtools/lib/librtc.sl
```

Example 19 Detecting memory leaks that are caused when an application overwrites a pointer that currently addresses a block of memory with another address or data

Sample Program

```
$ cat memleak1.c
1 #include<stdio.h>
2
3 int main() {
4
5  printf("Starting program\n");
6  int* han1 = (int*)malloc(sizeof(int));
7  int* han2 = (int*)malloc(sizeof(int));
8  han1 = han2;
9  free(han1);
10 }
```

Sample Debugging Session

```
(gdb) set heap-check on
(gdb) file memleak1
Reading symbols from memleak1...done.
(gdb) b main
Breakpoint 1 at 0x2bdc: file memleak1.c, line 5 from memleak1.
(gdb) r
Starting program: memleak1

Breakpoint 1, main () at memleak1.c:5
5         printf("Starting program\n");
(gdb) n
Starting program
6         int* han1 = (int*)malloc(sizeof(int));
(gdb)
7         int* han2 = (int*)malloc(sizeof(int));
(gdb)
8         han1 = han2;
(gdb)
9         free(han1);
(gdb)
10      }
(gdb) info leak
Scanning for memory leaks...


4 bytes leaked in 1 blocks

No.   Total bytes     Blocks     Address     Function
0       4               1       0x4042c3e0   main()
(gdb) info leak 0
4 bytes leaked at 0x4042c3e0 (100.00% of all bytes leaked)
#0  main() at memleak1.c:6
#1  _start() from /usr/lib/libc.2
#2  _start() from /opt/langtools/lib/librtc.sl
#3  $START$() from
```

Example 20 Detecting memory leaks that are caused when a pointer variable in an application addresses memory that is out of the scope of the application

Sample Program

```
$ cat memleak2.c
1  #include<stdio.h>
2
3  void func1(int* ptr1)
4  {
5  ptr1 = (int*)malloc(5*sizeof(int));
6  }
7
8  void func2(int** ptr)
9  {
10  func1(*ptr);
11 }
12 int main()
13
14 {
15  printf("Starting program\n");
16  int* han1;
17  func2(&han1);
18  printf("End of the program\n");
19 }
```

Sample Debugging Session

```
(gdb) set heap-check on
(gdb) file memleak2
Reading symbols from memleak2...done.
(gdb) b 19
Breakpoint 1 at 0x2c4c: file memleak2.c, line 19 from memleak2.
(gdb) r
Starting program: memleak2
Starting program
End of the program

Breakpoint 1, main () at memleak2.c:19
19      }
(gdb) info leak
Scanning for memory leaks...


20 bytes leaked in 1 blocks

No.   Total bytes     Blocks     Address     Function
0        20             1       0x4042c3d8   func1()
(gdb) info leak 0
20 bytes leaked at 0x4042c3d8 (100.00% of all bytes leaked)
#0  func1() at memleak2.c:5
#1  func2() at memleak2.c:10
#2  main() at memleak2.c:17
#3  _start() from /usr/lib/libc.2
```

Example 21 Detecting memory leaks when you free a structure or an array that has pointers which are not freed.

Sample Program

```
$ cat memleak3.c
1 #include<stdio.h>
2
3 struct stud
4 {
5 char* name;
6 int   id;
7 };
8
9 int main() {
10
11 struct stud *s1;
12  s1 = (struct stud*)malloc(sizeof(struct stud));
13  s1->name = (char*)malloc(50);
14  strcpy(s1,"Annie");
15  s1->id=10;
16  free(s1);
17 }
```

Sample Debugging Session

```
(gdb) set heap-check on
(gdb) file memleak3
Reading symbols from memleak3...done.
(gdb) b 17
Breakpoint 1 at 0x2c3c: file memleak3.c, line 17 from memleak3.
(gdb) r
Starting program:memleak3
Breakpoint 1, main () at memleak3.c:17
17      }
(gdb) info leak
Scanning for memory leaks...
50 bytes leaked in 1 blocks
No.   Total bytes     Blocks     Address     Function
0       50             1       0x4042a3e0   main()
(gdb) info leak 0
50 bytes leaked at 0x4042a3e0 (100.00% of all bytes leaked)
#0  main() at memleak3.c:13
#1  _start() from /usr/lib/libc.2
#2  _start() from /opt/langtools/lib/librtc.sl
#3  $START$() from usr/lib/libc.2
```

Example 22 Work-Around when program execution is in a frame that belongs to the GDB internal leak detection library

```
...
(gdb) set heap-check on
(gdb) r
Starting program: corruption
warning: Memory block (size = 80 address = 0x40453970) appears to be corrupted at the end.
Allocation context not found

#1  main() at corruption.c:4
#2  main_opd_entry() from
warning: Use command backtrace (bt) to see the current context.

Ignore top 4 frames belonging to leak detection library of gdb.

__rtc_event () at ../../../Src/gnu/gdb/infrtc.c:1173
warning: Source file is more recent than library library librtc.so.

1173    */
(gdb) bt
#0  __rtc_event () at ../../../Src/gnu/gdb/infrtc.c:1173
#1  0x200000007d0fbd40:0 in check_bounds (pointer=0x40453970, size=80,
     pclist=0x404309e4) at ../../../Src/gnu/gdb/infrtc.c:1278
#2  0x200000007d100f50:0 in rtc_record_free ()
     at ../../../Src/gnu/gdb/infrtc.c:2261
#3  0x200000007d1025a0:0 in free () at
../../../Src/gnu/gdb/infrtc.c:2575
#4  0x4000950:0 in main () at corruption.c:10
(gdb) info corruption
Analyzing heap ...

Current thread is inside the allocator. Try again later.

(gdb) frame 3
#3  0x200000007d1025a0:0 in free () at
../../../Src/gnu/gdb/infrtc.c:2575
2575              __rtc_event (RTC_HEAP_GROWTH, pointer,0,0);
(gdb) finish
Run till exit from #3  0x200000007d1025a0:0 in free ()
     at ../../../Src/gnu/gdb/infrtc.c:2575
0x4000950:0 in main () at corruption.c:10
10        free (x);
(gdb) info corruption
Analyzing heap ...
```

FAQ

1 Does WDB report all the leaks in a program?

WDB uses a conservative leak detection algorithm. As a result, all leaks may not be reported, but all reported leaks are definite leaks. WDB reports leaks only in the code path exercised in the current run.

2 I wrote a small sample program that allocates a block using `malloc()` and leaks the block immediately, by assigning `NULL` to the pointer, but WDB does not report this block as a leak. Why?

This is attributed to the leak detection algorithm followed by WDB. If the datum in the program address space masks a leak, the leak is not reported. In this case the address returned from `malloc()` is stored in the architecture registers and consequently masks the leak. Typically, if you call any function after the leak, such as a `printf()`, then WDB can catch the leak.

3 Does WDB support detection of leaks in a third party code?

Yes. WDB supports detection of leaks in a third party code also.

4 What are the commands that the batch mode of memory-debugging does not support?

For more information on the commands that are supported in batch mode, see "Summary of Memory Debugging Commands " (page 61)

5 Can WDB debug applications with user-defined memory management routines?

WDB can debug applications with memory management routines that are either user defined or are wrappers to the default memory management routines.

NOTE:

- This feature is not supported in the batch and the attach modes of debugging.
- In interactive mode, this feature will result in calls to the user defined memory management routines being re-routed to default memory management routines.

6 Which version of WDB supports debugging of applications with custom allocators?

WDB 5.5 and above versions support the debugging of applications with custom allocators.

7 Does WDB report the exact instant when the block becomes a leak?

No. WDB does not provide information on when the leak occurred. It reports only the allocation stack trace of the leaked block and does not report the stack trace where the block leaked.

8 Does WDB support debugging of C++ applications with calls to `new()` and `delete()`?

Yes. WDB supports debugging of C++ applications with `new()` and `delete()` calls only if they internally call `malloc()` and `free()`.

9 Does WDB support memory-debugging of long running applications?

Yes. WDB supports debugging of long running applications such as daemons. However, the daemons must be started with an explicit `LD_PRELOAD` of the correct version (32-bit or 64-bit) of `librtc.[sl|so]`, so that WDB can debug memory when it later attaches to the daemon process.

10 What is the work-around when the following message is displayed, when attempting to view the leak report with the `info leaks` command?

```
(gdb) info leaks
Current thread is blocked. Cannot detect leaks now.
```

You can switch execution to a thread, which is not blocked. To switch execution to a different thread, enter the following command at the gdb prompt:

```
(gdb)thread <thread-id>
```

11 Does the debugger find leaks in the executable from the startup of the application when debugging the application in attach mode?

In the case of Itanium binaries, and PA—RISC 64–bit binaries, the debugger finds leaks in the executable from the startup of the executable by default, when debugging in attach mode.

However, to find leaks in the executable from the startup of PA-RISC 32–bit binaries in the attach mode, the environment variable `RTC_INIT` must be set to `on` in addition to preloading the `librtc.[sl|so]` library before starting the application, as follows:

```
$ LD_PRELOAD=/opt/langtools/lib/librtc.sl RTC_INIT=on <executable>
```

If `RTC_INIT` is enabled, `librtc.[sl|so]` starts recording heap information for PA–RISC 32–bit process by default. Hence, you must set this environment variable only when memory debugging is required from the startup of the program

12 When attempting to view the leak report, the following error occurs:

```
(gdb) info leaks
Scanning for memory leaks...

Error downloading data !
(gdb)
```

What is the cause for this error and what is the work-around?

This error message is displayed when you attempt to view the heap profile or the leak profile of a debugged process, which is exiting or has exited program execution. As a work-around, you can place a breakpoint before the program exits and then enter the `info leaks` command or the `info heap` command.

13 What is the work-around if the following error message is displayed while debugging memory?

```
(gdb) info corruption
Current thread is inside the allocator. Try again later.
```

This error message signifies that the program execution is in a frame that belongs to a GDB internal leak detection library. When this error is encountered, it is not safe to enter commands that involve calls to the leak detection library procedures. The user must set the frame to the last leak detection library frame and enter the `finish` command before resuming to debug memory.

Example 22 (page 73) illustrates the use of the `finish` to resume memory debugging when the program execution is in a frame that belongs to the GDB internal leak detection library.

Index