

Debugging Memory Problems with MemoryScape™

Version 2024.4
November, 2024

PERFORCE

www.perforce.com



© 2024 Perforce Software, Inc. All rights reserved.
© 2007-2024 by Rogue Wave Software, Inc., a Perforce company ("Rogue Wave"). All rights reserved.
© 1998-2007 by Etnus LLC. All rights reserved.
© 1996-1998 by Dolphin Interconnect Solutions, Inc.
© 1993-1996 by BBN Systems and Technologies, a division of BBN Corporation.

Perforce and other identified trademarks are the property of Perforce Software, Inc., or one of its affiliates. Such trademarks are claimed and/or registered in the U.S. and other countries and regions. All third-party trademarks are the property of their respective holders. References to third-party trademarks do not imply endorsement or sponsorship of any products or services by the trademark holder. Contact Perforce Software, Inc., for further details.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of Rogue Wave.

Perforce has prepared this manual for the exclusive use of its customers, personnel, and licensees. The information in this manual is subject to change without notice, and should not be construed as a commitment by Perforce. Perforce assumes no responsibility for any errors that appear in this document.

TotalView and TotalView Technologies are registered trademarks of Rogue Wave. TVD is a trademark of Rogue Wave.

Perforce uses a modified version of the Microline widget library. Under the terms of its license, you are entitled to use these modifications. The source code is available at <https://rwkbp.makekb.com/>.

All other brand names are the trademarks of their respective holders.

ACKNOWLEDGMENTS

Use of the Documentation and implementation of any of its processes or techniques are the sole responsibility of the client, and Perforce Software, Inc., assumes no responsibility and will not be liable for any errors, omissions, damage, or loss that might result from any use or misuse of the Documentation.

ROGUE WAVE MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THE DOCUMENTATION. THE DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. ROGUE WAVE HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE DOCUMENTATION, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT SHALL PERFORCE SOFTWARE, INC. BE LIABLE, WHETHER IN CONTRACT, TORT, OR OTHERWISE, FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT, PUNITIVE, OR EXEMPLARY DAMAGES IN CONNECTION WITH THE USE OF THE DOCUMENTATION.

The Documentation is subject to change at any time without notice.

TotalView by Perforce
<http://totalview.io>

Contents

Checking for Problems	2
Programs and Memory	4
Behind the Scenes	8
Your Program's Data	9
The Data Section	9
The Stack	9
The Heap	13
Finding Heap Allocation Problems	13
Finding Heap Deallocation Problems	13
realloc() Problems	13
Finding Memory Leaks	14
Starting MemoryScape	16
Using MemoryScape Options	17
Preloading MemoryScape	18
Understanding How Your Program is Using Memory	19
Finding free() and realloc() Problems	21
Event and Error Notification 21	
Types of Problems	23
Freeing Stack Memory	23
Freeing bss Data	23
Freeing Data Section Memory	23
Freeing Memory That Is Already Freed	23
Tracking realloc() Problems	24
Freeing the Wrong Address	24
Finding Memory Leaks	25
Fixing Dangling Pointer Problems	28
Dangling Pointers	29
Batch Scripting and Using the CLI	31

Batch Scripting Using tvscript 31	
Using the dheap Command	31
dheap Example	32
Notification When free Problems Occur	32
Showing Backtrace Information: dheap -backtrace:	33
Guarding Memory Blocks: dheap -guards	33
Memory Reuse: dheap -hoard	34
Writing Heap Information: dheap -export	35
Filtering Heap Information: dheap -filter	35
Checking for Dangling Pointers: dheap -is_dangling:	36
Detecting Leaks: dheap -leaks	37
Block Painting: dheap -paint	37
Red Zones Bounds Checking: dheap -red_zones	38
Deallocation Notification: dheap -tag_alloc	42
TVHEAP_ARGS	42
Examining Memory	45
Block Properties	47
Memory Contents Tab	49
Additional Memory Block Information	50
Filtering 51	
Using Guard Blocks 51	
Using Red Zones	53
Using Guard Blocks and Red Zones	54
Block Painting 54	
Hoarding	56
Example 1: Finding a Multithreading Problem	56
Example 2: Finding Dangling Pointer References	56
Debugging with TotalView	58
Starting MemoryScape	61
Adding Programs and Files to MemoryScape	65
Attaching to Programs and Adding Core Files	66
Stopping Before Finishing Execution	66
Exporting Memory Data	66

MemoryScape Information	67
Where to Go Next	67
Basic Options	71
Advanced Options	74
Halt execution at process exit (standalone MemoryScape only)	75
Halt execution on memory event or error	75
Guard allocated memory	77
Use Red Zones to find memory access violations	78
Restricting Red Zones	79
Customizing Red Zones	79
Paint memory	80
Hoard deallocated memory	80
Where to Go Next	81
Controlling Program Execution from the Home Summary Screen	85
Controlling Program Execution from the Manage Processes Screen	85
Controlling Program Execution from a Context Menu	85
Where to Go Next	85
Information Types	86
Process and Library Reports	87
Chart Report	87
Where to Go Next	89
Error Notifications	90
Deallocation and Reuse Notifications	92
Where to Go Next	93
Window Sections	94
Block Information	96
Bottom Tabbed Areas	96
Where to Go Next	96
Heap Status Source Report	98
Heap Status Source Backtrace Report	101
Where to Go Next	101
Adding, Deleting, Enabling and Disabling Filters	103
Adding and Editing Filters	104
Where to Go Next	107
Examining Corrupted Memory Blocks	108
Viewing Memory Contents	110
Procedures for Exporting and Adding Memory Data	111

Using Saved State Information	111
Where to Go Next	112
Overview	113
Obtaining a Comparison	113
Memory Comparison Report	114
Where to Go Next	115
Saving Report Information	117
Using Remote Display	120
Compiling Programs	122
Linking with the dbfork Library	123
dbfork on IBM AIX on RS/6000 Systems	123
Linking C++ Programs with dbfork	123
dbfork and Linux or Mac OS X	124
dbfork and SunOS 5 SPARC	124
Ways to Start MemoryScape	125
Attaching to Programs	126
Setting Up MPI Debugging Sessions	127
Debugging MPI Programs	127
Debugging MPICH Applications	129
Starting MemoryScape on an MPICH Job	129
Attaching to an MPICH Job	130
Using MPICH P4 procgroup Files	130
Starting MPI Issues	131
Debugging IBM MPI Parallel Environment (PE) Applications	131
Using Switch-Based Communications	132
Performing a Remote Login	132
Starting MemoryScape on a PE Program	132
Attaching to a PE Job	133
Debugging LAM/MPI Applications	133
Debugging QSW RMS Applications	134
Starting MemoryScape on an RMS Job	134
Attaching to an RMS Job	134
Debugging Sun MPI Applications	134
Attaching to a Sun MPI Job	134

Linking Your Application with the Agent	136
Using env to Insert the Agent	139
Installing tvheap_mr.a on AIX	140
LIBPATH and Linking	140
Using MemoryScape in Selected Environments	142
MPICH	142
IBM PE	142
RMS MPI	142
Mac OS	143
Background	143
Calls to system() on Mac OS	143
Setting the Environment Variable TV_MACOS_SYSTEM	143
Linux	144
dlopen and RTLD_DEEPBIND	144
display_specifiers Command-Line Option	146
event_action Command-Line Option	147
Other Command Line Options	149
memscript Example	149
Invoking MemoryScape	150
Syntax	150
Options	150

Locating Memory Problems

It's frequently stated that 60% or 70% of all programming errors are memory related. So, while these numbers could be wrong, let's assume that they are right. While algorithmic errors often show themselves easily, memory errors are far more subtle. You can reproduce an algorithmic error with a pre-defined set of steps. In contrast, memory errors are seldom reproducible in this way. Worse, the problems they cause happen randomly, and may not occur every time a program is run.

Why are there so many memory errors? There are many answers. The primary reason is that programs are complicated, and the way in which memory should be managed isn't clear. Is a library function allocating its own memory, or should the program allocate it? Once it is allocated, does your program manage the memory or does the library? Something creates a pointer to something, then the memory is freed without any knowledge that something else is pointing to it. Or, and this is the most prevalent reason, a wide separation exists between lines of code, or the time when old code and new code was written. And, of course, there's always insufficient and inaccurate documentation.

In addition, some apparent problems might be irrelevant. If the program does not free the memory allocated for a small array, it doesn't mean much. Or, it can be more efficient not to free the memory since the operating system will free it for you when the program ends. On the other hand, if the program continually allocates memory without freeing it, it will eventually crash because there is no more memory available.

Checking for Problems

MemoryScape can help you locate many of your program's memory problems. For example, you can:

- **Stop execution when `free()`, `realloc()`, and other heap API problems or actions occur**

For example, if your program tries to free memory that it either can't or shouldn't free, MemoryScape can stop execution. This lets you quickly identify the statement causing the problem. For more information, see [“Finding `free\(\)` and `realloc\(\)` Problems”](#) on page 21.

Or, after identifying a memory block, you can tell MemoryScape to stop execution when your program either deallocates or reallocates it. For more information, see [“Deallocation and Reuse Notifications”](#) on page 92.

- **List leaks**

MemoryScape can display reports of your program's leaks—*leaks* are memory blocks that are allocated and are no longer referenced; that is, the program no longer can access the memory block, which means that memory is not available for any other use.

When your program allocates a memory block, MemoryScape creates and records a backtrace for the block—a *backtrace* is a list of stack frames. At a later time when you ask MemoryScape to display leak information, MemoryScape also displays this backtrace. This means that you'll instantly know where your program allocated the memory block. For more information, see [“Finding Memory Leaks”](#) on page 25.

- **Locate memory written beyond the bounds of an allocation**

MemoryScape can allocate memory immediately before and after the blocks your program allocates. When MemoryScape adds these blocks—called *guard blocks*—it also initializes them to a bit pattern. If this bit pattern changes, MemoryScape sees the change when your program deallocates the block, and stops execution.

MemoryScape can also check all guard blocks at any time during program execution. For more information, see [“Using Guard Blocks”](#) on page 51.

Another option is to use Red Zones, additional pages of memory allocated before or after your block. MemoryScape can apply Red Zones to your allocated blocks. If MemoryScape detects read or write access in the Red Zone outside the bounds of your allocated block, it halts program execution and notifies you of the underrun or overrun. For more information see [“Using Red Zones”](#) on page 53.

- **Paint allocated and deallocated blocks**

When your program's memory manager allocates or deallocates memory, MemoryScape can write a bit pattern into it. Writing this bit pattern is called *painting*.

If your program tries to dereference a pointer through painted memory, it may crash. Fortunately, MemoryScape will trap the action before your program dumps core, allowing you to see where the problem occurs. For more information, see [“Block Painting”](#) on page 54.

- **Hold on to deallocated memory**

When you are trying to identify memory problems, holding on to memory after your program releases it can sometimes help locate problems by forcing a memory error to occur at a later time. Holding onto freed memory is called *hoarding*. For more information, see [“Hoarding”](#) on page 56.

Programs and Memory

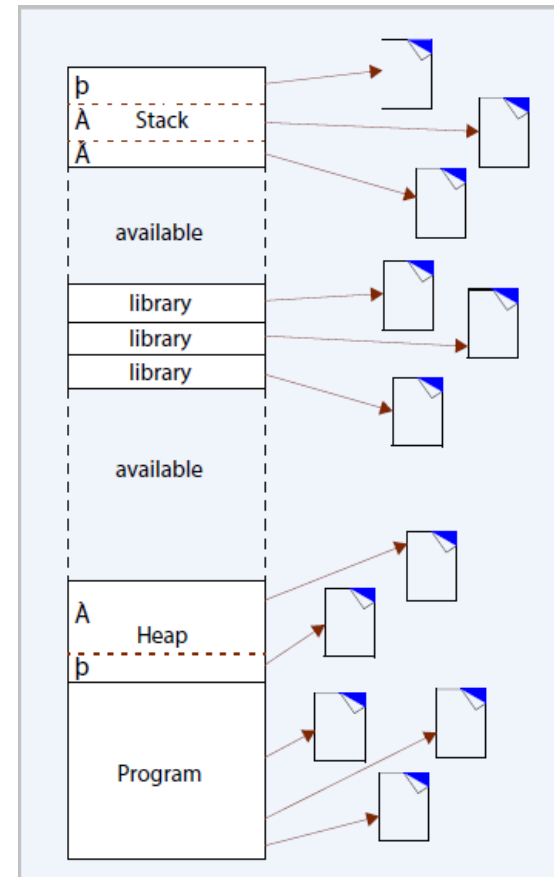
When you run a program, your operating system loads the program into memory and defines an address space in which the program can operate. For example, if your program is executing in a 32-bit computer, the address space is approximately 4 gigabytes.

NOTE >> This discussion is generally relevant to most computer architectures. For information specific to your system, check your vendor documentation.

An operating system does not actually allocate the memory in this address space. Instead, operating systems *memory map* this space, which means that the operating system relates the theoretical address space your program could use with what it actually will be using. Typically, operating systems divide memory into pages. When a program begins executing, the operating system creates a map that correlates the executing program with the pages that contain the program's information. [Figure 1](#) shows regions of a program where arrows point to the memory pages that contain different portions of your program.

[Figure 1](#) also shows a stack containing three stack frames, each mapped to its own page.

Figure 1: Mapping Program Pages



Similarly, the heap shows two allocations, each mapped to its own page. (This figure vastly simplifies actual memory mapping, since a page can have many stack frames and many heap allocations.)

The program did not emerge fully formed into this state. It had to be compiled, linked, and loaded. **Figure 2** shows a program whose source code resides in four files.

Running these files through a compiler creates object files. A linker then merges these object files and any external libraries needed into a load file. This load file is the executable program stored on your computer's file system.

When the linker creates the load file, it combines the information contained in each of the object files into one unit. Combining them is relatively straightforward. The load file at the bottom of **Figure 2** also details this file's contents, as this file contains a number of sections and additional information. For example:

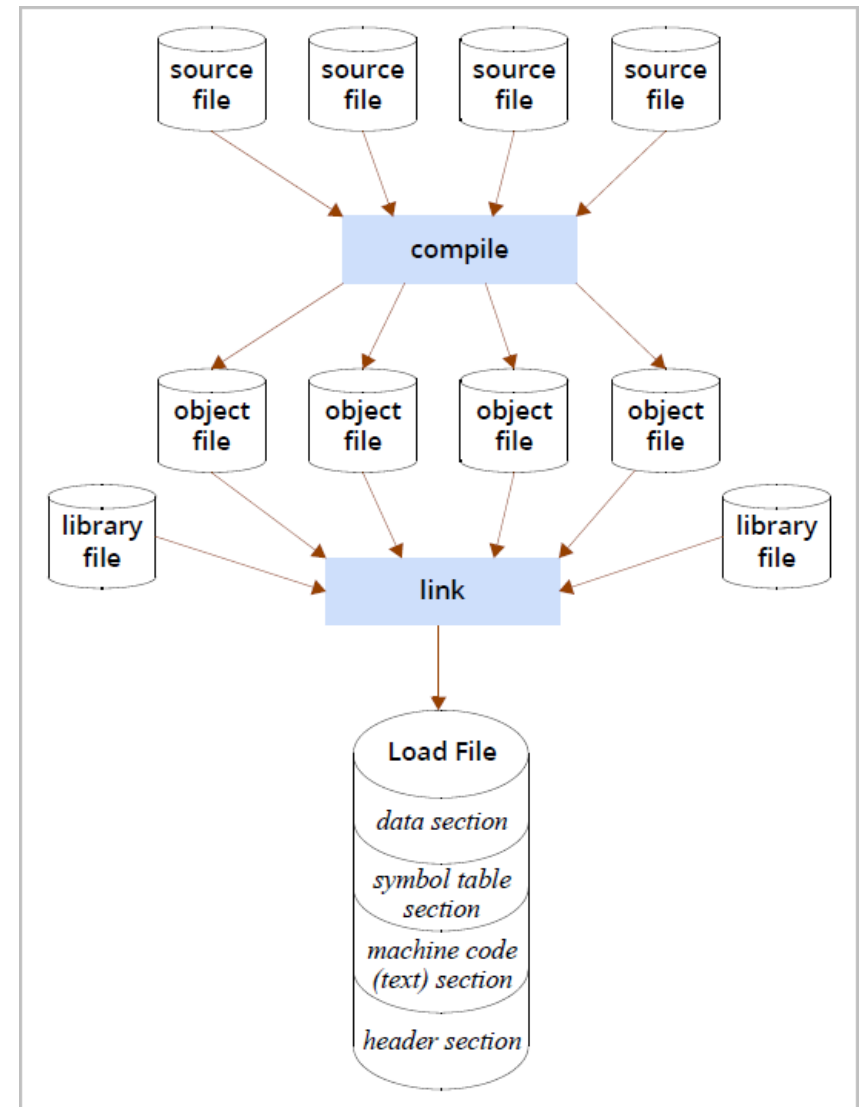
- **Data section**—contains static variables and variables initialized outside of a function. Here is a small sample program that shows these initializations:

```
int my_var1 = 10;
void main ()
{
    static int my_var2 = 1;
    int my_var3;
    my_var3 = my_var1 + my_var2;
    printf("here's what I've got: %i\n", my_var3);
}
```

The data section contains the **my_var1** and **my_var2** variables. In contrast, the memory for the **my_var3** variable is dynamically and automatically allocated and deallocated within the stack by your program's runtime system.

- **Symbol table section**—contains addresses (usually offsets) to the locations of routines and variables.

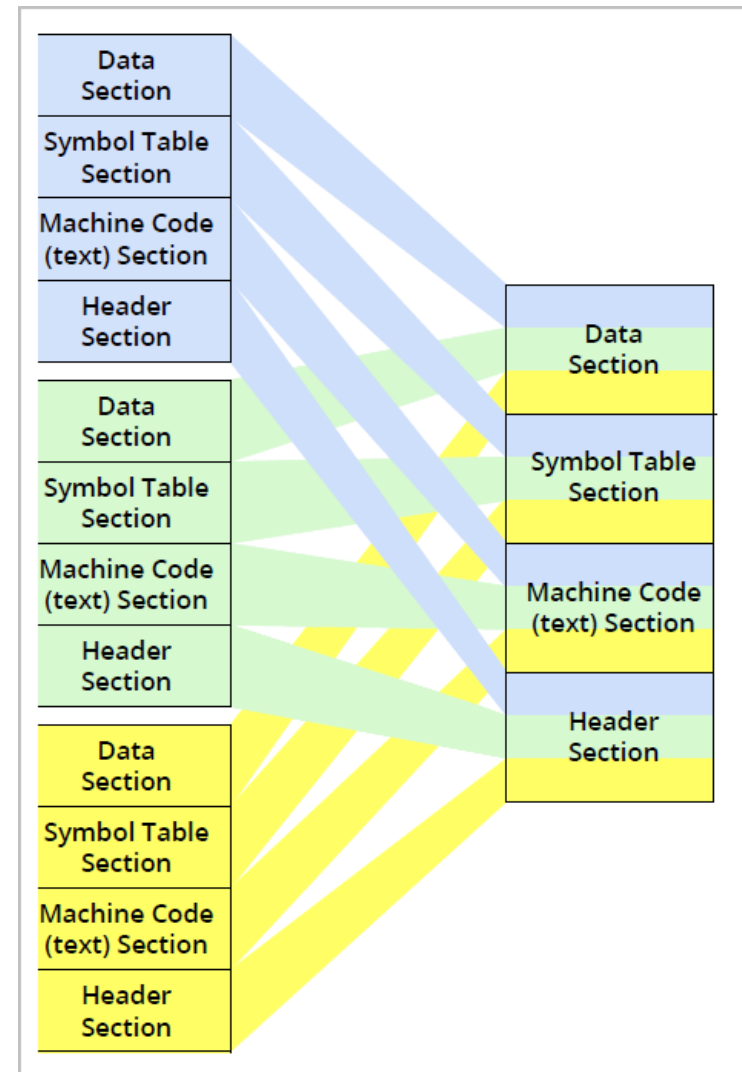
Figure 2: Compiling Programs



- **Machine code section**—contains an intermediate binary representation of your program. (It is intermediate because the linker has not yet resolved the addresses.)
- **Header section**—contains information about the size and location of information in all other sections of the object file.

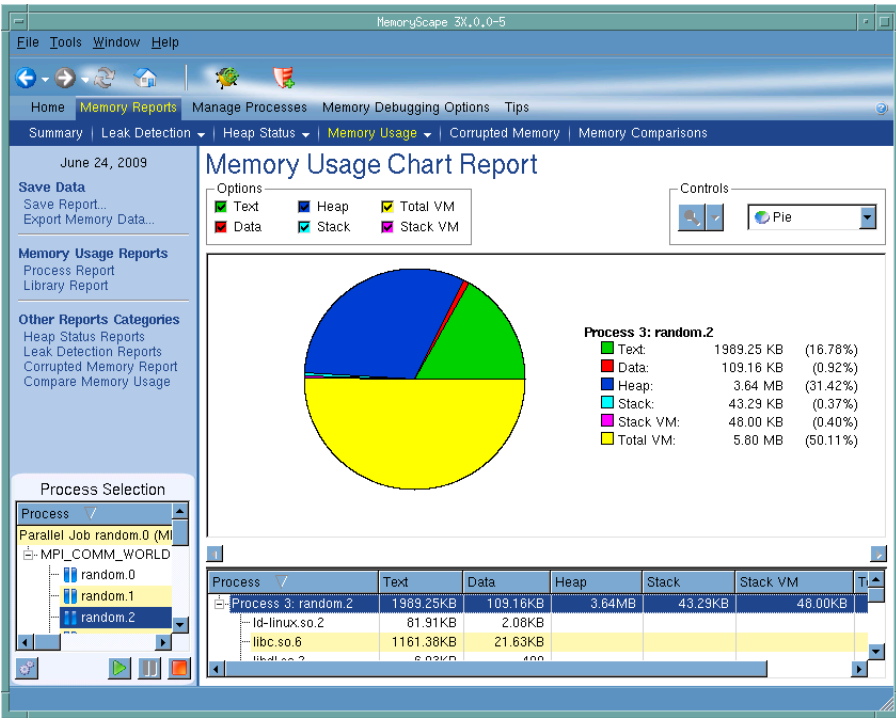
When the linker creates the load file from the object and library files, it interweaves these sections into one file. The linking operation creates something that your operating system can load into memory. [Figure 3](#) shows this process.

Figure 3: Linking a Program



MemoryScope can provide information about these sections and the amount of memory your program is using. To obtain this information, use the **Memory Reports | Memory Usage** and select **Chart report**, [Figure 4](#).

Figure 4: Memory Usage Chart Report



While there are other memory usage reports, the Chart Report provides a concise summary of how your program is using memory.

For information, see [Task 5: "Seeing Memory Usage"](#) on page 86.

Behind the Scenes

MemoryScape intercepts calls made by your program to heap library functions that allocate and deallocate memory by using the **malloc()** and **free()** functions and related functions such as **calloc()** and **realloc()**. The technique it uses is called *interposition*. MemoryScape's interposition technology uses an agent routine to intercept calls to functions in this library. This agent routine is sometimes called the **Heap Interposition Agent (HIA)**.

You can use MemoryScape with any allocation and deallocation library that uses such functions as **malloc()** and **free()**. Typically, this library is called *the malloc library*. For example, the C++ **new** operator is almost always built on top of the **malloc()** function. If it is, MemoryScape can track it. Similarly, if your Fortran implementation use **malloc()** and **free()** functions to manage memory, MemoryScape can track Fortran heap memory use.

You can interpose the agent in two ways:

- You can tell MemoryScape to preload the agent. *Preloading* means that the loader places an object before the object listed in the application's loader table.

When a routine references a symbol in another routine, the linker searches for that symbol's definition. Because the agent's routine is the first object in the table, your program invokes the agent's routine instead of the routine that was initially linked in.

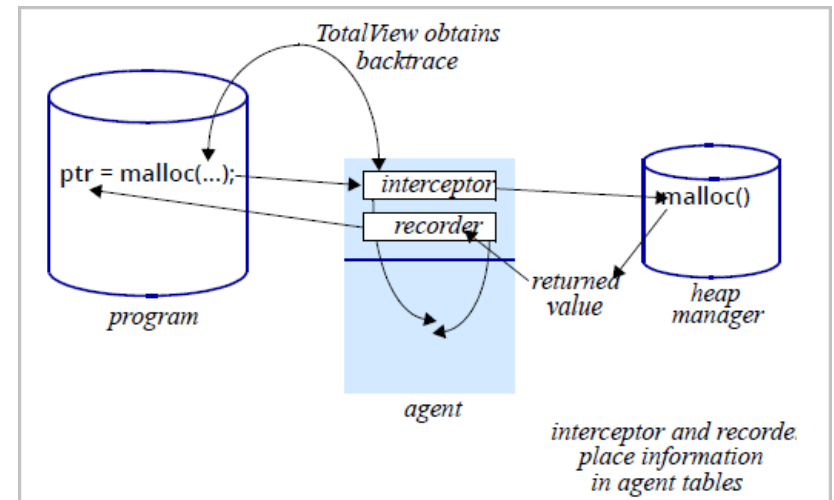
On Linux and Sun, MemoryScape sets an environment variable that contains the pathname of the agent's shared library. For more information, see "[Using env to Insert the Agent](#)" on page 139.

- If MemoryScape cannot preload the agent, you will need to explicitly link it into your program. For details, see [Creating Programs for Memory Debugging](#), on page 121.

If your program attaches to an already running program, you must explicitly link this other program with the agent.

After the agent intercepts a call, it calls the original function. This means that you can use MemoryScape with most memory allocators. [Figure 5](#) shows how the agent interacts with your program and the heap library.

Figure 5: Interposition



Because MemoryScape uses interposition, memory debugging can be considered non-invasive. That is, MemoryScape doesn't rewrite or augment your program's code, and you don't have to do anything in your program. Because the agent lives in the user space, it will add a small amount of overhead to the program's behavior, but it should not be significant.

Your Program's Data

Your program's data resides in the following places:

- [The Data Section](#)
- [The Stack](#)
- [The Heap](#)

The Data Section

Your program uses the data section for storing static and global variables. Memory in this section is permanently allocated, and the operating system sets its size when it loads your program. Variables in this section exist for the entire time that your program executes.

Errors can occur if your program tries to manage this section's memory. For example, you cannot free memory allocated to variables in the data section. In general, data section errors are usually related to not understanding that the program cannot manage data section memory.

The Stack

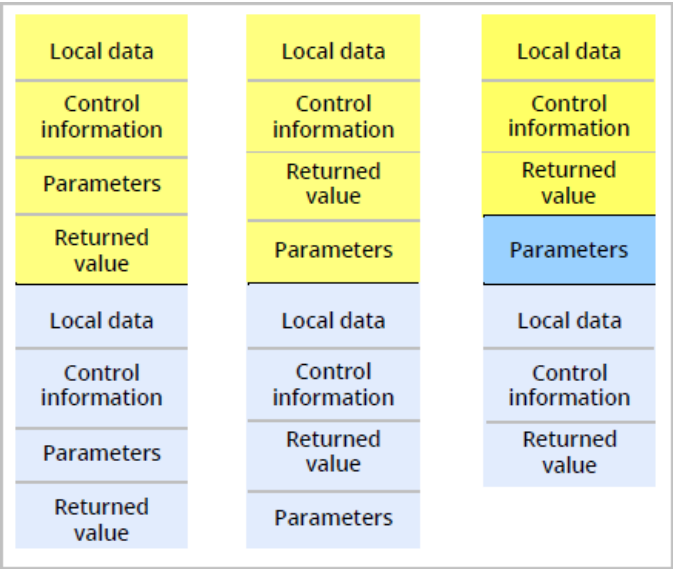
Memory in the stack section is dynamically managed by your program or operating system's memory manager. Consequently, your program cannot allocate memory in the stack or deallocate memory in it.

NOTE >>“Deallocates” means that your program tells a memory manager that it is no longer using this memory. The next time your program calls a routine, the new stack frame overwrites the memory previously used by other routines. In almost all cases, deallocated memory, whether on the stack or the heap, just hangs around in its pre-deallocation state until it gets reassigned.

The stack differs from the data section in that your program implicitly manages its space. What's in it one minute might not be there a minute later. Your program's runtime environment allocates memory for stack frames as your program calls routines and deallocates these frames when execution exits from the routine.

A stack frame contains control information, data storage, and space for passed-in arguments (parameters) and the returned value (and much more). [Figure 6](#) shows three ways in which a compiler can arrange stack frame information.

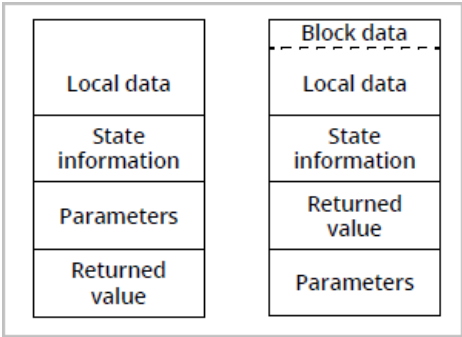
Figure 6: Placing Parameters



In this figure, the left and center stack frames have different positions for the parameters and returned value. The stack frame on the right is a little more complicated. In this version, the parameters reside in a stack memory area that doesn't belong to either stack frame.

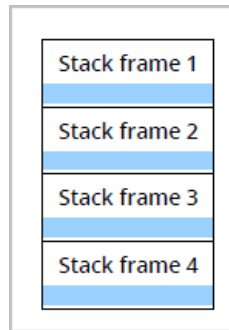
If a stack frame contains local (sometimes called *automatic*) variables, where is this memory placed? If the routine has blocks in which memory is allocated, where on the stack is this memory for these additional variables placed? Although there are many variations, Figure 7 shows two of the more common ways to allocate memory.

Figure 7: Local Data in a Stack Frame

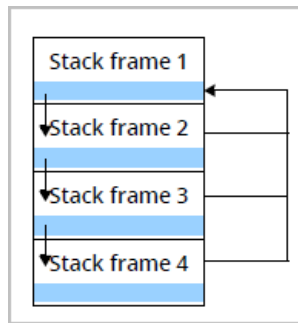


The blocks on the left show a data block allocated within a stack frame on a system that ignores your routine's block structure. The compiler figures out how much memory your routine needs, and then allocates enough memory for all of a routine's automatic variables. These kinds of systems minimize the time necessary to allocate memory. Other systems dynamically allocate the memory required within a routine as the block is entered, and then deallocate it as execution leaves the block. (The blocks on the right show this.) These systems minimize a routine's size.

As your program executes routines, routines call other routines, placing additional routines on the stack. Figure 8 shows four stack frames. The shaded areas represent local data.

Figure 8: Four Stack Frames

What happens when a program passes a pointer to memory in a stack frame to lower frames? [Figure 9](#) shows a program passing a pointer to memory in stack frame 1 down to lower stack frames.

Figure 9: Passing Pointers

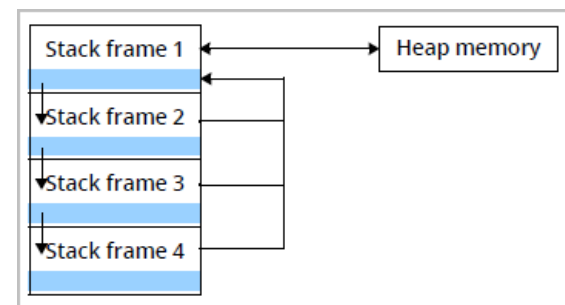
In this figure, the arrows on the left represent an address contained within a pointer, an address that is passed down the stack. The lines and arrow on the right indicate the place to which the pointer is pointing. A pointer to memory in frame 1 is passed to frame 2, which passes the pointer to frame

3, and then to frame 4. In all frames, the pointer points to a memory location in frame 1. Stated in another way, the pointers in frames 2, 3, and 4 point to memory in another stack frame. This is the most efficient way for your program to pass data from one routine to another since your program passes the pointer instead of the actual data. Using the pointer, the program can both access and alter the information that the pointer is pointing to.

NOTE >> Sometimes you read that data can be passed “by-value” (which means copying it) or “by-reference” (which means passing a pointer). This really isn’t true. Something is always copied. “Pass by reference” means that instead of copying the data, the program copies a pointer to the data.

Because the program’s runtime system owns stack memory, you cannot free it. Instead, your program’s runtime system frees it when it pops a frame from the stack.

One of the reasons for memory problems is that it may not be clear which component owns a variable’s memory. For example, [Figure 10](#) shows a routine in frame 1 that has allocated memory in the heap, and which passes a pointer to that memory to other stack frames.

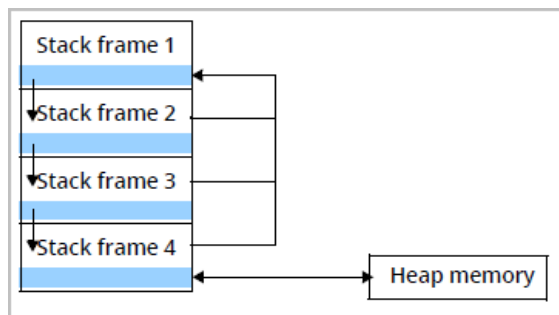
Figure 10: Allocating a Memory Block

If the routine executing in frame 4 frees this memory, all pointers to that memory are dangling; that is, they point to deallocated memory. If the program's memory manager reallocates this heap memory block, the data accessible by all the pointers is both invalid and wrong. Note that if the memory manager doesn't immediately reuse the block, the data accessed through the pointers is still correct.

The timing of the reallocation and reuse of a block by another allocation request means there is no guarantee that the data is correct when the program accesses the block, and there is never a pattern to when the block's data changes. Consequently, the problem occurs only intermittently, which makes it nearly impossible to locate. Worse, development systems usually are not as memory stressed as production systems, so the problem may occur only in the production environment.

Another common problem occurs when you allocate memory and assign its location to an automatic variable, shown in [Figure 11](#).

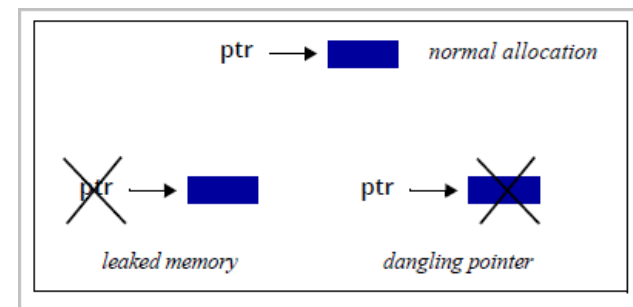
Figure 11: Allocating a Block from a Stack Frame



If frame 4 returns control to frame 3 without deallocating the heap memory it created, this memory is no longer accessible, and your program can no longer use this memory block. It has *leaked* this memory block.

NOTE >> If you have trouble remembering the difference between a leak and a dangling pointer, the following figure may help. In both cases, your program allocates heap memory, and the address of this memory block is assigned to a pointer. A leak occurs when the pointer gets deleted, leaving a block with no reference. In contrast, a dangling pointer occurs when the memory block is deallocated, leaving a pointer that points to deallocated memory. Both are shown in [Figure 12](#).

Figure 12: Leaks and Dangling Pointers



MemoryScape can tell you about all of your program's leaks. For information on detecting leaks, see ["Finding Memory Leaks"](#) on page 25.

The Heap

The *heap* is an area of memory that your program uses when it wants to dynamically allocate space for data. While using the heap gives you a considerable amount of flexibility, your program must manage this resource. That is, the program must explicitly allocate and deallocate this space. In contrast, the program does not allocate or deallocate memory in other areas.

Because allocations and deallocations are intimately linked with your program's algorithms and, in some cases, the way the program uses this memory is implicit rather than explicit, problems associated with the heap are the hardest to find.

Finding Heap Allocation Problems

Memory allocation problems are seldom due to allocation requests. Because an operating system's virtual memory space is large, allocation requests usually succeed. Problems most often occur if you are either using too much memory or leaking it. Although problems are rare, you should always check the value returned from calls to allocation functions such as **malloc()**, **calloc()**, and **realloc()**. Similarly, you should always check whether the C++ **new** operator returns a null pointer. (Newer C++ compilers throw a **bad_alloc** exception.) If your compiler supports the **new_handler** operator, you can throw your own exception.

Finding Heap Deallocation Problems

MemoryScape can let you know when your program encounters a problem in deallocating memory. Some of the problems it can identify are:

- **free() not allocated:** An application calls the **free()** function by using an address that is not in a block allocated in the heap.

- **realloc() not allocated:** An application calls the **realloc()** function by using an address that is not in a block allocated in the heap.
- **Address not at start of block:** A **free()** or **realloc()** function receives a heap address that is not at the start of a previously allocated block.

If a library routine uses the program's memory manager (that is, it is using the heap API) and a problem occurs, MemoryScape still locates the problem. For example, the **strdup()** string library function calls the **malloc()** function to create memory for a duplicated string. Since the **strdup()** function is calling the **malloc()** function, MemoryScape can track this memory.

MemoryScape can stop execution just before your program misuses a heap API operation. This lets you see what the problem is before it actually occurs. (For a reminder, see ["Behind the Scenes"](#) on page 8.)

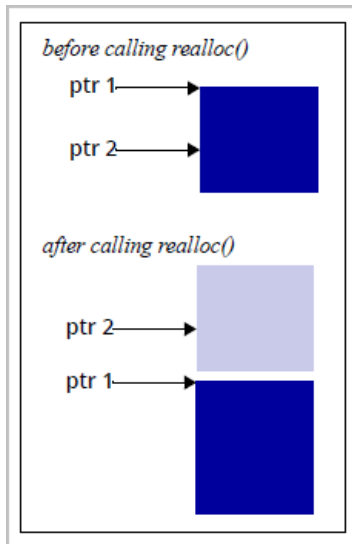
NOTE >>When Memory Scape is used with TotalView, execution stops before your program's heap manager deallocates memory, you can use the Thread > Set PC command to set the PC to a line after the **free()** request. This means that you can continue debugging past a problem that might cause your program to crash. In this case, the problem is solved while in the debugger. You still need to fix the problem.

realloc() Problems

The **realloc()** function can either extend a current memory block, or create a new block and free the old. When it creates a new block, it can create problems. Although you can check to see which action occurred, you need to code **realloc()** usage defensively so that problems do not occur. Specifically, you must change every pointer pointing to the memory block that was reallocated so that it points to the new one. Also, if the pointer doesn't point to the beginning of the block, you need to take some corrective action.

In [Figure 13](#), two pointers are pointing to a block. After the `realloc()` function executes, **ptr1** points to the new block. However, **ptr2** still points to the original block, a block that a program deallocated and returned to the heap manager.

Figure 13: realloc() Problem



If you use block painting, MemoryScape can initialize the first block with a bit pattern. If your program is able to display the contents of this block to you, you'll be able to see what kind of problem occurred.

Finding Memory Leaks

A memory "leak" is a block of memory that a program allocates that is no longer referenced. (Technically, there's no such thing as a memory leak. Memory doesn't leak and can't leak.) For example, when your program allocates memory, it assigns the block's location to a pointer. A leak can occur if one of the following occurs:

- You assign a different value to that pointer.
- The pointer was a local variable and execution exited from the block.

If your program leaks a lot of memory, it can run out of memory. Even if it doesn't run out of memory, your program's memory footprint becomes larger. This increases the amount of paging that occurs as your program executes. Increased paging makes your program run slower.

Here are some of the circumstances in which memory leaks occur:

- **Orphaned ownership**—Your program creates memory but does not preserve the address so that it can deallocate it at a later time.

The following example makes this (extremely) obvious:

```
char *str;
for( i = 1; i <= 10; i++ )
{
    str = (char *)malloc(10*i);
}
free( str );
```

In the loop, your program allocates a block of memory and assigns its address to **str**. However, each loop iteration overwrites the address of the previously created block. Because the address of the previously allocated block is lost, its memory can never be made available to your program.

- **Concealed allocation**—Creating a memory block is separate from using it.

Because all programs rely on libraries in some fashion, you must understand what responsibilities you have for allocating and managing memory. As an example, contrast the **strcpy()** and **strdup()** functions. Both do the same thing—they copy a string. However, the **strdup()** function uses the **malloc()** function to create the memory it needs, while the **strcpy()** function uses a buffer that your program creates.

In many cases, your program receives a handle from a library. This handle identifies a memory block that a library allocated. When you pass the handle back to the library, it knows which memory block contains the data you want to use or manipulate. There may be a considerable amount of memory associated with the handle, and deleting the handle without telling the library to deallocate the memory associated with the handle leaks memory.

- **Changes in custody**—The routine creating a memory block is not the routine that frees it. (This is related to concealed allocation.)

For example, routine 2 asks routine 1 to create a memory block. At a later time, routine 2 passes a reference to this memory to routine 3. Which of these blocks is responsible for freeing the block?

This type of problem is more difficult than other types of problems in that it is not clear when your program no longer needs the data. The only thing that seems to work consistently is reference counting. In other words, when routine 2 gets a memory block, it increments a counter. When it passes a pointer to routine 3, routine 3 also increments the counter. When routine 2 stops executing, it decrements the counter. If it is zero, the executing routine frees the memory. If it isn't zero, another routine frees it at another time.

- **Underwritten destructors**—When a C++ object creates memory, it must have a destructor that frees it. No exceptions. This doesn't mean that a block of memory cannot be allocated and used as a general buffer. It just means that when an object is destroyed, it needs to completely clean up after itself; that is, the program's destructor must completely clean up its allocated memory.

For more information, see [“Finding free\(\) and realloc\(\) Problems”](#) on page 21.

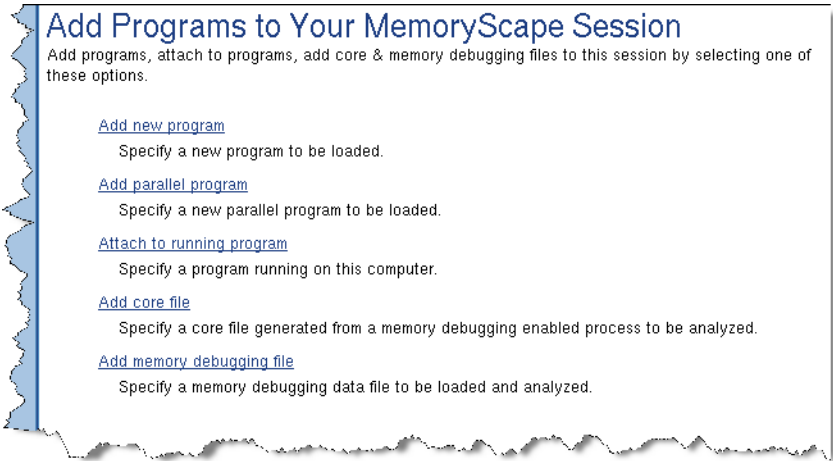
Starting MemoryScape

On most architectures, there is not much you need to do to prepare your program for memory debugging. In most cases, just compile your program using the **-g** command-line option. In some cases, you may need to link your program with the MemoryScape agent. (See [Creating Programs for Memory Debugging](#),” on page 121 for more information.)

Here is how you start MemoryScape:

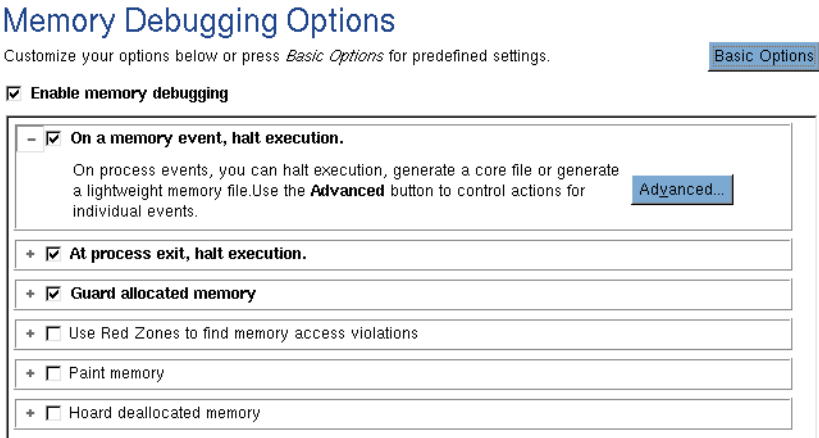
- 1. Start MemoryScape from a shell window. On a Macintosh, depending upon how it was installed, you can start it using the MemoryScape program icon.
- 2. If you didn't specify a program name on the command line, select **Add new program**, [Figure 14](#).

Figure 14: Add Programs to Your MemoryScape Session



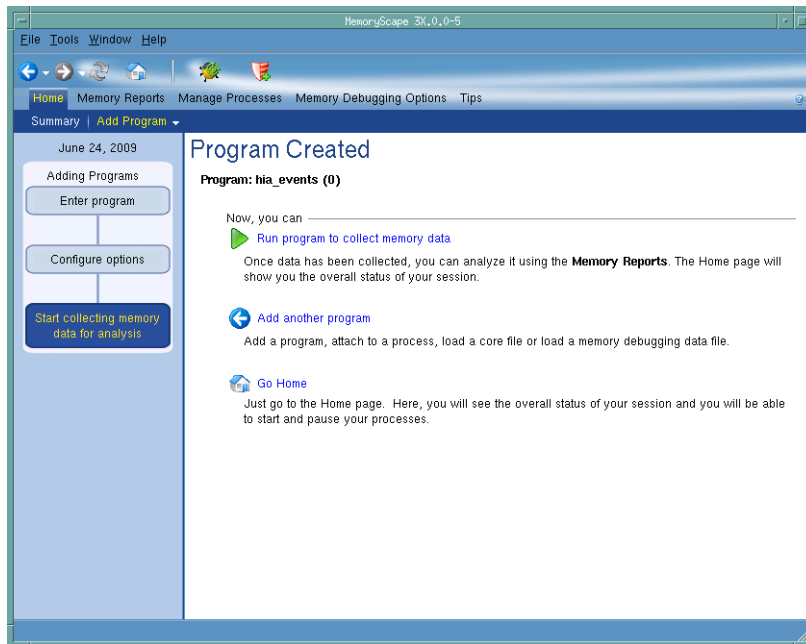
- 3. If you need to, select your memory debugging options. In most cases, the defaults are fine. In some cases, you may want to select **Low**, **Medium**, **High**, or **Extreme**. If you need finer control, select the **Advanced Options** button, [Figure 15](#). For more information, see [Task 3: “Setting MemoryScape Options”](#) on page 71.

Figure 15: Memory Debugging Options



- 4. After setting options, MemoryScape displays a screen that lets you start program execution, [Figure 16](#).

Figure 16: Program Created Screen



Whenever your program is stopped—which happens when you halt the program, when a memory problem occurs, or just before the program exits—MemoryScape can create a report that describes leaks or one that shows or describes currently allocated memory blocks.

Using MemoryScape Options

Before your program begins execution, you can set other options by selecting controls in the **Memory Debugging Options** screen shown in Figure 15. (Some of these options can be set at other times.) Here's a summary of these options:

- **Halt execution at process exit**—Stops execution before your program exits. This lets you analyze the memory information that MemoryScape has collected. If the program does finish executing, MemoryScape discards this memory information, and you can no longer view its reports.
- **Halt execution on memory event or error**— Stops execution and notifies you if a heap event such as a deallocation or a problem occurs. This is called *event notification*. (See “[Event and Error Notification](#)” on page 21 for more information.) By default, this is set to **On**.
- **Guard allocated memory**—Surrounds allocations with a small amount of additional memory. By default, it uses 8 bytes of memory. It also writes a pattern into this memory. These additional memory blocks are called *guard blocks*. If your program overwrites these blocks, you can tell that a problem occurred either by asking for a report or by an event notification when your program deallocates a guarded block.
- **Use Red Zones to find memory access violations**—Adds a Red Zone to your allocated blocks. The Red Zone is an additional page of memory located either before or after your block. If MemoryScape detects access in the Red Zone that is outside the bounds of your allocated block, it halts execution of your program and notifies you of the underrun or overrun. For more information see “[Using Red Zones](#)” on page 53.
- **Paint memory**—Paints allocated and deallocated memory and the pattern that MemoryScape uses when it paints this memory. For more information, see “[Finding free\(\) and realloc\(\) Problems](#)” on page 21 and [Task 3: “Setting MemoryScape Options”](#) on page 71.

- **Hoard deallocated memory**—Retains deallocated memory blocks, how much memory to use for these blocks, and the number of blocks to retain. For more information, see [Task 3: “Setting MemoryScape Options”](#) on page 71.

Preloading MemoryScape

MemoryScape must be able to preload your program with its agent. In many cases, it does this automatically. However, MemoryScape cannot preload the agent for applications that run on IBM RS/6000 platforms. For more information, see [“Creating Programs for Memory Debugging”](#) on page 121.

Understanding How Your Program is Using Memory

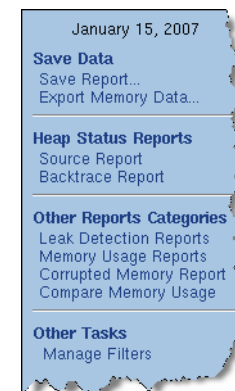
MemoryScape can help you understand how your program is using memory in these ways:

- Its **Memory Usage** reports show how much memory your program's processes are using. (See [Task 5: "Seeing Memory Usage"](#) on page 86.)
- Its compare feature displays the differences between your program's current memory state and saved memory data or the differences between two sets of saved memory data. (See [Task 13: "Comparing Memory"](#) on page 113.)
- Its heap status reports tell you how much memory your program allocated and where it was allocated. (See [Task 8: "Obtaining Detailed Heap Information"](#) on page 97.)

You can examine imported memory state information in the same way as the memory for a live process. For example, you can look for leaks, graphically display the heap, and so on.

Use the **Export Memory Data** command on the left side of many screens when you want to save memory data, [Figure 17](#).

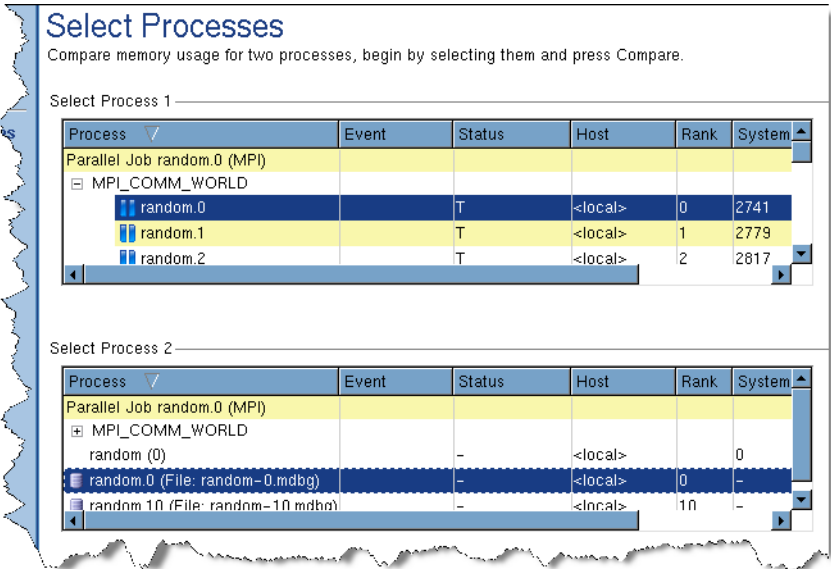
Figure 17: Memory Reports Commands



At a later time, you can import this data back into MemoryScape using the **Add memory debugging file** command, which is on the **Home | Add Program** screen.

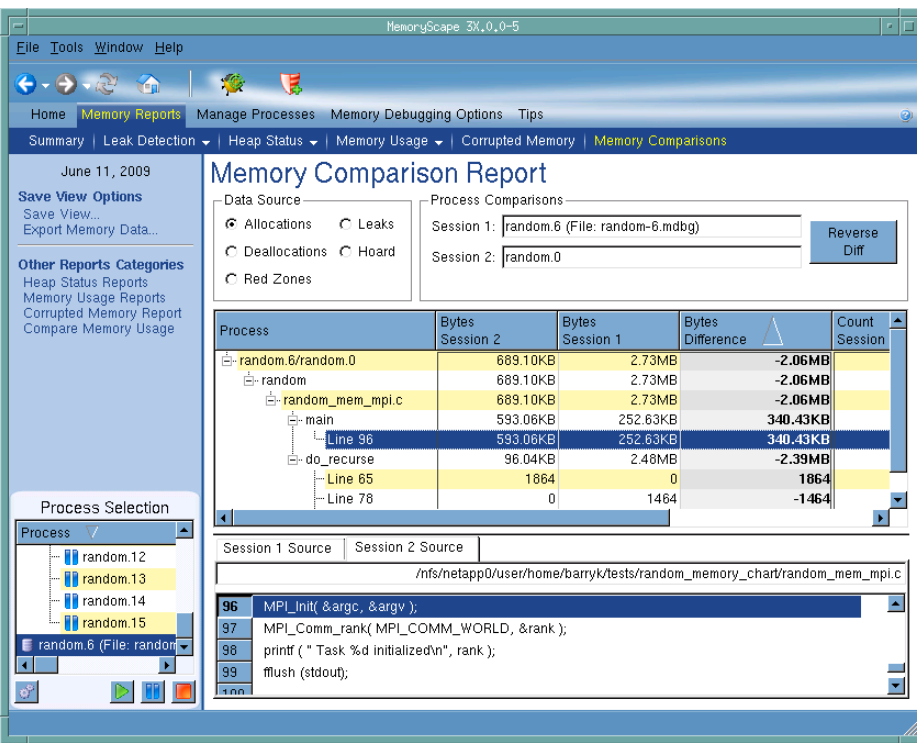
MemoryScape treats this imported information in nearly the same way as the information it has accumulated for a live process. After selecting **Memory Reports | Memory Comparisons**, MemoryScape displays a screen containing all processes and imported files, [Figure 18](#).

Figure 18: Select Processes and Files



After you select two processes, MemoryScape can display the differences between this information, [Figure 19](#).

Figure 19: Memory Comparison Report



Finding free() and realloc() Problems

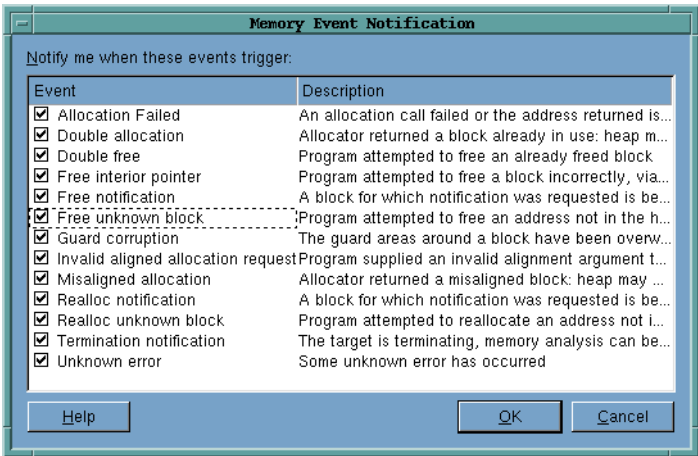
MemoryScape detects problems that occur when you allocate, reallocate, and free heap memory.

This memory is usually allocated by the **malloc()**, **calloc()**, and **realloc()** functions, and deallocated by the **free()** and **realloc()** functions. In C++, MemoryScape tracks the **new** and **delete** operators. If your Fortran programs and libraries use the heap API, MemoryScape tracks their dynamic memory use. Some Fortran systems use the heap API for assumed-shape, automatic, and allocatable arrays. See your system's **man** pages and other documentation for more information.

Event and Error Notification

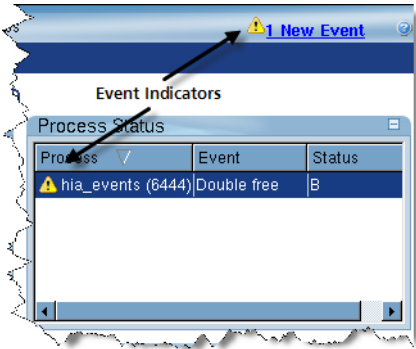
There are a number of events that can cause MemoryScape to stop your program's execution, and you can control which of these will actually halt execution by selecting the **Advanced** button in the Halt execution area of the **Memory Debugging Options** screen, [Figure 20](#).

Figure 20: Memory Error Notification Dialog Box



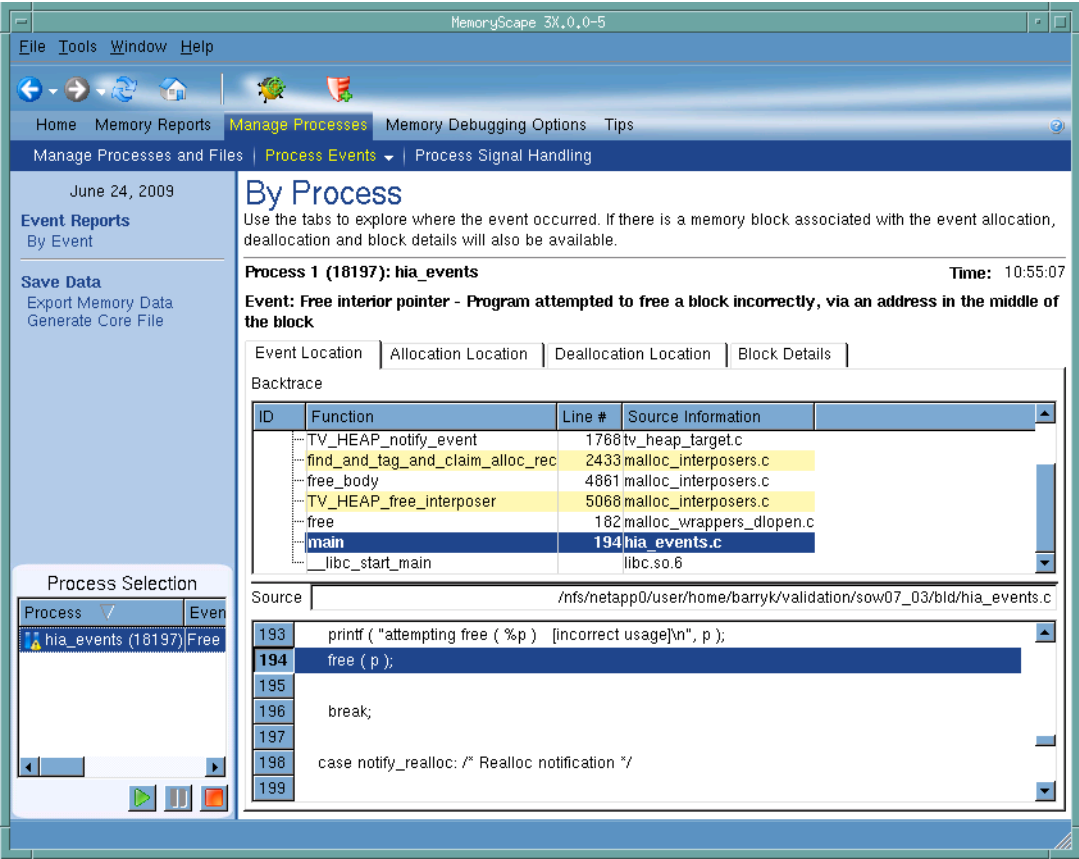
When one of these errors occurs, MemoryScape places event indicators by the process and at the top of the window, [Figure 21](#).

Figure 21: Event Indicators



When you click on one of these indicators, MemoryScape displays the **Manage Processes | Process Events** screen, [Figure 22](#).

Figure 22: Process Event Screen



This screen contains the following kinds of information:

- The **bold** information reports the type of error or event that occurred.
- The **Event Location** tab contains the function backtrace if the error or event is related to a block allocated on the heap. This is the backtrace that exists when the event occurred. Depending upon the problem, you may also want to examine the information in the next two tabs.

- The **Block Details** tab displays this information in a manner similar to that shown in a graphical heap display.
- MemoryScope retains information about the backtrace that existed when your program allocated and deallocated the memory block. Select which to display using either the **Allocation Location** or **Deallocation Location** tab.

If a memory error occurred, the deallocation backtrace is often the same as the backtrace shown in the **Event Location** tab. If the memory error occurs after your program deallocated this memory, the backtraces are different.

Types of Problems

This section presents some trivial programs that illustrate some of the **free()** and **realloc()** problems that MemoryScope detects. The errors shown in these programs are obvious. Errors in your program are, of course, more subtle.

Freeing Stack Memory

The following program allocates stack memory for the **stack_addr** variable. Because the memory was allocated on the stack, the program cannot deallocate it.

```
int main (int argc, char *argv[])
{
    void *stack_addr = &stack_addr;
    /* Error: freeing a stack address */
    free(stack_addr);
    return 0;
}
```

Freeing bss Data

The **bss** section contains uninitialized data. That is, variables in this section have a name and a size but they do not have a value. Specifically, these variables are your program's uninitialized static and global variables. Because they exist in a data section, your program cannot free their memory.

The following program tries to free a variable in this section:

```
static int bss_var;

int main (int argc, char *argv[])
{
    void *addr = (void *) (&bss_var);
    /* Error: address in bss section */
    free(addr);
    return 0;
}
```

Freeing Data Section Memory

If your program initializes static and global variables, it places them in your executable's data section. Your program cannot free this memory.

The following program tries to free a variable in this section:

```
static int data_var = 9;

int main (int argc, char *argv[])
{
    void *addr = (void *) (&data_var);
    /* Error: address in data section */
    free(addr);
    return 0;
}
```

Freeing Memory That Is Already Freed

The following program allocates some memory, and then releases it twice. On some operating systems, your program will raise a SEGV on the second free request.

```
int main ()
```

```
{
void *s;
/* Get some memory */
s = malloc(sizeof(int)*200);
/* Now release the memory */
free(s);
/* Error: Release it again */
free(s);
return 0;
}
```

```
int main (int argc, char *argv[])
{
char *s, *misaligned_s;
/* Get some memory */
s = malloc(sizeof(int)*64);
/* Release memory using a misaligned address */
misaligned_s = s + 8;
free(misaligned_s);
free(s);
return 0;
}
```

Tracking realloc() Problems

The following program passes a misaligned address to the **realloc()** function.

```
int main (int argc, char *argv[])
{
char *s, *bad_s, *realloc_s;
/* Get some memory */
s = malloc(sizeof(int)*64);
/* Reallocate memory using a misaligned address */
bad_s = s + 8;
realloc_s = realloc(bad_s, sizeof(int)*256);
return 0;
}
```

In a similar fashion, MemoryScape detects **realloc()** problems caused by passing addresses to memory sections whose memory your program cannot free. For example, MemoryScape detects problems if you try to do any of the following:

- Reallocate stack memory.
- Reallocate memory in the data section.
- Reallocate memory in the **bss** section.

Freeing the Wrong Address

MemoryScape can detect when a program tries to free a block that does not correspond to the start of a block allocated using the **malloc()** function. The following program illustrates this problem:

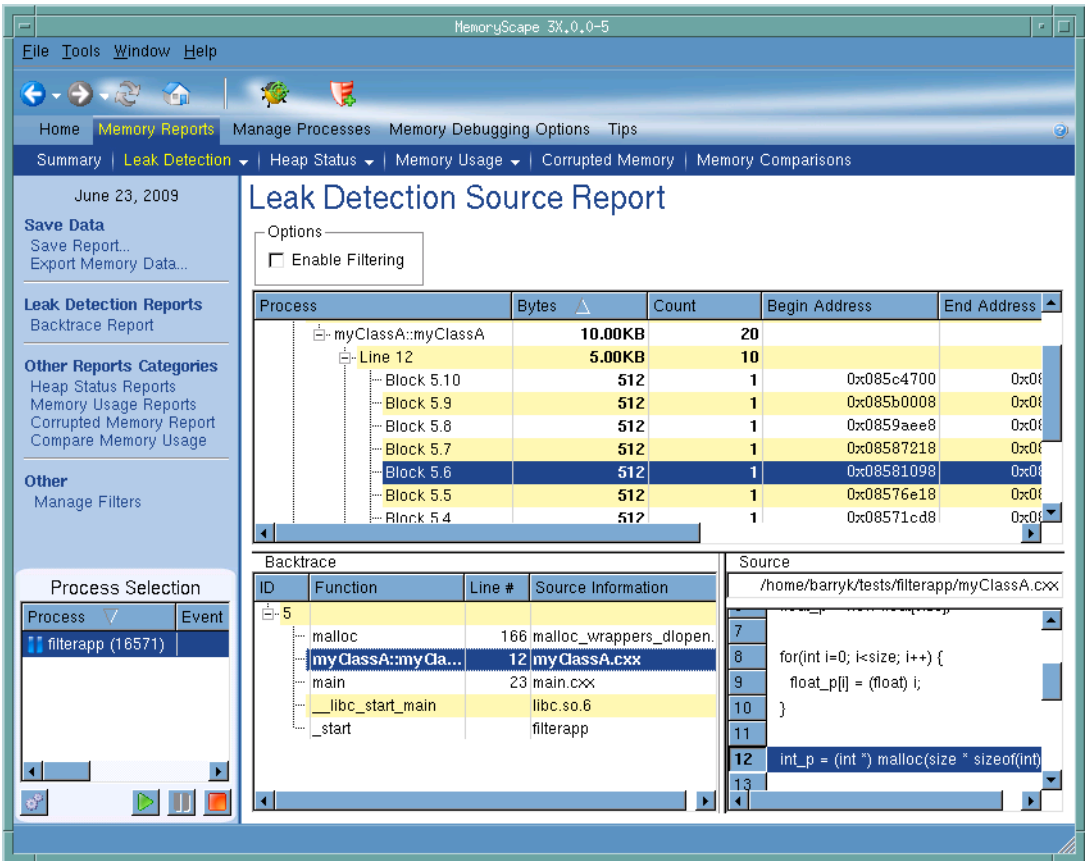
Finding Memory Leaks

MemoryScape can locate your program's memory leaks and display information about them. Here's what you can do:

1. Run the program and then halt it when you want to look at memory problems. You should allow your program to run for a while before stopping execution to give it enough time to create leaks.
2. Select **Memory Reports | Leak Detection**, [Figure 23](#).
3. Select a report. For example, you might select **Source Report**.

The top portion shows all of your program's files. The second column in this list shows the number of bytes that are leaked from code in those files. You may want to click on the **Bytes** header so that MemoryScape displays which files have the greatest number of leaks.

Figure 23: Leak Detection Source Report



4. After you select a leak in the top part of the window, the bottom of the window shows a backtrace of the place where your program allocated the memory. After you select a stack frame in the **Backtrace** pane, MemoryScape displays the statement where the block was created.

The backtrace that MemoryScape displays is the backtrace that existed when your program made the heap allocation request. It is not the current backtrace.

Many users like to generate a report that contains all leaks for the entire program. If you are running with TotalView, you can set a breakpoint on your program's Exit statement. Otherwise, MemoryScape will automatically stop your program during your program's exit. After your program stops executing, generate a **Leak Detection** Report. You may want to write this report to disk.

MemoryScape uses a conservative approach to finding memory leaks, searching roots from the stack, registers, and data sections of the process for references into the heap. Although leaks will not be falsely reported, some leaks may be missed. If you are within a method that has leaks, you may need to step out of the method for the leak to be reported. Leak detection may be sensitive to the compiler used to build the program.

Fixing Dangling Pointer Problems

NOTE >> You must be using MemoryScape with TotalView to be able to find dangling pointers.

Fixing dangling pointer problems is usually more difficult than fixing other memory problems. First of all, you become aware of them only when you realize that the information your program is manipulating isn't what it is supposed to be. Even more troubling, these problems can be intermittent, happening only when your program's heap manager reuses a memory block. For example, if nothing else is running on your computer, your program may never reuse the block. If there are a large number of jobs running, it could reuse a deallocated block quickly.

After you identify that you have a dangling pointer problem, you have two problems to solve. The first is to determine where your program freed the memory block. The second is to determine where it *should* free this memory. MemoryScape tools that can help you are:

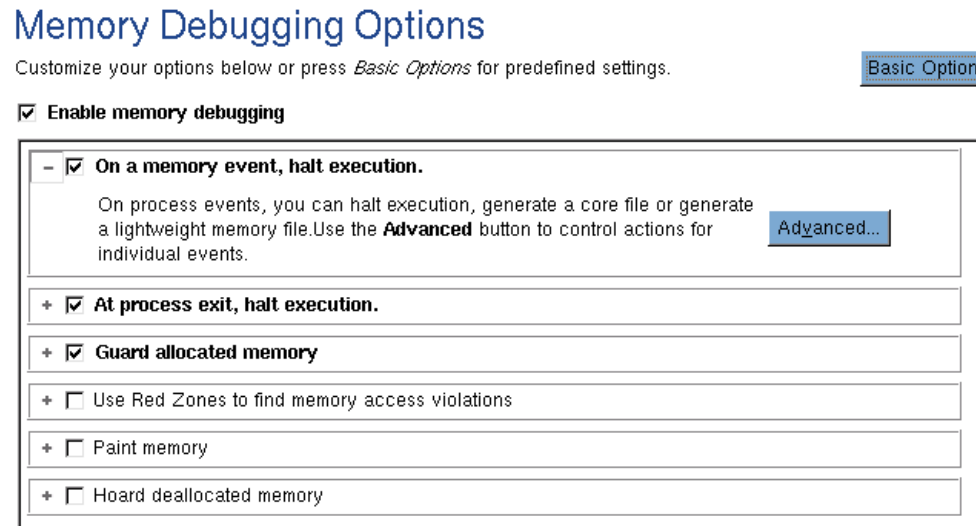
- **Block painting**, in which MemoryScape writes a bit pattern into allocated and deallocated memory blocks.
- **Hoarding**, in which MemoryScape holds onto a memory block when the heap manager receives a request to free it. This is most often used to get beyond where a problem occurs. By allowing the program to continue executing with correct data, you sometimes have a better chance to find the problem. For example, if you also paint the block, it becomes easy to

tell what the problem is. In addition, your program might crash. (Crashing while you are in TotalView is a good thing, because it will show the crash point, and you will immediately know where the problem is.)

- **Watchpoints**, in which TotalView stops execution when a new value is written into a memory block. If MemoryScape is painting deallocated blocks, you immediately know where your program freed the block.
- **Block tagging**, in which TotalView stops execution when your program deallocates or reallocates memory.

Enable painting and hoarding in the Memory Debugging Options Page, [Figure 24](#).

Figure 24: Memory Debugging Options



You can turn painting and hoarding on and off. In addition, you can tell MemoryScape what bit patterns to use when it paints memory. For more information, see “[Block Painting](#)” on page 54.

Dangling Pointers

If you enable memory debugging, TotalView displays information in the Variable Window about the variable’s memory status; that is, whether the memory is allocated or deallocated. The following small program allocates a memory block, sets a pointer to the middle of the block, and then deallocates the block:

```
main(int argc, char **argv)
{
    int *addr = 0; /* Pointer to start of block. */
    int *misaddr = 0; /* Pointer to interior of block. */
```

```
    addr = (int *) malloc (10 * sizeof(int));
    misaddr = addr + 5; /* Point to block interior */

    /* Deallocate the block. addr and */
    /* misaddr are now dangling. */
    free (addr);
}
```

Figure 25 shows two Variable Windows. Execution was stopped before your program executed the **free()** function. Both windows contain a memory indicator saying that blocks are allocated.

Figure 25: Allocated Descriptions in a Variable Window

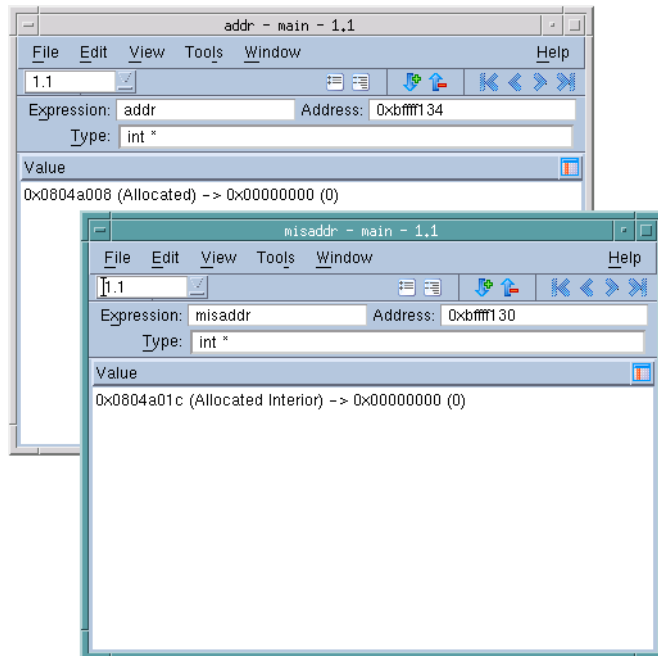
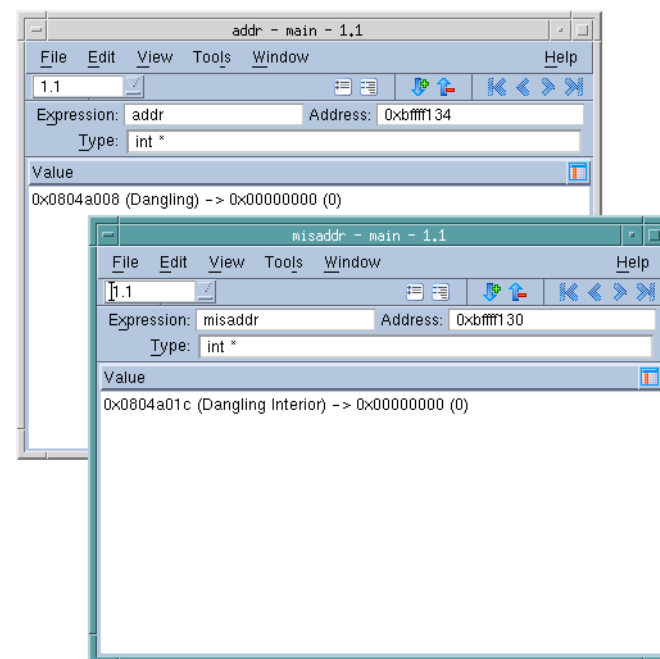


Figure 26: Dangling Description in a Variable Window



After your program executes the **free()** function, the messages change, [Figure 26](#).

Batch Scripting and Using the CLI

Batch Scripting Using tvscript

TotalView and MemoryScape can run unattended if you use the **tvscript** shell command. This is called batch debugging because like all batch programs, you do not need to use them interactively. In addition, you can invoke **tvscript** using **cron** to schedule debugging at off-hours, so reports will be ready when you want them.

The commands that **tvscript** executes can be entered in two ways. The first is to just use command-line options. The second—and recommended—is to create a file containing commands that **tvscript** executes. A file can contain Tcl callback functions to be called when an event occurs within your program. These callback functions can also include CLI commands.

Here is an example of how **tvscript** is invoked using command-line options on an MPI program:

```
tvscript -mpi "Open MPI" -tasks 4 \
  -create_actionpoint "hello.c#14=>show_backtrace" \
  ~/Tests/hello-mpi/hello
```

Some of the events that **tvscript** can act on are memory events. For example, if a **double_free** event occurs, you can create a Tcl callback function that tells you more about what the event. This callback function can include CLI commands.

"Batch Debugging Using tvscript" in the *Classic TotalView Reference Guide* explains how to use the **tvscript** command.

Using the dheap Command

NOTE >> You must be using MemoryScape with TotalView to have access to the CLI.

NOTE >> For a complete description of the **dheap** command, see the **dheap command description** in the *Reference Guide*.

The **dheap** command tracks memory problems from within the CLI. It supports the same functionality as the GUI, with some additional options. Here are actions available through the **dheap** command:

- To see the status of MemoryScape, use the **dheap** command with no arguments, or with the **-status** argument.
- To enable and disable MemoryScape, use **dheap -enable** and **dheap -disable**.
- To display information about the heap, use **dheap -info**. You can show information for the entire heap or limit what TotalView displays to just a part of it.
- To limit backtrace information to only the information that is important to you, use the **dheap -backtrace** command.
- To report errors when data is written outside a memory allocation, use the **dheap -guard** command.
- To determine whether there are pointers that point to or within a deallocated memory block, use the **dheap -is_dangling** command.
- To be notified of memory allocation or reallocation, use the **dheap -tag_alloc** command.

- To start and stop error notification, use **dheap -notify** and **dheap -nonotify**.
- To filter the information displayed, use **dheap -filter**.
- To check for leaks, use **dheap -leaks**.
- To paint memory with a bit pattern, use **dheap -paint**.
- To hoard memory, use **dheap -hoard**.
- To export view information, use **dheap -export**.
- To detect bounds and use-after-free errors, use **dheap -red_zones**.
- To compare memory states, either against a baseline or against a saved state or between two saved states, use **dheap -compare**.

NOTE >>Some **dheap** options are not available in the GUI.

dheap Example

The following example shows typical CLI information returned after Memory-Scape locates an error:

```
d1.<> dheap
      process:   Enable   Notify   Available
1      (18993):      yes      yes      yes
1.1    realloc: Address does not match any allocated
block.: 0xbfffd87c

d1.<> dheap -info -backtrace
process 1      (18993):
0x8049e88 --      0x8049e98      0x10 [      16]
flags: 0x0 (none)
:   realloc   PC=0x400217e5 [../malloc_wrappers_dlopen.c]
:   argz_append PC=0x401ae025 [/lib/i686/libc.so.6]
:   __newlocale PC=0x4014b3c7 [/lib/i686/libc.so.6]
:
```

```
...
../malloc_wrappers_dlopen.c]
:   main      PC=0x080487c4 [../realloc_prob.c]
:   __libc_start_main PC=0x40140647 [/lib/i686/libc.so.6]
:   _start     PC=0x08048621 [../realloc_prob]

0x8049f18 --      0x8049f3a      0x22 [      34]
flags: 0x0 (none)
:   realloc   PC=0x400217e5 [../malloc_wrappers_dlopen.c]
:   main      PC=0x0804883e [../realloc_prob.c]
:   __libc_start_main PC=0x40140647 [/lib/i686/libc.so.6]
:   _start     PC=0x08048621 [../realloc_prob]
```

The information displayed in this example is explained in more detail in the following sections.

Notification When free Problems Occur

If you type **dheap -enable -notify** and then run your program, Memory-Scape notifies you if a problem occurs when your program tries to free memory.

When execution stops, you can type **dheap** (with no arguments), to display information about what happened. You can also use the **dheap -info** and **dheap -info -backtrace** commands to display additional information. The information displayed by these commands lets you locate the statement in your program that caused the problem. For example:

```
d1.<> dheap
      process:   Enable   Notify   Available
1      (18993):      yes      yes      yes
1.1    realloc: Address does not match any allocated block.:
0xbfffd87c
```

For each allocated region, the CLI displays the start and end address, and the length of the region in decimal and hexadecimal formats. For example:

```
d1.<> dheap
      process:   Enable   Notify   Available
```

```
1      (30420):      yes      yes      yes
1.1  free: Address is not the start of any allocated block.:
      free: existing allocated block:
      free: start=0x08049b00 length=(17 [0x11])
      free: flags: 0x0 (none)
      free: malloc      PC=0x40021739 [./.../
malloc_wrappers_dlopen.c]
      free: main      PC=0x0804871b [../free_prob.c]
      free: __libc_start_main PC=0x40140647 [/lib/i686/
libc.so.6]
      free: _start      PC=0x080485e1 [./.../free_prob]

      free: address passed to heap manager: 0x08049b08
```

MemoryScape can also tell you when your program deallocates or reallocates tagged blocks. For more information, see “Deallocation Notification: `dheap -tag_alloc`” on page 42.

Showing Backtrace Information: `dheap -backtrace`:

The backtrace associated with a memory allocation can contain many stack frames that are part of the heap library, MemoryScape’s library, and other related functions and libraries. You are not usually interested in this information, since these stack frames aren’t part of your program. Using the `-backtrace` option lets you manage this information, as follows:

- **`dheap -backtrace -set_trim value`**
Removes—that is, trims—this number of stack frames from the top of the backtrace. This lets you *hide* the stack frames that you’re not interested in as they come from libraries.
- **`dheap -backtrace -set_depth value`**
Limits the number of stack frames to the value that you type as an argument. The *depth* value starts after the *trim* value. That is, the number of excluded frames does not include the frames that were trimmed.

Guarding Memory Blocks: `dheap -guards`

When your program allocates a memory block, MemoryScape can surround this block with additional memory. It will also initialize this memory to a bit pattern. When MemoryScape checks these blocks, it can tell if your program overwrote the blocks.

Checks can be made in the following ways:

- Use the **`dheap -guard -check`** command while the process is stopped. MemoryScape will respond by writing information about all overwritten guard blocks.
- Use the **`dheap -notify`** command. If you’ve turned on notification, MemoryScape checks guard blocks when your program deallocates a memory block. If that memory block’s guards were altered, the CLI stops program execution and MemoryScape writes information.

NOTE >>You should set the TotalView VERBOSE setting to WARNING. Setting it lower than this suppresses this output. Setting it higher tends to bury the information in debugger runtime information.

Use the `dheap -guard -set` command to turn this feature on or off. To see guard block status, use the **`dheap -guard`** command without an argument. For example:

```
d1.<> dheap -guard
      process: Enabled Max Size Pre Post Pre Post
1 (13071):      yes  0    8    8 0x77777777 0x99999999
```

If you are using the **`dheap -info`** command, you can include guard block information in the output by typing `dheap -info -show_guard_settings`.

Memory Reuse: dheap -hoard

In some cases, you may not want your system's heap manager to immediately reuse memory. You would do this, for example, when you are trying to find problems that occur when more than one process or thread is allocating the same memory block. Hoarding allows you to temporarily delay the block's release to the heap manager. When the hoard has reached its capacity in either size or number of blocks, MemoryScape releases previously hoarded blocks back to your program's heap manager.

The order in which MemoryScape releases blocks is the order in which it hoards them. That is, the first blocks hoarded are the first blocks released—this is a first-in, first-out (FIFO) queue.

Hoarding is a two-step process, as follows:

1. Use the **dheap -enable** command to tell MemoryScape to track heap allocations.
2. Use the **dheap -hoard -set on** command to tell MemoryScape not to release deallocated blocks back to the heap manager. (The **dheap -hoard -set off** command tells MemoryScape to no longer hoard memory.) After you turn hoarding on, use the **dheap -hoard -set_all_deallocs on** command to tell MemoryScape to start hoarding blocks.

At any time, you can obtain the hoard's status by typing the **dheap -hoard** command. For example:

```
d1.<> dheap -hoard
```

process:	Enabled	All deallocs	Max size	Max blocks	Size	Blocks
1 (10883):	yes	yes	16 (kb)	32	15 (kb)	9

The **Enabled** column contains either **yes** or **no**, which indicates whether hoarding is enabled. The **All deallocs** column indicates if hoarding is occurring. The next columns show the maximum size in kilobytes and number of blocks to which the hoard can grow. The last two columns show the current size of the hoard, again, in kilobytes and the number of blocks.

As your program executes, MemoryScape adds the deallocated region to a FIFO buffer. Depending on your program's use of the heap, the hoard could become quite large. You can control the hoard's size by setting the maximum amount of memory in kilobytes that MemoryScape can hoard and the maximum number of hoarded blocks.

dheap -hoard -set_max_kb num_kb

Sets the maximum size in kilobytes to which the hoard is allowed to grow. The default value on many operating systems is 32KB.

dheap -hoard -set_max_blocks num_blocks

Sets the maximum number of blocks that the hoard can contain.

You can tell which blocks are in the hoard by typing the **dheap -hoard -display** command. For example:

```
d1.<> dheap -hoard -display
```

process	1 (10883):			
	0x804cdb0 --	0x804d3b0	0x600 [1536]
flags:	0x32 (hoarded)			
	0x804d3b8 --	0x804dab8	0x700 [1792]
flags:	0x32 (hoarded)			
	0x804dac0 --	0x804e2c0	0x800 [2048]
flags:	0x32 (hoarded)			
	0x804fce8 --	0x804fee8	0x200 [512]
flags:	0x32 (hoarded)			
	0x804fef0 --	0x80502f0	0x400 [1024]
flags:	0x32 (hoarded)			

You can enable autoshrinking when hoarding by entering **dheap -hoard -autoshrink -set on**. This allows the hoard to contract automatically when memory is short. When an allocation request fails because of a shortage of

memory and autoshrinking is enabled, the HIA will eject a block from the hoard and automatically retry the request. This will continue until either the allocation succeeds, or the hoard is completely exhausted. In the latter case, the normal '**allocation operation returned null**' event is raised.

One other feature is associated with autoshrinking: a notification threshold size, given in kb. If, during the course of autoshrinking, the size of the hoard in kb crosses from above to below the threshold size defined by **-autoshrink -set_threshold_kb num_kb**, a new event is raised. This is to alert the user that space is running out. The event will be particularly useful when the size of the hoard is unlimited (which means that blocks released by the application are not returned to the heap manager), and therefore the size of the hoard really does reflect how much space is left.

One concern regarding the threshold event is that it could be reported many times if the size of the hoard fluctuates, crossing and recrossing the threshold. To reduce the noise, a count is associated with the threshold. The count is decremented each time the event is raised. This continues until the count reaches zero, after which the count is no longer decremented, and the event is not raised. Notification can be reactivated by re-priming the counter using the **-set_threshold_trigger** option. The default value for the number of times the event is triggered is 1. The trigger is independent of any event filtering that may be active.

The **-autoshrink -reset**, **-autoshrink -reset_threshold_kb**, and **-autoshrink -reset_threshold_trigger** options unset the TotalView settings for these controls. The HIA will determine its settings using the values in either the **TVHEAP_ARGS** environment variable, the HIA configuration file, or its default values.

Writing Heap Information: **dheap -export**

You may want to write the information that MemoryScape collects about your program to disk so that you can examine it at a later time. Or, you may want to save information from different sessions so that you can compare changes that you've made.

You can save MemoryScape information by using the **dheap -export** command. This command has two sets of options: one contain options you must specify, the other contains options that are optional. In all cases, you must use the:

- **-output** option to name the file to which MemoryScape writes information.
- **-data** option to name which data MemoryScape includes.

For example:

dheap -export -output heap.txt -data leaks

You can also add **-set_show_code** and **-set_show_backtraces**. These options are most often used to restrict the amount of information that MemoryScape displays. You can also use the **-check_interior** option to tell MemoryScape that if a pointer is pointing into a block instead of at the block's beginning, then the block shouldn't be considered as being leaked.

Filtering Heap Information: **dheap -filter**

Depending upon the way in which your program manages memory, MemoryScape might be managing a lot of information. You can filter this information down to focus on things that are important to you at the

moment by using filters. These filters can only be created using the GUI. However, after you create a filter using the GUI, you can apply it from within the CLI by using the **dheap -filter** commands.

Here is an excerpt from a CLI interaction:

```
d1.<> dheap -filter -list
Filtering of heap reports is 'disabled'
Individual filters are set as follows:
    Disabled  MyFilter  Function contains strdup

d1.<> dheap -filter -enable MyFilter
d1.<> dheap -filter -enable
d1.<> dheap -filter -list
Filtering of heap reports is 'enabled'
Individual filters are set as follows:
    Enabled   MyFilter  Function contains strdup

d1.<>
```

Notice that TotalView automatically knew about your filters. That is, it always reads your filter file. However, TotalView ignores the file until you both enable the file and enable filtering. That is, while the following two commands look about the same, they are different:

```
dheap -filter -enable MyFilter
dheap -filter -enable
```

The first command tells MemoryScape that it could use the information contained within the **MyFilter** filter. However, MemoryScape only uses it after you enter the second command.

Checking for Dangling Pointers: dheap -is_dangling:

The **dheap -is_dangling** command lets you determine if a pointer is still pointing into a deallocated memory block.

You can also use the **dheap -is_dangling** command to determine if an address refers to a block that was once allocated but has not yet been recycled. That is, this command lets you know if a pointer is pointing into deallocated memory.

Here's a small program that illustrates a dangling pointer:

```
main(int argc, char **argv)
{
    int *addr = 0;          /* Pointer to start of block.    */
    int *misaddr = 0;       /* Pointer to interior of block. */

    addr = (int *) malloc (10 * sizeof(int));
                                /* Point to interior of the block. */
    misaddr = addr + 5;

                                /* addr and misaddr now dangling. */
    free (addr);
    printf ("addr=%lx, misaddr=%lx\n",
           (long) addr, (long) misaddr);
}
```

If you set a breakpoint on the **printf()** statement and probe the addresses of **addr** and **misaddr**, the CLI displays the following:

```
d1.<> dheap -is_dangling 0x80496d0
           process:          0x80496d0
           1 (19405):        dangling

d1.<> dheap -is_dangling 0x80496e4
           process:          0x80496e4
           1 (19405):        dangling interior
```

This example is contrived. When creating this example, the variables were examined for their address and their addresses were used as arguments. In a realistic program, you'd find the memory block referenced by a pointer and then use that value. In this case, because it is so simple, using the CLI **dprint** command gives you the information you need. For example:

```
d1.<> dprint addr
addr = 0x080496d0 (Dangling) -> 0x00000000 (0)
```

```
d1.<> dprint misaddr
misaddr = 0x080496e4 (Dangling Interior) -> 0x00000000 (0)
```

If a pointer is pointing into memory that is deallocated, and this memory is being hoarded, the CLI also lets you know that you are looking at hoarded memory.

Detecting Leaks: dheap -leaks

The **dheap -leaks** command locates memory blocks that your program allocated and are no longer referenced. It then displays a report that describes these blocks; for example:

```
d1.<> dheap -leaks
process 1 (32188): total count 9, total bytes 450
* leak 1 -- total count 9 (100.00%), total bytes 450 (100%)
  -- smallest / largest / average leak: 10 / 90 / 50
: malloc PC=0x40021739 [./.../malloc_wrappers_dlopn.c]
: main PC=0x0804851e [./.../local_leak.cxx]
: __libc_start_main PC=0x40055647 [./lib/i686/libc.so.6]
: _start PC=0x080483f1 [./.../local_leak]
```

If you use the **-check_interior** option, MemoryScape considers a block as being referenced if a pointer exists to memory inside the block.

In addition to providing backtrace information, the CLI:

- Consolidates leaks made by one program statement into one leak report. For example, leak 1 has nine instances.
- Reports the amount of memory consumed for a group of leaks. It also tells you what percentage of leaked memory this one group of memory is using.
- Indicates the smallest and largest leak size, as well as telling you what the average leak size is for a group.

You might want to paint a memory block when it is deallocated so that you can recognize that the data pointed to is out-of-date. Tagging the block so that you can be notified when it is deallocated is another way to locate the source of problems.

Block Painting: dheap -paint

When your program allocates or deallocates a block, MemoryScape can paint the block with a bit pattern. This makes it easy to identify uninitialized blocks, or blocks pointed to by dangling pointers.

Here are the commands that enable block painting:

- **dheap -paint -set_alloc on**
- **dheap -paint -set_dealloc on**
- **dheap -paint -set_zalloc on**

Use the **dheap -paint** command to check the kind of painting that occurs and what the current painting pattern is. For example:

```
d1.<> dheap -paint
process: Alloc Dealloc AllocZero pattern pattern
1 (1012): yes yes no 0xa110ca7f 0xde110cf
```

Some heap allocation routines such as **calloc()** return memory initialized to zero. Using the **-set_zalloc_on** command allows you to separately enable the painting of the memory blocks altered by these kinds of routines. If you do enable painting for routines that set memory to zero, MemoryScape uses the same pattern that it uses for a normal allocation.

Here's an example of painted memory:

```
d1.<> dprint *(red_balls)
*(red_balls) = {
  value = 0xa110ca7f (-1592735105)
```

```
x = -2.05181867705792e-149
y = -2.05181867705792e-149
spare = 0xa110ca7f (-1592735105)
colour = 0xa110ca7f -> <Bad address: 0xa110ca7f>
}
```

The **0xa110ca7f** allocation pattern resembles the word “allocate”. Similarly, the **0xdea110cf** deallocation pattern resembles “deallocate”.

Notice that all of the values in the **red_balls** structure in this example aren't set to **0xa110ca7f**. This is because the amount of memory used by elements of the variable use more bits than the **0xa110ca7f** bit pattern. The following two CLI statements show the result of printing the **x** variable, and then casting it into an array of two integers:

```
d1.<> dprint (red_balls)->x
(red_balls)->x = -2.05181867705792e-149
d1.<> dprint {*(int[2]*)&(red_balls)->x}
*(int[2]*)&(red_balls)->x = {
  [0] = 0xa110ca7f (-1592735105)
  [1] = 0xa110ca7f (-1592735105)
```

(Diving in the GUI is much easier.)

You can tell MemoryScape to use a different pattern by using the following two commands:

- **dheap -paint -set_alloc_pattern *pattern***
- **dheap -paint -set_dealloc_pattern *pattern***

Red Zones Bounds Checking: dheap -red_zones

The Red Zones feature helps catch bounds errors and use-after-free errors. The basic idea is that each allocation is placed in its own page. An allocation is positioned so that if an overrun, that is, an access beyond the end of the allocation, is to be detected, the end of the allocation corresponds to the end of the page.

The page following that in which the allocation lies is also allocated, though access to this page is disabled. This page is termed the *fence*. Should the application attempt to access a location beyond the end of the allocation, that is, in the fence, the operating system sends the target a segment violation signal. This is caught by a signal handler installed by the HIA. The HIA examines the address that caused the violation. If it lies in the fence, then the HIA raises an overrun bounds error using the normal event mechanism.

If, however, the address does not lie in any region that the HIA “owns,” the HIA attempts to replicate what would have happened if the HIA’s signal handler were not in place. If the application had installed a signal handler, then this handler is called. Otherwise, the HIA attempts to perform the default action for the signal. It should be clear from this that the HIA needs to interpose the signal’s API to ensure that it always remains the installed handler as far as the operating system is concerned. At the same time, it needs to present the application with what it expects.

Underruns, or errors where the application attempts to read before the start of an allocation, are handled in a similar way. Here, though, the allocation is positioned so that its start lies at the start of the page, and the fence is positioned to precede the allocation.

One complication that arises concerns overrun detection. The architecture or definition of the allocation routines may require that certain addresses conform to alignment constraints. As a consequence, there may be a conflict between ensuring that the allocation's start address has the correct alignment, and ensuring that the allocation ends at the end of the page.

Use-after-free errors can also be detected. In this case, when the block is deallocated, the pages are not returned to the operating system. Instead, the HIA changes the state of the allocation's table entry to indicate that it is in the deallocated state, and then disables access to the page in which the allocation lies. This time, should the application attempt to access the block after it has been deallocated, a signal will be raised. Again, the HIA examines the faulting address to see what it knows about the address, and then either raises an appropriate event for TotalView, or forwards the signal on.

A key feature distinguishing TotalView Red Zones is that they can be engaged and disengaged at will during the course of the target's execution. The settings can be adjusted so that new allocations have different properties from existing allocations. Since Red Zones can be turned on or off, some of the application's requests can be satisfied by the Red Zones allocator, and others by the standard heap manager. The HIA keeps track of which allocator is responsible for, or owns, each block.

The **dheap -red_zones [-status [-all]]** option displays the current HIA Red Zone settings. By default, **dheap -red_zones** displays only those settings that can vary in the current mode, so that, for example, in overrun mode, the settings for fences and end positioning are not shown. The **dheap -red_zones -status -all** command will cause all settings to be shown, including those that are overridden for the current mode.

Please note that the abbreviation **-rz** can be used in the CLI for **-red_zones**.

The **dheap -red_zones -stats [<start_addr [<end_addr]]** option shows statistics relating to the HIA's Red Zones allocator for the optionally specified address range. If no range is specified the statistics are shown for the entire address space. These are:

- number of allocated blocks
- sum of the space requests received by the Red Zones allocator for allocated blocks
- sum of the space used for fences for allocated blocks
- overall space used for allocated blocks

The same set of statistics are also shown for deallocated blocks. In addition, the space used for each category is also shown as a percentage of the overall space used for Red Zones.

dheap -red_zones -info [<start_addr [<end_addr]] shows the Red Zones entries for allocations (and deallocations) lying in the optionally specified range. If no range is specified the entries are shown for the entire address space.

Red Zones is enabled using **dheap -red_zones -set on**, and disabled with **dheap -red_zones -set off**. **dheap -red_zones -reset** allows the HIA to determine its setting using the usual rules.

dheap -red_zones -set_mode sets the HIA in one of several Red Zone modes. When a new allocation is requested, the HIA will override the actual settings for some of the individual controls, and instead use values that correspond to that mode. The settings that are affected are: **pre-fence**, **post-fence**, and **end-positioning**. The other settings, like **use-after-free**, **exit value**, and **alignment**, take their values from the actual settings of those controls.

The Red Zone modes are:

- **dheap -red_zones -set_mode overrun**

The settings used are those that allow overruns to be detected. These are: **no** for **pre-fence**, **yes** for **post-fence**, and **yes** for **end positioned**.

- **dheap -red_zones -set_mode underrun**

The settings used are those that allow underruns to be detected. These are: **yes** for **pre-fence**, **no** for **post-fence**, and **no** for **end positioned**.

- **dheap -red_zones -set_mode unfenced**

The settings used are those that allow **use_after_frees** to be detected. These are: **no** for **pre-fence**, **no** for **post-fence**. **End positioned** is determined from the control's setting.

- **dheap -red_zones -set_mode manual**

All settings are determined from their actual values.

Use the **dheap -red_zones -set_pre_fence (on | off) | -reset_pre_fence** commands to adjust the pre-fence control. However, the setting is ignored unless the mode is manual.

Use the the **dheap -red_zones -set_post_fence (on | off) | -reset_post_fence** commands to adjust the post-fence control. However, the setting is ignored unless the mode is manual.

To enable the use-after-free control, enter **dheap -red_zones -set_use_after_free on**. To disable the control enter **"off"**. If enabled, any subsequent allocations will be tagged such that the allocation and its fences are retained when the block is deallocated. Access to the block is disabled when it is deallocated to allow attempts to access the block to be detected.

The alignment control **dheap -red_zones -set_alignment <integer>** regulates the alignment of the start address of a block issued by the Red Zones allocator. An alignment of zero indicates that the default alignment for the

platform should be used. An alignment of two ensures that any address returned by the Red Zones allocator is a multiple of two. In this case, if the length of the block is odd, then the end of the block will not line up with the end of the page containing the allocation. An alignment of one would be necessary for the end of the block to always correspond to the end of the page.

Adjusting the fence size is done through the **dheap -red_zones -set_fence_size <integer>** command. A fence size of zero indicates that the default fence size of one page should be used. If necessary, the fence size is rounded up to the next multiple of the page size. In most cases it should not be necessary to adjust this control. One instance where it may be useful, however, is where it is suspected that a bounds error is a consequence of a badly coded loop, and the stride of the loop is large. In such a case, a larger fence may be helpful.

dheap -red_zones -set_end_aligned (on | off) controls whether the allocation is positioned at the end of the containing page or at its start. The control in the HIA is always updated, though the actual value is ignored in overrun and underrun modes.

Use **dheap -red_zones -set_exit_value <integer>** to adjust the exit value used if the HIA terminates the target following detection of a Red Zone error. Generally, the application would fail if it is allowed to continue after a Red Zone error has been detected. In order to allow some control over the application's exit code, the HIA will call exit when an error is detected. The value it passes to exit as a termination code can be controlled, so that if the application is run from scripts the cause for the termination can be determined.

The **dheap -red_zones -size_ranges ...** option for Red Zones allows the user to restrict the use of Red Zones to allocations of specified sizes. If Red Zones are engaged and size ranges are enabled, the Red Zones allocator will be used if the size of the request lies in one of the defined size ranges. A value is deemed to lie in a range if $\text{start} \leq \text{size} \leq \text{end}$.

To make typing a bit easier, **-size_ranges** can be abbreviated to **-sr**.

A range having an end of 0 is interpreted as having no upper limit. Thus if the end is 0, the size matches the range if it is at least as large as the start.

The HIA supports a number of size ranges. This allows particular ranges of sizes to be included or excluded. The Red Zones allocator is used if the size of the request lies in any one of these ranges. The HIA does not check to see that ranges don't overlap or are otherwise consistent.

The determination of whether the Red Zones allocator should be used is made at the time of the original allocation. Thus, once an allocator has taken ownership of a block, that allocator is used for the remainder of the block's life. In particular, all **realloc** operations are handled by the same allocator, irrespective of the size range settings at the time of reallocation.

There are two attributes associated with each range. The first is the **"in_use"** attribute. This is ignored by the HIA, and is provided for the benefit of TotalView. The motivation here is to allow TotalView to keep the state that would otherwise be lost if the target is detached, and then reattached to later.

The second attribute is the **"active"** attribute. This indicates if the size range is active, and therefore whether it is used by the HIA when determining whether the Red Zones allocator should be used.

The TotalView cli command **dheap -red_zones -size_ranges -set on** enables size ranges, and **dheap -red_zones -size_ranges -set off** disables size ranges. If size ranges are disabled, but Red Zones are enabled, the Red Zones allocator will be used for all allocations.

dheap -red_zones -size_ranges -reset unsets the TotalView setting for the enable/disable control.

The **dheap -red_zones -size_ranges -status [-all] <id_range>** command shows the current settings of the size ranges. The absence of an **<id_range>** is equivalent to an ID range of "0:0". By default, only "in_use" size ranges are displayed. To display all known ranges, specify **-all**. **<id_range>** must be in one of the following formats:

x:y = id's from x to y

:y = id's from 1 to y

x: = id of x and higher

x = id is x

To set a size range identified by **<id>** to a particular size range the **dheap -red_zones -size_ranges -set_range <id> <size_range>** command is used. **<size_range>** must be in one of the following formats:

x:y = allocations from x to y

:y = allocations from 1 to y

x: = allocations of x and higher

x = allocations of x

To reset an id or range of ids use **dheap -red_zones -size_ranges -reset_range <id_range>**. **<id_range>** must be in a format defined above.

The **dheap -red_zones -size_ranges -set_in_use (on | off) <id_range>** option adjusts the “in_use” attribute of all the size ranges whose ids lie within **<id_range>**. **dheap -red_zones -size_ranges -set_active (on | off) <id_range>** adjusts the “active” attribute of all the size ranges whose ids lie within **<id_range>**.

The following Red Zones command options unset the TotalView settings for these controls:

```
dheap -red_zones -reset_mode
dheap -red_zones -reset_pre_fence
dheap -red_zones -reset_post_fence
dheap -red_zones -reset_use_after_free
dheap -red_zones -reset_alignment
dheap -red_zones -reset_fence_size
dheap -red_zones -reset_exit_value
dheap -red_zones -reset_end_aligned
```

When the above commands are entered, the HIA will determine its settings using the values in the **TVHEAP_ARGS** environment variable, the HIA configuration file, or its default values.

Deallocation Notification: dheap -tag_alloc

You can tell MemoryScape to tag information within MemoryScape’s tables and to notify you when your program either frees a block or passes it to **realloc()** by using the following two commands:

- **dheap -tag_alloc -notify_dealloc**
- **dheap -tag_alloc -notify_realloc**

Tagging is done within MemoryScape’s agent. It tells MemoryScape to watch those memory blocks. Arguments to these commands tell MemoryScape which blocks to tag. If you do not type address arguments, TotalView notifies you when your program frees or reallocates an allocated block. The following example shows how to tag a block and how to see that a block is tagged:

```
d1.<> dheap -tag_alloc -notify_dealloc 0x8049a48
process 1 (19387): 1 record(s) update
d1.<> dheap -info
process 1 (19387):
    0x8049a48 -- 0x8049b48 0x100 [ 256]
    flags: 0x2 (notify_dealloc)
    0x8049b50 -- 0x8049d50 0x200 [ 512]
    flags: 0x0 (none)
    0x8049d58 -- 0x804a058 0x300 [ 768]
    flags: 0x0 (none)
```

Using the **-notify_dealloc** subcommand tells MemoryScape to let you know when a memory block is freed or when **realloc()** is called with its length set to zero. If you want notification when other values are passed to the **realloc()** function, use the **-notify_realloc** subcommand.

After execution stops, here is what the CLI displays when you type another **dheap -info** command:

```
d1.<> dheap -info
process 1 (19387):
    0x8049a48 -- 0x8049b48 0x100 [ 256]
    flags: 0x3 (notify_dealloc, op_in_progress)
    0x8049b50 -- 0x8049d50 0x200 [ 512]
    flags: 0x0 (none)
    0x8049d58 -- 0x804a058 0x300 [ 768]
```

TVHEAP_ARGS

Environment variable for presetting MemoryScape values

When you start TotalView, it looks for the **TVHEAP_ARGS** environment variable. If it exists, TotalView reads values placed in it. If one of these values changes a MemoryScape default value, MemoryScape uses this value as the default.

If you select a **<Default>** button in the GUI or a reset option in the CLI, MemoryScape resets the value to the one you set here, rather than to its default.

TVHEAP_ARGS Values

The values that you can enter into this variable are as follows:

display_a_llocations_on_exit=boolean

Tells MemoryScape to dump the allocation table when your program exits. If your program ends because it received a signal, MemoryScape might not be able to dump this table.

backtrace_depth=depth

Sets the backtrace depth value. See “[Showing Backtrace Information: dheap -backtrace:](#)” on page 33 for more information.

backtrace_trim=trim

Sets the backtrace trim value. See “[Showing Backtrace Information: dheap -backtrace:](#)” on page 33 for more information.

enable_event_filtering=boolean

notify_free_not_allocated=boolean

notify_realloc_not_allocated=boolean

notify_addr_not_at_start=boolean

notify_double_alloc=boolean

notify_guard_corruption=boolean

notify_alloc_not_in_heap=boolean

notify_alloc_null=boolean

notify_alloc_returned_bad_alignment=boolean

notify_bad_alignment_argument=boolean

=boolean

notify_realloc=boolean

notify_double_dealloc=boolean

Same meanings as **dheap -event_filter**.

enable_guard_blocks=boolean

guard_max_size=positive-integer

guard_pre_size=positive-integer

guard_post_size=positive-integer

guard_pre_pattern=integer

guard_post_pattern=integer

Same meanings as **dheap -guard**.

enable_hoarding=boolean

hoard_all_blocks=boolean

hoard_maximum_num_blocks=positive-integer

hoard_maximum_kb=positive-integer

Same meanings as **dheap -hoard**.

enable_hoard_autoshrink=boolean

hoard_autoshrink_threshold_kb=positive-integer

hoard_autoshrink_trigger_count=positive-integer

enable_red_zones=boolean

enable_rz_size_ranges=boolean

rz_alignment=positive-integer

rz_detect_underrun=boolean

rz_detect_overrun=boolean

rz_detect_use_after_free=boolean

rz_end_alligned=boolean

rz_exit_val=integer

rz_fence_size=positive-integer

rz_mode=overrun | underrun | unfenced | manual

rz_size_range=positive-integer,positive-integer,positive-integer

rz_size_range_enable=positive-integer,boolean

Same meanings as with **dheap -red_zones**.

memalign_strict_alignment_even_multiple

MemoryScape provides an integral multiple of the alignment rather than the even multiple described in the Sun **memalign** documentation. By including this value, you are telling MemoryScape to use the Sun alignment definition. However, your results might be inconsistent if you do this.

output_fd=*int***output_file=***pathname*

Sends output from MemoryScape to the file descriptor or file that you name.

paint_on_alloc=*boolean***paint_on_dealloc=***boolean***paint_on_zalloc=***boolean***paint_alloc_pattern=***integer***paint_dealloc_pattern=***integer*

Same meanings as with **dheap -paint**

verbosity=*int*

Sets MemoryScape's verbosity level. If the level is greater than 0, MemoryScape sends information to **stderr**. The values you can set are:

0: Display no information. This is the default.

1: Print error messages.

2: Print all relevant information.

This option is most often used when debugging MemoryScape problems.

Setting the TotalView **VERBOSE** CLI variable does about the same thing.

Example

For more than one value, separate entries with spaces and place the entire entry within quote. For example:

```
setenv TVHEAP_ARGS="output_file=my_file backtrace_depth=16"
```

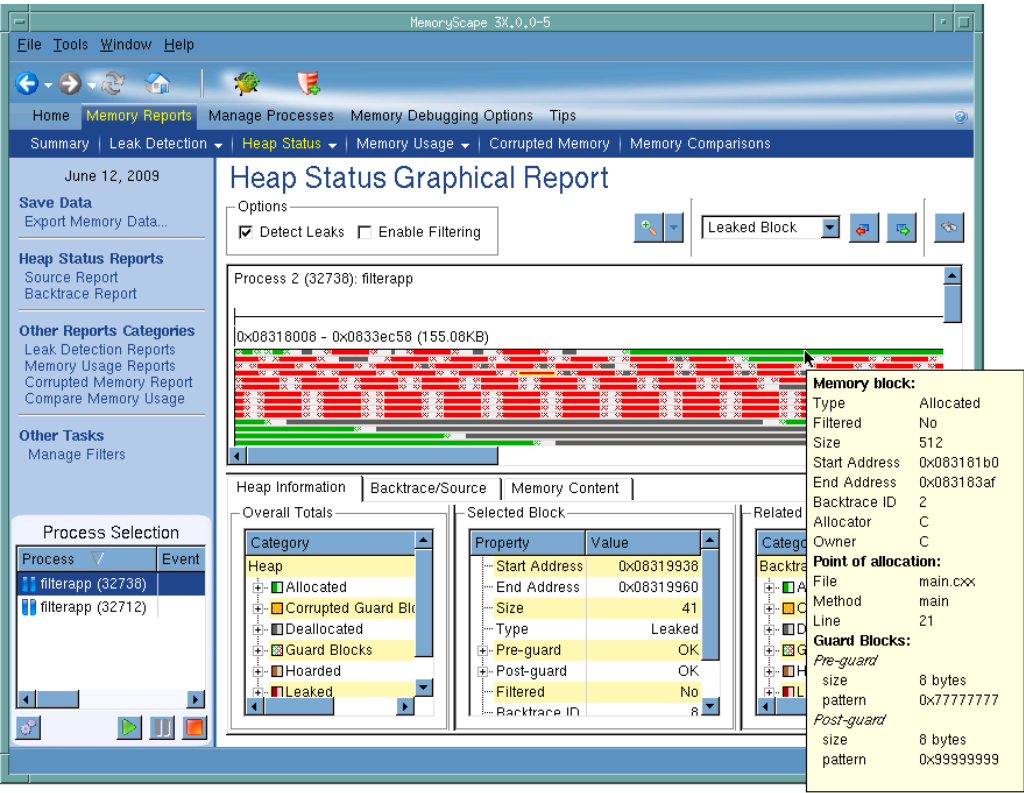
Examining Memory

So far, you've been reading about memory errors. If only things were this simple. The large amount of memory available on a modern computer and the ways in which an operating system converts actual memory into virtual memory may hide many problems. At some point, your program can hit a wall—thrashing the heap to find memory it can use or crashing because, while memory is available, the operating system can't find a block big enough to contain your data. In these circumstances, and many others, you should examine the heap to determine how your program is managing memory.

NOTE >> MemoryScape can display a lot of information, at times too much. You can simplify MemoryScape reports using a filter that suppresses the display of information. For more information on filters, see [Task 10: “Filtering Reports”](#) on page 103.

Begin analyzing data by displaying the **Memory Reports | Heap Status | Graphical Report**. After generating the report, MemoryScape displays a visual report of the heap, [Figure 27](#).

Figure 27: Heap Status Graphical Report



If you place the mouse cursor over a block, MemoryScape displays information about this block in a tooltip.

The display area has three parts. The small upper area contains controls that specify which content MemoryScape displays. The middle contains many bars, each of which represents one allocation. The bar's color indicates if the memory block is allocated, deallocated, leaked, or in the hoard.

The bottom area has three divisions.

- The **Overall Totals** area summarizes the kinds of information displayed within the top area as well as providing a key to the colors used when drawing the blocks.
- If you select a block in the top area, the **Selected Block** area (obscured by the information pop-up) contains information about this block. When you select a block, MemoryScape highlights it within the top area.

- The **Related Blocks** area on the right displays how many other blocks your program allocated from the same location. (Actually, this just shows how many allocations had the same backtrace. If your program got to the same place in different ways, each different way would have a different backtrace, so MemoryScape doesn't consider them related.)

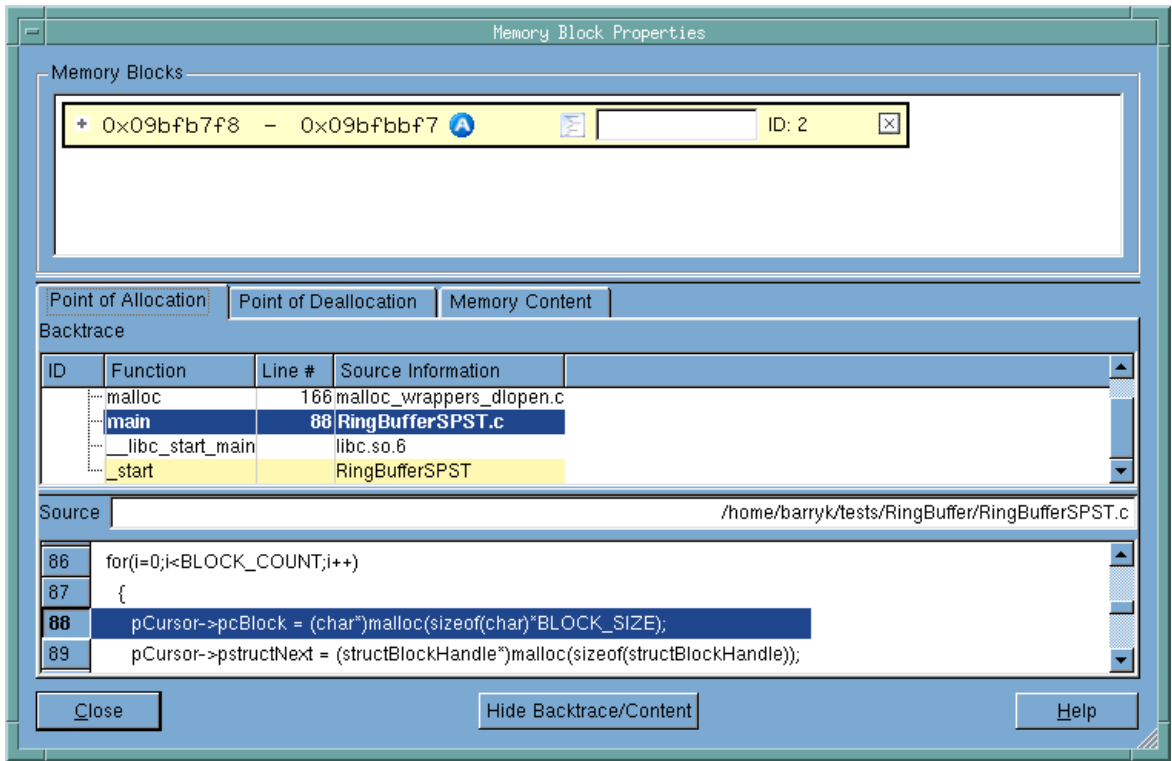
Now that you have this information, you can begin making decisions. Obviously, you should fix the leaks. If there were a lot of small blocks, is your program allocating memory too frequently? Should it be allocating memory in larger blocks and managing the allocated memory directly? Is there a pattern of allocations and deallocations that prevents reuse?

NOTE >>Memory managers tend to be lazy. Unless they can easily reuse memory, they just get more. If you use the Memory Usage Page to monitor how your program is using memory, you'll probably find that your program only gets bigger. Once your program grabs memory from the operating system, it doesn't usually give it back. And, while it could reuse this memory if your program deallocates the block, it is easier and quicker just to grab new memory.

Block Properties

In many places within MemoryScape, you can right click on a displayed block and select **Properties**, which launches the Memory Block Properties dialog, [Figure 28](#).

Figure 28: Memory Block Properties



MEMORY BLOCKS: Contains a list of all memory blocks for which you have requested property information. Notice the + symbol. When selected, MemoryScape displays more information about the block. Click on parts of the third picture in this help topic for information.

POINT OF ALLOCATION: When selected, MemoryScape displays information it has collected about the memory block at the time when it was allocated. Click over other areas in this picture for information.

POINT OF DEALLOCATION: When selected, MemoryScape displays information it has collected about the memory block at the time when it was deallocated. If this tab is empty, the block has not yet been deallocated. The information displayed in this area is the same as the **Point of Allocation** tab.

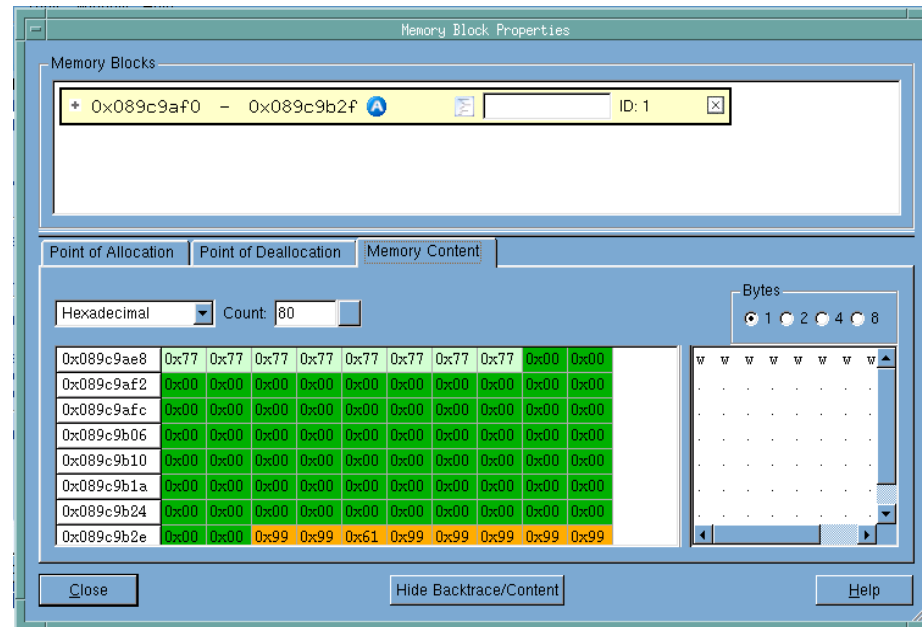
MEMORY CONTENTS: When selected, MemoryScape displays information about the bytes contained within the block. Click on parts of the second picture in this help topic for information.

HIDE BACKTRACE/CONTENT: Hides the bottom part of this window so that only the Memory Blocks area is visible.

Memory Contents Tab

When you click the Memory Contents tab, MemoryScape displays the contents of this memory block, [Figure 29](#), in a manner similar to that used in the shell `od` command.

Figure 29: Memory Block Properties, Memory Contents tab



MemoryScape displays more information about the block. Click on parts of the third picture in this help topic for information.

DISPLAY FORMAT: Tells MemoryScape to display information in one of the following ways: **Hexadecimal**, **Decimal**, **Unsigned Decimal**, **Octal**, **Character**, **Float**, **Binary**, or **Address**. How many characters and the size of the blocks are set using the **Count** and **Bytes** controls.

COUNT: Controls how many memory blocks MemoryScape will display.

BYTES: Specifies the number of bytes to display in each block.

CONTENTS AREA: The left column names the first memory address being displayed in the remaining columns in the row. The colors used to display blocks indicates their status, as follows:

- Corrupted Guard Blocks
- Allocated
- Deallocated
- Guard Blocks
- Hoarded
- Leaked

CHARACTER DISPLAY AREA: If the contents of a memory address can be interpreted as a character, it is displayed in this area.

CLOSE: Closes this window

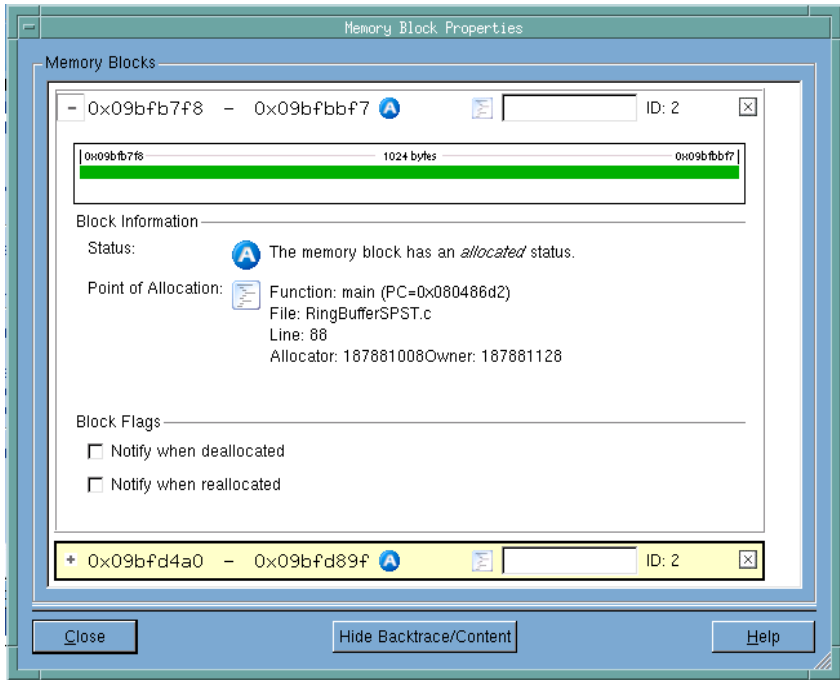
HIDE BACKTRACE/CONTENT: Hides the bottom part of this window so that only the Memory Blocks area is visible.

HELP: Tells MemoryScape to display Help text.

Additional Memory Block Information

If you expand the top area of the Block Properties window manually or if you click the Hide Backtrace/Content button, you can see additional information about the memory block, [Figure 30](#).

Figure 30: Memory Block Properties, Hide Backtrace/Content



BLOCK: Lists the starting and ending address of each block, its status—means allocated. As this window contains an entry for each block for which you've requested properties, more than one block can be displayed, as is shown here.

COMMENT AREA: As an aid to remember which block is which in this window, add a comment to the block.

GRAPHIC DISPLAY: Shows a display of the blocks similar to how it is shown in the Heap Status Graphics display and other places where memory is displayed graphically. In this example, guard blocks were used, and they are indicated by the lighter green area at the ends of the block

BLOCK INFORMATION: Contains status information about the block.

BLOCK FLAGS: Selecting a checkbox tells MemoryScape that it should stop execution and notify you when the block is deallocated or reallocated.

CLOSE: Closes this window.

SHOW BACKTRACE/CONTENT: Shows the Backtrace/Content part of this window that was previously concealed.

HELP: Displays Help text.

Filtering

You can remove information from Backtrace and Source reports by adding a filter. For example, suppose you do not want MemoryScape to show blocks related to a standard function such as **strdup()**. By creating and applying a filter, MemoryScape removes this function's information from reports. The exception is the **Heap Status Graphical Report**. In this report, filtered blocks are displayed using a lighter version of their ordinary color.

Filtering simplifies the display so you can more easily focus on problems. For example, filtering allows you to remove leaks originating in libraries for which you have no control.

Filtering by a function name is just one option. For more information, see [Task 10: "Filtering Reports"](#) on page 103.

Using Guard Blocks

When a program allocates a memory block, it is responsible for not writing data outside the block. For example, if you allocate 16 bytes, you do not want to write 32 bytes of information into it because this data would corrupt the information contained in the next block.

MemoryScape can help you detect problems related to writing data either before or after a block by surrounding blocks with a small amount of additional memory. It will also write a pattern into this memory. These additional memory blocks are called *guard blocks*. If your program writes data into these blocks, MemoryScape can tell that a problem occurred, as follows:

- When you are displaying a Heap Status report, you can ask for a Corrupted Guard Blocks report. The Heap Status report also shows the guard regions and corrupted blocks.
- When your program deallocates memory, MemoryScape can check the deallocated block's guards. If they've been altered—that is, if your program has written data into them—MemoryScape can stop execution and alert you to the problem.

For example, suppose your program allocates 16 bytes and you write 64 bytes of data. While the first 16 bytes are correctly written, the remaining 48 aren't. This write request will also overwrite the guard blocks for the current

block and the block that follows, as well as some of the next block's data. That is, you will have inadvertently changed some data—data that when accessed will be incorrect.

Asking for notification when the block is deallocated lets you know that a problem has occurred. Because you now know which block was corrupted, you can begin to locate the cause of the problem. In many cases, you will rerun your program, focusing on those blocks.

Using Red Zones

NOTE >> Red Zone controls appear on the toolbar only on platforms that support Red Zone capabilities. Memory debugging Red Zones are supported on Linux, Solaris, and Mac OS X.

As discussed in [“Using Guard Blocks”](#) on page 51, when a program allocates a memory block, it is responsible to not write or read data outside the block. Red Zones give you another tool, like guard blocks, for detecting access violations.

MemoryScape can immediately detect when your program oversteps the bounds of your allocated memory by protecting it with a Red Zone, a page of memory placed either before or after your allocated block. If your program tries to read or write in this Red Zone, MemoryScape halts the execution of your program and notifies you. When the Red Zone is placed before the block, MemoryScape detects underruns; conversely, if the Red Zone is placed after the block, it detects overruns. It can also detect if your program accesses the block after it has been freed.

Since your program halts when MemoryScape detects an overrun or under-run, you know exactly where it overstepped the bounds of the block. The event information shows the current stack trace, to allow you to pinpoint where the event occurred. It also shows where the corrupted block was allocated and deallocated, if appropriate.

Using Red Zones can significantly increase your program’s memory consumption, so MemoryScape provides ways to limit their use. You can turn them on and off at any time during your program’s execution using the Red Zone button on the toolbar. You can also restrict Red Zone use by size range,

defining a range such that only block sizes within that range will be protected by Red Zones. You can use multiple ranges collectively to exclude allocations of a particular size.

NOTE >> For a detailed description of how Red Zones work, see [“Red Zones Bounds Checking: dheap -red_zones”](#) on page 38.

Using Guard Blocks and Red Zones

Guard blocks and Red Zones complement each other in several ways and can be used together to find your memory corruption problem. While Red Zones have a high memory consumption overhead, they provide immediate notification when an illegal read or write occurs. Guard blocks add minimal overhead, but detect only illegal writes, and report corruptions only when the block is deallocated or when you generate a report. Finding a memory problem with MemoryScape may require a couple of passes to narrow down your problem.

Start by enabling guard blocks prior to running your program. You can run a Corrupt Memory report at any time to see whether you have any corrupted blocks. The report shows the blocks that are corrupt and where they were initially allocated. In your next run, turn off guard blocks and turn Red Zones on. If memory is tight, enable Red Zones only where needed, either manually or by using size ranges. MemoryScape should detect when a block is overwritten and stop execution.

A caveat here: the layout of memory is controlled by the heap manager and the operating system. Depending on the size of your allocated block and its alignment on the page of memory, there may be a gap between the block and the Red Zone. The overrun or underrun must be large enough to span the gap and reach the Red Zone, or MemoryScape will not generate an event. This is a potential issue only if you are using **memalign** or **posix_memalign**. For **malloc** and **calloc**, the gap will probably be one less than the

size of the alignment, three or seven bytes. In any case, the block will still show up as having a corrupted guard block, because guard blocks are placed immediately before and after your allocated block without any gap.

Block Painting

Your program should initialize memory before it is used, and it should never use memory that is deallocated. MemoryScape can help you identify these kinds of problems by writing a bit pattern into memory as it is allocated or deallocated. You can either specify a pattern or use the default, as follows:

- The default allocation pattern is **0xa110ca7f**, which was chosen because it resembles the word “allocate”.
- The default deallocation pattern is **0xdea110cf**, which was chosen because it resembles the word “deallocate”. In most cases, you want MemoryScape to paint memory blocks when your program allocates them.

If your program displays this value, you’ll be able to tell what the problem is. In some cases, using these values will cause your program to crash. Because MemoryScape traps this action, you can investigate the cause of the problem.

You can turn painting on and off without restarting your program. If, for example, you change the deallocation pattern, you’ll have a better idea when your program deallocated the block. That is, because MemoryScape is using

a different pattern after you change it, you will know if your program allocated or deallocated the memory block before or after you made the change.

If you are painting deallocated memory, you could be transforming a working program into one that no longer works. This is good, as MemoryScape will be telling you about a problem.

Hoarding

You can stop your program's memory manager from immediately reusing memory blocks by telling MemoryScape to hoard (that is, retain) blocks. Because memory blocks aren't being immediately reused, your program doesn't immediately overwrite the data within them. This allows your program to continue running with the correct information even though it is accessing memory that should have been deallocated. Because it has been hoarded, the data within this memory is still correct. If this weren't the case, any pointers into this memory block would be dangling. In some cases, this uncovers other errors, and these errors can help you track down the problem.

If you are painting and hoarding deallocated memory (and you should be), you might be able to force an error when your program accesses the painted memory.

MemoryScape holds on to hoarded blocks for a while before returning them to the heap manager so that the heap manager can reuse them. As MemoryScape adds blocks to the hoard, it places them in a first-in, first-out list. When the hoard is full, MemoryScape releases the oldest blocks back to your program's memory manager.

To hoard all deallocated memory, set the maximum KB and blocks to **unlimited** by entering **0** in the hoarding control fields. To prevent or delay your program from running out of memory when you use this setting, use the advanced option to set MemoryScape to automatically release hoarded memory when available memory gets low.

You can also set a threshold for the hoard size so MemoryScape can warn you when available memory is getting low. If the hoard size drops below the threshold, MemoryScape halts execution and notifies you. You can then view a Heap Status or Leak report to see where your memory is being allocated.

Example 1: Finding a Multithreading Problem

When a multithreaded program shares memory, problems can occur if one thread deallocated a memory block while another thread is still using it. Because threads execute intermittently, problems are also intermittent. If you hoard memory, the memory will stay viable for longer because it cannot be reused immediately.

If intermittent program failures stop occurring, you know what kind of problem exists.

One advantage of this technique is that you can relink your program (as is described in [Creating Programs for Memory Debugging](#), on page 121) and then run MemoryScape against a production program that was not compiled using the **-g** compiler debugging option. If you see instances of the hoarded memory, you'll instantly know problems have occurred.

This technique often requires that you increase the number of blocks being hoarded and the hoard size.

Example 2: Finding Dangling Pointer References

Hoarding is most often used to find dangling pointer references. Once you know the problem is related to a dangling pointer, you need to locate where your program deallocated the memory. One technique is to use block tag-


ging (see [Task 6: “Using Runtime Events”](#) on page 90). Another is to use block painting to write a pattern into deallocated memory. If you also hoard painted memory, the heap manager will not be able to reallocate the memory as quickly.

If the memory was not hoarded, the heap manager could reallocate the memory block. When it is reallocated, a program can legitimately use the block, changing the data in the painted memory. If this occurs, the block is both legitimately allocated and its contents are legitimate in some context. However, the older context was destroyed. Hoarding delays the recycling of the block. In this way, it extends the time available for you to detect that your program is accessing deallocated memory.

Debugging with TotalView

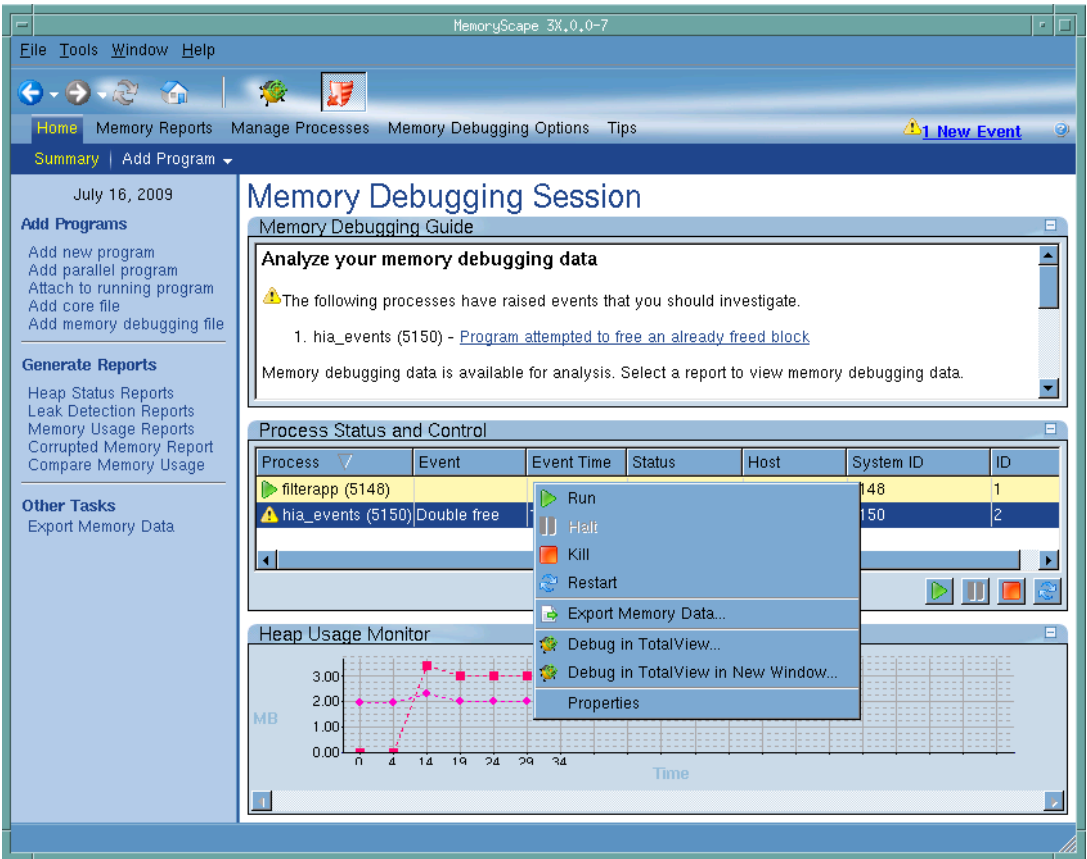
You may find that you want to exert greater control over your process execution than MemoryScape provides, or you may want to examine variables as you go. To do so, MemoryScape can bring up the TotalView Process Window.

There are two ways to debug with TotalView:

- Select a process and use the pop-up menu options Debug in TotalView or Debug in TotalView in New Window, [Figure 31](#).
- Use the icon in the MemoryScape toolbar: 

Be aware that opening the TotalView Process Window from within MemoryScape does not initialize TotalView in the same way as starting TotalView directly. The definitions in your .tvdrf file and your saved breakpoints are not loaded. However, you can load a breakpoint file using the Action Point menu item in the Process Window. If you need the definitions in your .tvdrf file, start TotalView first and open MemoryScape from within TotalView.

Figure 31: Memory Debugging Session Menu



Memory Tasks

This chapter describes the tasks that you can perform using MemoryScape. While each of these tasks can be read separately, you might want to skim them in the order in which they are presented when you are learning MemoryScape.

- [Task 1: “Getting Started”](#) on page 61
- [Task 2: “Adding Parallel Programs”](#) on page 69
- [Task 3: “Setting MemoryScape Options”](#) on page 71
- [Task 4: “Controlling Program Execution”](#) on page 82
- [Task 5: “Seeing Memory Usage”](#) on page 86
- [Task 6: “Using Runtime Events”](#) on page 90
- [Task 7: “Graphically Viewing the Heap”](#) on page 94
- [Task 8: “Obtaining Detailed Heap Information”](#) on page 97
- [Task 9: “Seeing Leaks”](#) on page 102
- [Task 10: “Filtering Reports”](#) on page 103
- [Task 11: “Viewing Corrupted Memory”](#) on page 108
- [Task 12: “Saving and Restoring Memory State Information”](#) on page 111
- [Task 13: “Comparing Memory”](#) on page 113

- [Task 14: “Saving Memory Information as HTML”](#) on page 116
- [Task 15: “Hoarding Deallocated Memory”](#) on page 118
- [Task 16: “Painting Memory”](#) on page 119

The tasks described in the chapter assume that you are familiar with the memory debugging concepts presented in [“Locating Memory Problems”](#). If you haven't yet read that chapter, you should read it before trying to understand or perform any of the tasks described in this chapter.

Task 1: Getting Started

This task shows you how to start your memory debugging session. It also presents an overview of the kinds of activities you might perform.

The sections within this task are:

- “Starting MemoryScape” on page 61
- “Adding Programs and Files to MemoryScape” on page 65
- “Attaching to Programs and Adding Core Files” on page 66
- “Stopping Before Finishing Execution” on page 66
- “Exporting Memory Data” on page 66
- “MemoryScape Information” on page 67
- “Where to Go Next” on page 67

Starting MemoryScape

There are five different ways to start debugging memory:

1. Display the MemoryScape window by typing:

memscape

After MemoryScape displays its opening screen, you can begin adding programs and files to your memory debugging session. If you are running MemoryScape on a Macintosh, you can double-click on the program icon. [Figure 33](#) on page 63 shows starting MemoryScape from a shell window and parts of screens you’ll use before you actually start debugging memory.

For additional information, see [Adding Programs and Files to MemoryScape](#).

2. Directly invoke MemoryScape upon a file by typing:

memscape program_name

MemoryScape responds by displaying its **Home | Summary** screen. You can now use the run control to start your program. ([Figure 34](#) on page 64.)

If your program needs command-line options, you can add them by right-clicking on the program’s name and selecting **Properties**, [Figure 32](#).

Figure 32: Properties Dialog Box

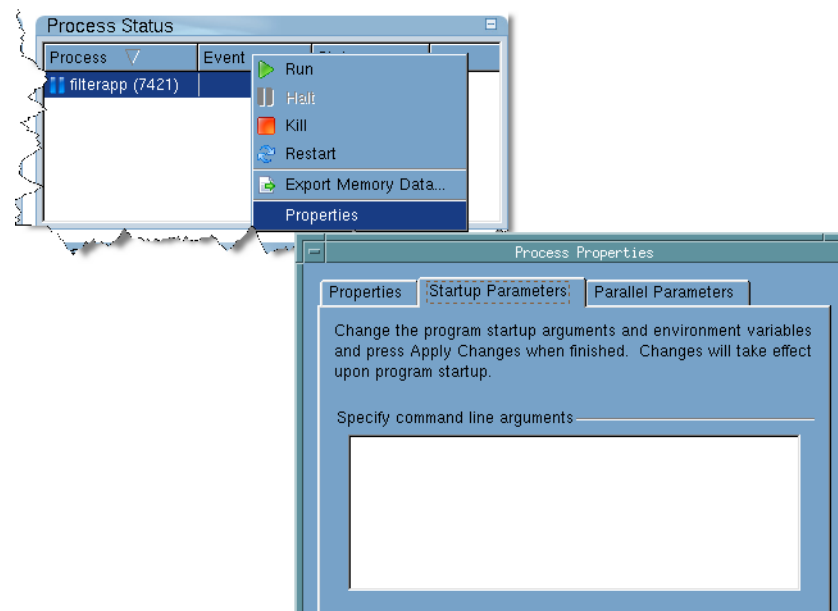


Figure 33: Starting MemoryScape

I Programs to Your MemoryScape Session

Programs, attach to programs, add core & memory debugging files to this session by selecting options.

[Add new program](#)

Specify a new program to be loaded.

[Add parallel program](#)

Specify a new parallel program to be loaded.

[Attach to running program](#)

Specify a program running on this computer.

[Add core file](#)

Specify a core file generated from a memory debugging enabled process to be analyzed.

[Add memory debugging file](#)

Specify a memory debugging data file to be loaded and analyzed.

Memory Reports Manage Processes Memory Debugging Options Tips

Summary | Add Program

June 11, 2009

Adding Programs

Enter core file

Analyze data using
memory reports
debugging

▼

Provides event notifications and leak detection. It allows the best performance for your process.

diagram

Provides corrupted memory detection by applying guard blocks. Performance may degrade slightly.

h

Provides memory over run alerts by monitoring Red Zone violations. Your memory consumption will increase significantly.

reme

Enables all options. There is a risk that performance may suffer and you will use more memory.

```
417 % pwd
/nfs/netapp0/user/home/barryk/test
stovepipe.etnus.com:/home/barryk/t
418 % memscape
```

MemoryScape 3X.0.0-5

File Tools Window Help

Home Memory Reports Manage Processes Memory Debugging Options Tips

Summary | Add Program

June 23, 2009

Add Programs

- Add new program
- Add parallel program
- Attach to running program
- Add core file
- Add memory debugging file

Generate Reports

- Heap Status Reports
- Leak Detection Reports
- Memory Usage Reports
- Corrupted Memory Report
- Compare Memory Usage

Other Tasks

- Export Memory Data

Memory Debugging Session

Memory Debugging Guide

Analyze your memory debugging data

Memory debugging data is available for analysis. Select a report to view memory debugging data.

- [Leak detection source report](#)
- [Heap graphical report](#)
- [Heap status source report](#)
- [Memory debugging reports summary](#)

Process Status and Control

Process	Rank	Event	Event Time	Status	Host
Parallel Job random.0 (MPI)					
MPI_COMM_WORLD					
random.15	15			Stopped	<local>

Heap Usage Monitor

Graph showing memory usage (MB) over time (0 to 114). The y-axis ranges from 0.85 to 3.85 MB. The x-axis ranges from 0 to 114. The graph shows a steady increase in memory usage over time.

Memory Debugging Options

Customize your options below or press *Basic Options* for predefined settings.

☒ **Enable memory debugging**

☒ **On a memory event, halt execution.**

On process events, you can halt execution, generate a core file or generate a lightweight memory file. Use the **Advanced** button to control actions for individual events.

If you change these arguments after program execution starts, restart your program for the changes to take effect.

3. Directly invoke MemoryScape, adding needed program arguments.

memscape *program_name* -a arguments

Type all your program's arguments after the **-a**. The next time you invoke MemoryScape on your program, these arguments will automatically be used.

MemoryScape displays its **Home | Summary** screen. (Figure 34.) Information on changing these options is discussed in the previous bullet.

4. Directly invoke MemoryScape, adding needed command-line options.
Here are two skeletons on starting MemoryScape:

memscape [*program_name* [*memscapes_options*] [-a arguments]]

memscape *memscapes_options*

MemoryScape options are typed either first or second, depending on if you are also naming a file. In all cases, arguments to your program must be the last arguments on the command line.

For more on options, see [MemoryScape Command-Line Options](#), on page 150.

5. Invoke MemoryScape in batch mode, as follows:

memscript options

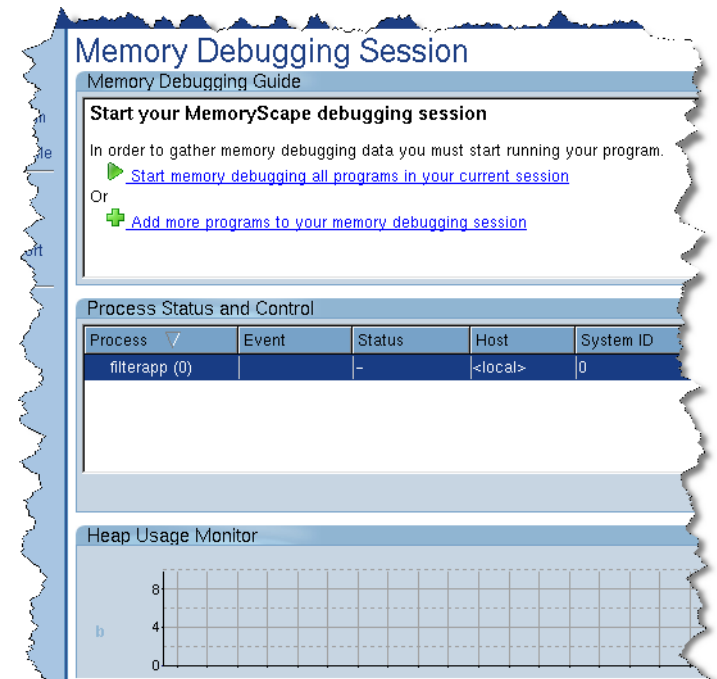
For more on batch mode, see [MemoryScape Scripting](#), on page 146.

Before you begin program execution, you may want to configure MemoryScape for the different kinds of activities that it can perform. For more information, see [Task 3: "Setting MemoryScape Options"](#) on page 71. In most cases, the default options will meet your needs.

MemoryScape can provide memory information about a process only when the process is stopped. Therefore, when you ask for a report, MemoryScape stops execution. MemoryScape also stops execution when you ask for memory usage information. However, it quickly restarts execution after it collects usage information.

You will probably use the commands on the **Home | Summary** screens to stop and restart execution. The graph here either starts MemoryScape or adds more programs, [Figure 34](#).

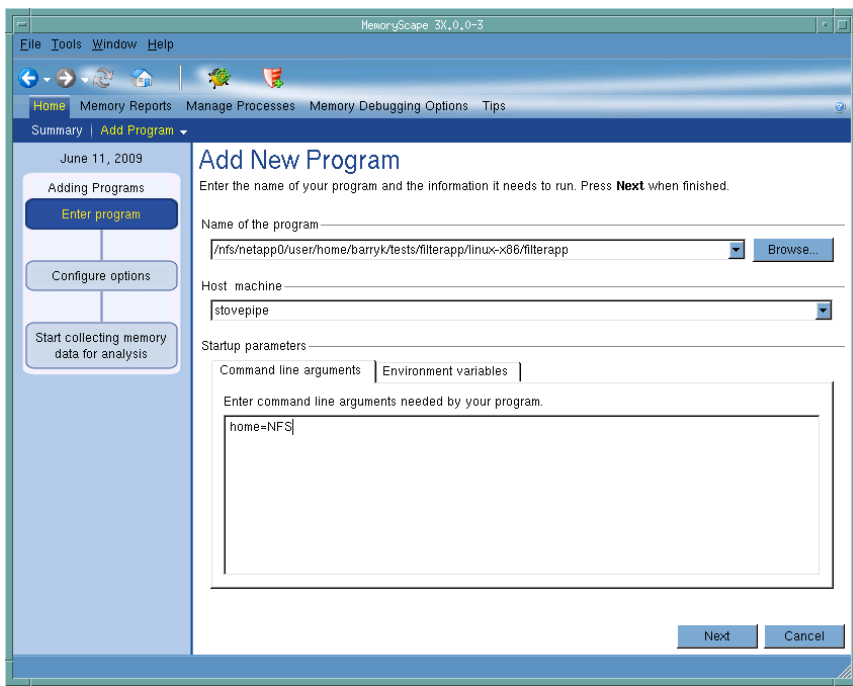
Figure 34: MemoryScape Home Page



Adding Programs and Files to MemoryScape

To add programs, select the link on the **Add Programs to Your MemoryScape Session** page. This page is automatically displayed if you do not name a program when you invoke MemoryScape. In addition, you can display this screen by selecting **Home | Add Program**. In general, to add a program, just enter the information requested on a screen. [Figure 35](#) shows the screen displayed when you select **Add new program**.

Figure 35: Add New Program Screen



Notice the left portion of the screen. If adding a file has more than one operation, the graphic displays the progression.

You can enter these kinds of information:

- The name of your program. Either a relative or absolute path name. In addition, you can use the **Browse** button to locate the file.
- The name of the computer on which your program will execute. In most cases, you'll be running your program on the same machine as you used to invoke MemoryScape.

You can either type the name of the machine on which your program will execute or select a previously named machine. The machine entered must have the same architecture as the machine on which you are running MemoryScape. For example, if you are running MemoryScape on a linux-x86-64 machine, you cannot name a Linux-arm64system.

- Any command-line arguments. Arguments that you would use when invoking your program without MemoryScape. Enter them in the same way as on the command line.

If you enter or change an argument after your program begins executing, your changes take effect only when you restart your program.

- Any new or changed environment variables. Enter them one to a line in the following format: *variable-name=value*.

Again, your changes take effect only when you restart your program.

When a program executes upon a remote host, MemoryScape launches a helper program named **tvdsvr** on that machine. This small program interacts with your program and MemoryScape. Altogether, you'll now have three running programs:

- MemoryScape on your local computer
- The program from which you will be obtaining memory information and which is running on the remote host

-
- **tvdsrv**, also running on the remote host

Attaching to Programs and Adding Core Files

If you wish to attach to a program that is already running or a core file, click either the **Attach to running program** or **Add core file** link on the **Add Programs to Your MemoryScape Session** page.

Within the **Attach to a Running Program** screen, select the processes you would like to attach to, then click the **Next** button. Adding a core file is done in exactly the same way as a regular program except that you need to name the location of the core file and the executable.

NOTE >> MemoryScape requires that all programs use the MemoryScape agent. In most cases, it does this behind the scenes for you before the program begins executing. It cannot do this for an already executing program or for a core file. So, just attaching to an already running program will not provide the information you need as the agent isn't being used with your program. In some cases, you may want to add it by starting the program using the shell `env` command. However, the best alternative is to link the MemoryScape agent. For details, see [“Linking Your Application with the Agent”](#) on page 136.

Stopping Before Finishing Execution

Immediately before your program finishes executing, MemoryScape halts the program. This lets you examine memory state information at that time.

Stopping program execution when the program stops is optional. You can use a MemoryScape option that lets your program finish executing. If you do this, however, MemoryScape discards the state information associated with the program. (For information on changing this option, see [Task 3: “Setting MemoryScape Options”](#) on page 71.)

MemoryScape allows you to halt your program's execution at any time. However, it does not allow you to select the exact code location for your program to stop, and the program may stop inside a `malloc` or `new` call. If this happens, you may see an odd corrupt guard block or leak in your reports. When your process resumes execution, it will clear up the odd result.

If you require fine program control using breakpoints, or you need thread control, you will need to use TotalView with MemoryScape. See [Debugging with TotalView](#) in the [Locating Memory Problems](#) section of this documentation.

Exporting Memory Data

The information MemoryScape displays is created by analyzing information it stores while your program executes and it reflects the current state. In many cases, you will also want to compare this current state against an older state. To do this, you will need to use the **Export Memory Data** command found on the left side of many screens to write memory information to disk. At a later time, you can read this exported data back into MemoryScape and then either compare it to the current state or explore its information as if it were the current state.

NOTE >> We recommend that you export information frequently. It is better to never use this information instead of wishing that you had taken the time to export it.

MemoryScape Information

Most of the information you will use is contained within the **Memory Reports** tab. Here is an overview of the kinds of reports that you can receive.

Figure 36: Report Tabs



- **Leak Detection.** The Backtrace and Source reports in this section are identical in format to those shown in the Heap Status reports. They differ in that they only show leaked memory. You can group these leaks in different ways. One of the most useful is to isolate the largest leaks and then search for solutions to these problems first. For information on these reports, see [Task 9: “Seeing Leaks”](#) on page 102.

These reports, as well as the Heap Status reports, can contain considerable information. You can exclude information from Source and Backtrace reports by filtering it. See [Task 10: “Filtering Reports”](#) on page 103 for more information.

- **Heap Status.** The reports in this section give you information on all of your program’s allocations and deallocations. In particular, MemoryScape also groups allocations by the place where the allocation occurred. This information includes the backtrace—which is your program’s call stack when the allocation occurred—and the source line that allocated the memory.

The Backtrace and Source reports present this information in tabular form. The **Heap Status Graphical Report** is an easy way to browse through the program’s allocated blocks. Also, placing your mouse over a block gives you information about that block. (See [Figure 61](#) on page 95.)

For information on these Heap Status reports, see [Task 7: “Graphically Viewing the Heap”](#) on page 94 and [Task 8: “Obtaining Detailed Heap Information”](#) on page 97.

- **Corrupted Memory.** MemoryScape can analyze your program’s memory blocks to see if the program wrote past a block’s boundaries. If you had set the enable guard blocks option, selecting this report shows corrupted blocks.

For information, see [Task 11: “Viewing Corrupted Memory”](#) on page 108.

- **Memory Comparisons.** MemoryScape lets you save memory state information and read it back in. After it is read back in, you can compare it against the current memory state. Or, you can read in an another save memory state and compare the states against one another.

After MemoryScape reads in saved information, you can obtain reports on the information in exactly the same way as you obtain reports for an executing program.

For information, see [Task 13: “Comparing Memory”](#) on page 113.

- **Memory Usage.** MemoryScape can display charts of how much memory is being used and how this memory is allocated in sections of your program. It can do this for one or more of your program’s processes.

For information, see [Task 5: “Seeing Memory Usage”](#) on page 86.

Where to Go Next

[Task 3: “Setting MemoryScape Options”](#) on page 71

Describes how to change the MemoryScape default settings or to add additional capabilities.

Task 4: “Controlling Program Execution” on page 82

Shows how to start, stop, and restart your program in addition to several other operations.

Display MemoryScape Reports

Most of the tasks that you’ll perform with MemoryScape involve creating reports. The introduction to this chapter (“[Memory Tasks](#)” on page 60) contains a list of these tasks.

Task 2: Adding Parallel Programs

Everything discussed in [Task 1: “Getting Started”](#) on page 61 also applies to parallel programs, but additional information is required before you can collect memory information on programs running in parallel.

First, select **Home | Add Program**. (This is also the screen displayed if you do not name a program when starting MemoryScape.) Next, select **Add parallel program** to launch the Add New Program dialog, [Figure 37](#).

Figure 37: Add New Program (Parallel) Screen

The information entered here could also be entered directly on a command line, with no difference in the way your program behaves. For detailed information, see [“Setting Up MPI Debugging Sessions”](#) on page 127.

The information that is unique to parallel programs is as follows:

MPI Type

Select one of the MPI systems in this list.

Tasks

Enter the number of processes that your program should create. This is equivalent to the **-np** argument used by most MPI systems.

Nodes

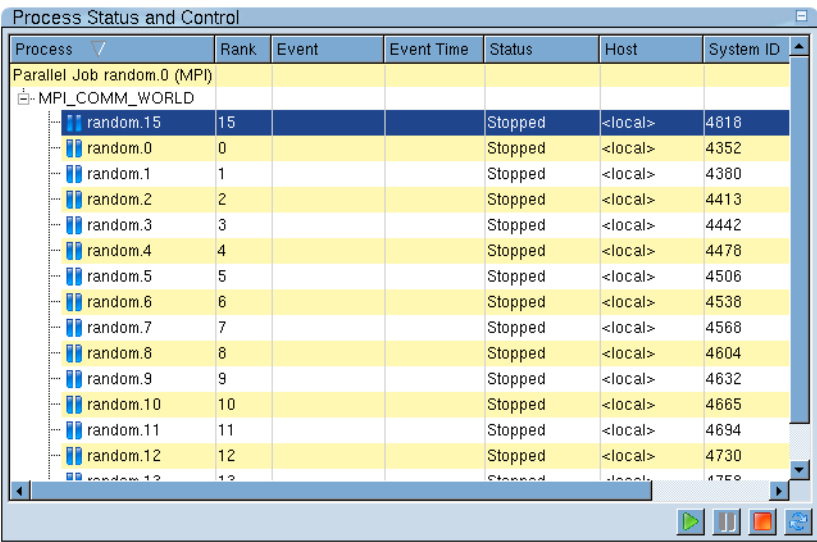
Some MPI systems let you specify the number of nodes upon which your tasks will execute. For example, suppose your program will use 16 tasks. If you specify four nodes, four tasks would execute on each node.

MPI launcher arguments

If you need to send command-line information to your MPI system, enter them in this area.

After selecting memory debugging options (the next task) and starting execution, MemoryScape begins capturing information from each executing task. The sole difference between using MemoryScape on a parallel program versus a non-parallel program is that you have many processes to examine instead of just one. [Figure 38](#) shows the Process Status and Control area for a 32-process MPI job.

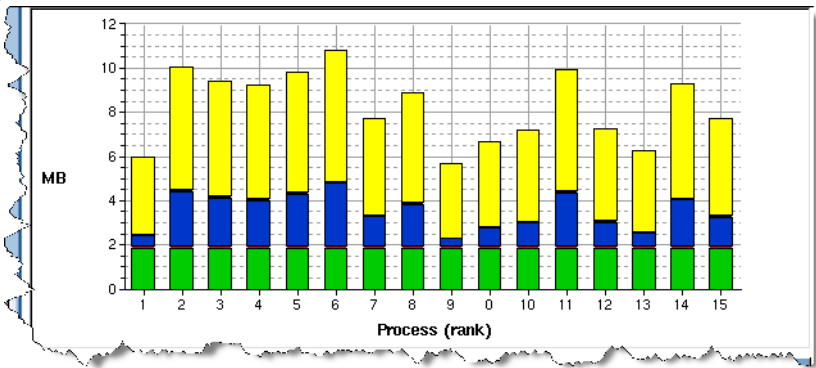
Figure 38: Process Status and Control Area



Process	Rank	Event	Event Time	Status	Host	System ID
Parallel Job random.0 (MPI)						
MPL_COMM_WORLD						
random.15	15			Stopped	<local>	4818
random.0	0			Stopped	<local>	4352
random.1	1			Stopped	<local>	4380
random.2	2			Stopped	<local>	4413
random.3	3			Stopped	<local>	4442
random.4	4			Stopped	<local>	4478
random.5	5			Stopped	<local>	4506
random.6	6			Stopped	<local>	4538
random.7	7			Stopped	<local>	4568
random.8	8			Stopped	<local>	4604
random.9	9			Stopped	<local>	4632
random.10	10			Stopped	<local>	4665
random.11	11			Stopped	<local>	4694
random.12	12			Stopped	<local>	4730
random.13	13			Stopped	<local>	4758

MemoryScape will be collecting data on each of the 32 processes, allowing you to examine memory information for each. Examining each, however, is seldom productive. Instead, you need to focus in on where problems may occur. The Memory Usage charts are often the best place to start. Figure 39 shows part of a stacked bar chart.

Figure 39: Process Status and Control Area, Stacked Bar Chart



Given this graph, you probably want to start with a look at process 6, as it is using the most memory. You might also want to use memory comparison features to compare process 6 to process 9, the process using the least memory.

If possible, run your program a few times, stopping it periodically to get an idea of how it uses memory so you can identify any patterns. Another time, you may want to stop the program periodically when you see memory use changing and then export memory data. In this way, you can perform detailed analyses later. This is particularly important in situations where access to HPC machines is limited.

NOTE >>For remote processes, the only way to reliably configure MemoryScape is to set the TVHEAP_ARGS variable. For information, see “TVHEAP_ARGS” on page 42.

Task 3: Setting MemoryScape Options

This task discusses options that control MemoryScape activities. MemoryScape default options are probably right for most memory debugging tasks, but in some cases, you may need additional activities. These are set using the advanced **Memory Debugging Options** screen.

Some actions must be set before program execution begins. Others can be set anytime. Also, while you normally enable and disable activities by selecting basic options, you can also enable them from the advanced options page.

Before reading this task, you should be familiar with the following information:

“Locating Memory Problems”

Contains an overview of memory concepts and MemoryScape.

Task 1: “Getting Started” on page 61

Tells you how to start MemoryScape. It also contains an overview of the kinds of information you can obtain.

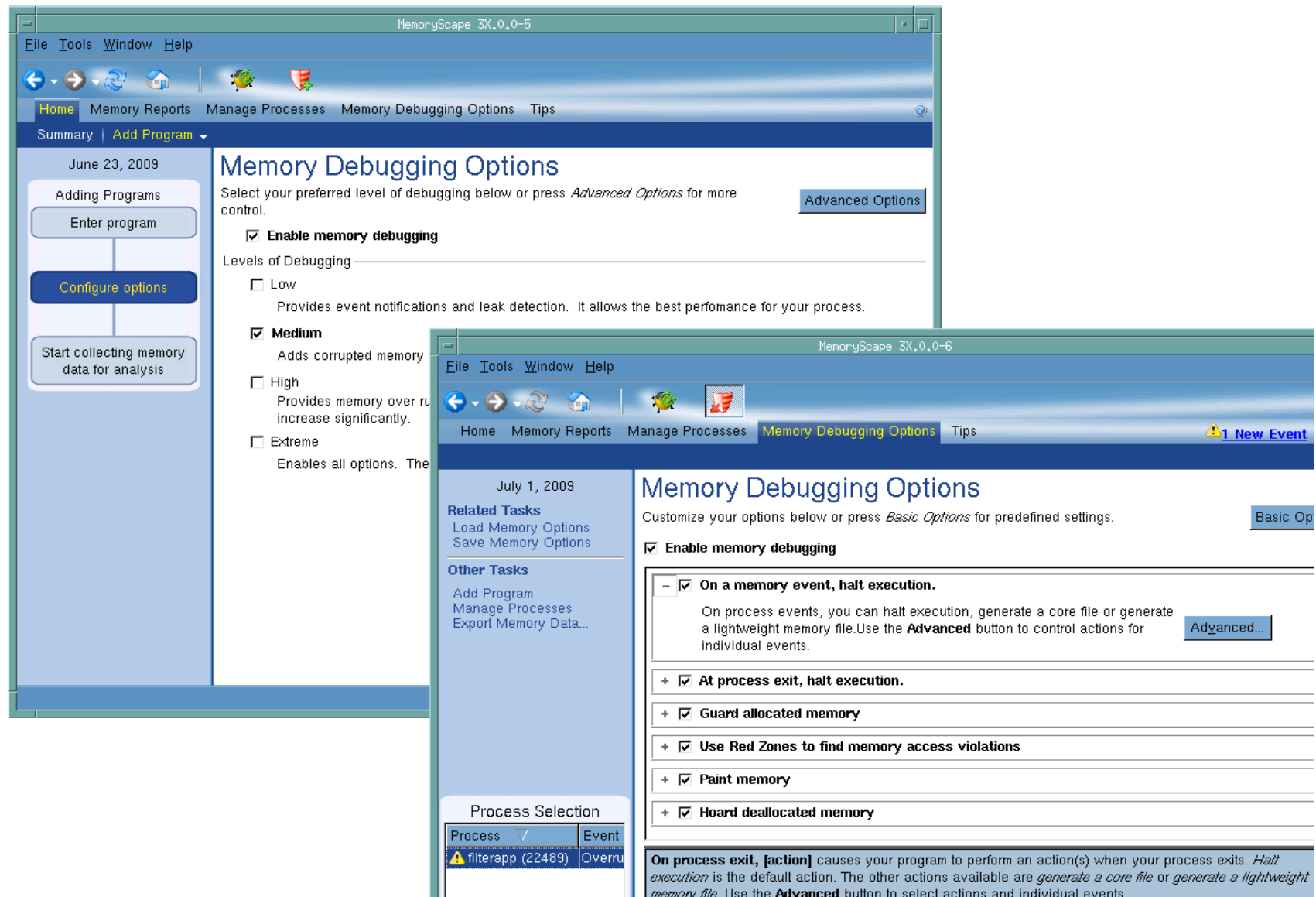
Topics within this task are:

- “Basic Options” on page 71
- “Advanced Options” on page 74
- “Where to Go Next” on page 81

Basic Options

MemoryScape automatically displays the **Memory Debugging Options** screen after you add a program. You can also display it by selecting **Memory Debugging Options** on the primary navigation bar, [Figure 40](#) on page 73.

Figure 40: Memory Debugging Options



These options control the level of debugging activity: **Low, Medium, High, or Extreme.**

Low

Records all memory requests, including calls to **malloc()**, **realloc()**, and other calls to your malloc library. It also includes calls to C++ operators such as **new** and **delete** that indirectly use this library. It can even include memory management functions performed by some Fortran libraries.

When you ask for a report, MemoryScape analyzes this recorded information and displays the information you request.

This selection also sets automatic event notification.

In most cases, **Low** is all you'll ever need.

Medium

In addition to performing all operations indicated by **Low**, MemoryScape writes guard blocks before and after allocated memory blocks. For more information on guard blocks and locating corrupted memory, see [Task 11: "Viewing Corrupted Memory"](#) on page 108.

This setting increases the size of the allocated memory block, but the extra overhead is small.

Select **Medium** only when you need to check for corrupted memory.

High

In addition to performing all operations indicated by the **Low** level, MemoryScape writes a Red Zone after the allocated memory block and notifies you if you access memory in this Red Zone. This is called an overrun.

In this case, the additional allocated memory can result in significant extra overhead.

Select **High** only if you need to check for memory overruns. As an alternative, select Low and use the Red Zones button on the task bar to turn on Red Zones as needed for suspect allocations.

To find underruns in your allocated memory, see ["Use Red Zones to find memory access violations"](#) on page 78.

Extreme

In addition to performing all operations indicated by the **Low, Medium, and High** levels, MemoryScape paints allocated and deallocated memory as well as hoard memory.

These activities both decrease performance and use more memory. For information on when to use these features, see [Task 15: "Hoarding Deallocated Memory"](#) on page 118 and [Task 16: "Painting Memory"](#) on page 119.

Select **Extreme** only if you need hoarding and painting.

NOTE >> Red Zone controls appear on the toolbar only on platforms that support Red Zone capabilities. See the Platform Guide for specific platform support. This applies to the High and Extreme levels only.

For a combination of options unavailable on this screen, see ["Advanced Options"](#) on page 74, which allow you to select options a la carte, as well as change default settings.

Advanced Options

If you need to fine tune MemoryScape behavior, use the Advanced Options display on the **Memory Debugging Options** screen.

Select the **Advanced Options** button to display advanced options, [Figure 40](#) on page 73.

NOTE >>The only way to reliably place MemoryScape configuration information into remote processes is to set the TVHEAP_ARGS variable. For information, see [“TVHEAP_ARGS”](#) on page 42.

Use the **Basic Options** button to return to the original display.

This screen has six sets of controls. In this figure, all six are selected. The initial options set here are pre-determined by the **Low, Medium, High, and Extreme** selections in the basic screen.

NOTE >>Notice the process controls on the left of this screen. Select one or more processes to limit the activities being performed.

Activities you can modify, enable, or disable:

- [“Halt execution at process exit \(standalone MemoryScape only\)”](#) on page 75
- [“Halt execution on memory event or error”](#) on page 75
- [“Guard allocated memory”](#) on page 77
- [“Use Red Zones to find memory access violations”](#) on page 78
- [“Paint memory”](#) on page 80
- [“Hoard deallocated memory”](#) on page 80

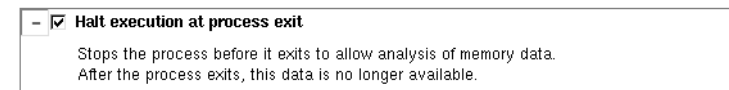
Halt execution at process exit (standalone MemoryScape only)

This option is visible only when running MemoryScape standalone without TotalView.

When selected ([Figure 41](#)), MemoryScape halts your program’s execution just before it stops executing. (In most cases, this is immediately before your program executes its **_exit** routine.) At this time, you can analyze the memory data that MemoryScape has collected.

If this option isn’t set, your program executes the **_exit** routine and MemoryScape discards the data it has collected for that process. This means that MemoryScape cannot analyze your program’s memory use, so we recommend that you leave this set.

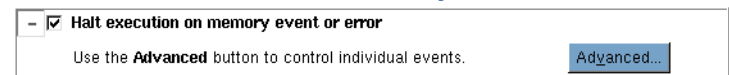
Figure 41: Halt execution at process exit Options



Halt execution on memory event or error

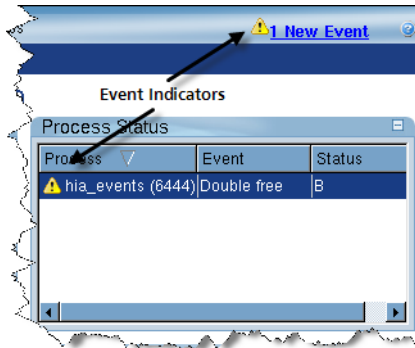
When selected, MemoryScape halts your program’s execution just before it detects that a problem will occur when your program calls a function in the malloc library. We recommend that you leave this set

Figure 42: Halt execution on memory event or error



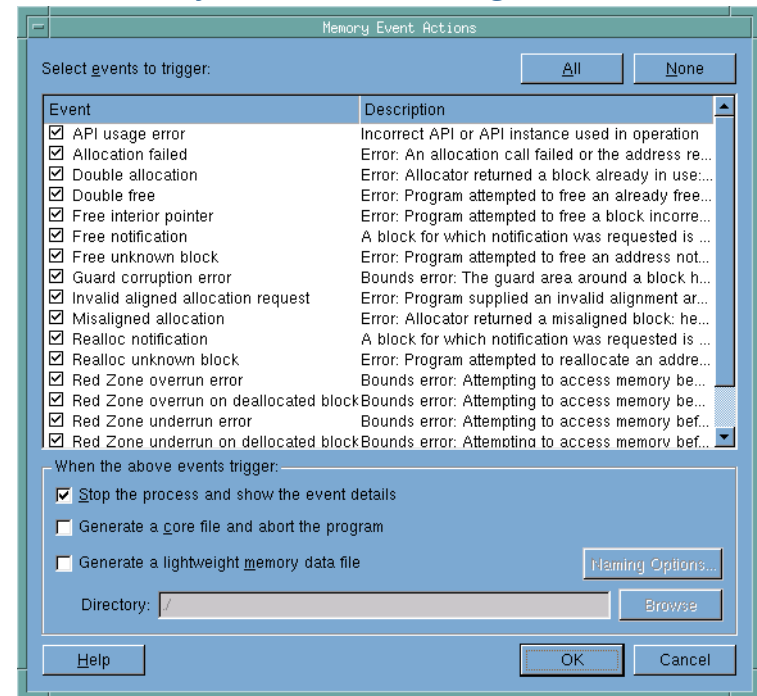
When your program allocates memory, MemoryScape records this and other information in an internal table. Every time your program deallocates or reallocates memory, it checks what is about to be done against this information. If it detects that a problem will occur, it stops execution and displays an event indicator, shown in Figure 43.

Figure 43: Event Indicators



MemoryScape can watch for a number of events. Its default is to watch for all, but you can specify specific events using the **Advanced** button which launches the Memory Event Actions Dialog Box, Figure 44.

Figure 44: Memory Event Actions Dialog Box



NOTE >> This snapshot was created using TotalView Debugger Team and Team Plus. If you are not licensed for these products, then the two options Generate a core file and abort the program and Generate a lightweight memory data file are unavailable.

1. Select events to trigger: Select or unselect events. In most cases, you should unselect events coming from a library or system you can't control.

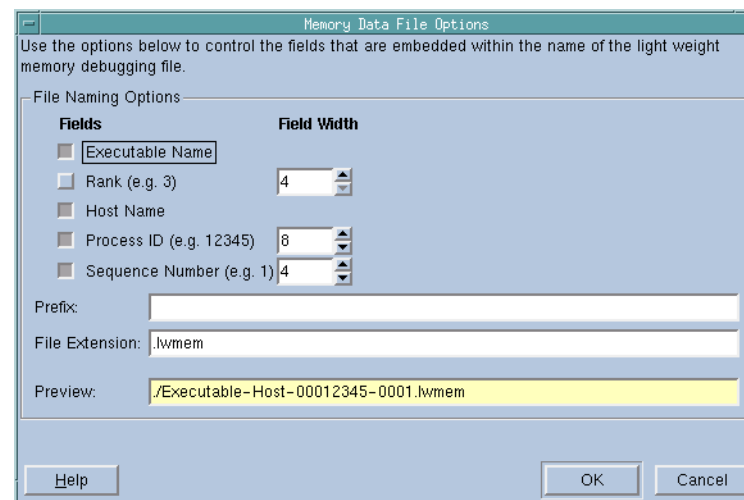
2. When the above events trigger: Select the action to perform, as follows:

- **Stop the process and show the event details:** When selected, additional options are available to generate a core file or lightweight memory file.
- **Generating a core file:** MemoryScape writes the file to disk and aborts execution. (The operating system routines that generate a core file cause the program to be aborted.) As you are still within MemoryScape, you can restart your program.
- **Generate a lightweight memory file:** MemoryScape creates a file similar to that written when you use the **File > Export** command. This file can then be read back into MemoryScape in the same way as exported **.dbg** files. In contrast to a core file, your program continues to execute after MemoryScape writes the file.

You can name the directory into which the MemoryScape writes the file by identifying the location in the **Directory** field or use the Browse button to navigate to a directory.

You can control this file's name using the **Naming Options** button which launches the Memory Data File Options dialog, [Figure 45](#).

Figure 45: Memory Data File Options



If you change the prefix or file extension, the change is reflected in the **Preview** area.

Guard allocated memory

When your program allocates a memory block, MemoryScape can write additional memory segments both before and after the block. These segments are called *guards*. Immediately after creating the guards, MemoryScape initializes them to a bit pattern.

These guards can help you identify corrupted memory in two ways:

- When your program deallocates a memory block, MemoryScape checks to determine if the pattern differs from when it first wrote the block. If so, your program will have overwritten memory that it shouldn't have used.

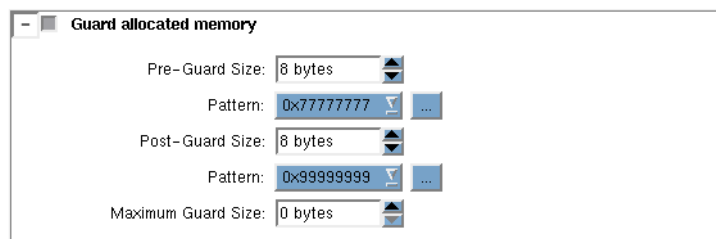
This indicates that memory was corrupted. (MemoryScope writes one pattern into the guard preceding a block and another into the one after it.)

- Whenever you halt execution, you can ask MemoryScope to check for corrupted guard blocks, which looks at all guard blocks surrounding your allocations.

For information on guard blocks, see [Task 11: “Viewing Corrupted Memory”](#) on page 108.

Enable guard blocks by selecting either a level of **Medium** or selecting the check box in the **Advanced** screen. [Figure 46](#) shows the portion of that screen that controls guard blocks.

Figure 46: Guard allocated memory Option



These options control the size of guard blocks and their pattern.

You can also set a maximum size for the guard blocks that surround a memory allocation. This can be useful because the size actually used for a guard block can be greater than the pre-guard and post-guard sizes due to the way an operating system aligns information. If memory is tight, setting a value here ensures that blocks do not use an excessive amount of memory.

If the value is set to zero (0), which is the default, MemoryScope does not set a maximum size.

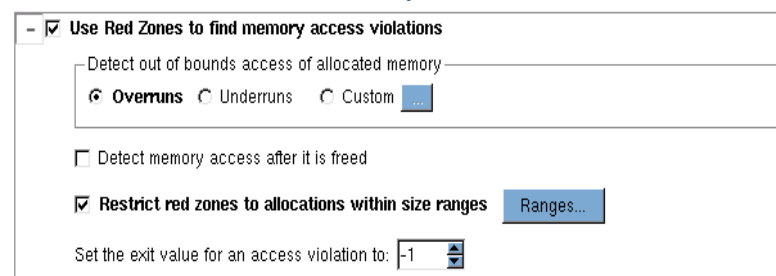
Use Red Zones to find memory access violations

When your program allocates a memory block, MemoryScope can write an additional memory buffer either before or after the block. These buffers are called Red Zones. MemoryScope watches the Red Zone for illegal read or write access.

NOTE >> Red Zone controls appear on the toolbar only on platforms that support Red Zone capabilities. See the Platform Guide for specific platform support.

To enable Red Zones, set the debugging level to **High** (in the Basic Options) or select the option **“Use Red Zones to find memory access violations”** on the Advanced Options screen, [Figure 47](#).

Figure 47: Advanced Red Zone Option




This dialog controls the type of error to detect and defines ranges and values for some of the detection types. In addition to detecting overruns and underruns (discussed above), you can customize Red Zone definitions, discussed in [“Customizing Red Zones”](#) on page 79.

Red Zones can help identify where an illegal access occurs in these cases:

- Overruns: reads or writes past the bounds of the allocated block, detected by placing a Red Zone after the block.

- Underruns: reads or writes before the bounds of the allocated block, detected by placing a Red Zone before the block.
- Read / write access after a block is deallocated.

If you select **“Detect memory access after it has been freed”**, you are notified when this occurs. MemoryScape retains the deallocated blocks to monitor them for both read and write access. This option will increase memory consumption in your program.

One way to limit the overhead incurred in using Red Zones is to restrict their use to blocks of specified sizes. (See **“Restricting Red Zones”** on page 79 for details on using the option **“Restrict Red Zones to allocations within size ranges.”**) Another way is to turn Red Zones on and off during program execution using the Red Zones button  on the toolbar. Because of the increased memory consumption associated with Red Zones, use them with caution.

To know when your program exits due to an access violation, use the last option, **“Set the exit value for an access violation.”**

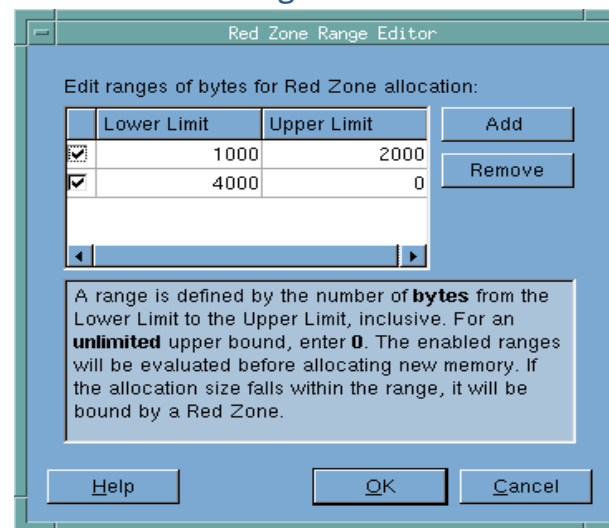
NOTE >>For a detailed description of how Red Zones work, see **“Red Zones Bounds Checking: dheap -red_zones”** on page 38.

Restricting Red Zones

You can restrict Red Zones to apply to blocks of defined sizes. For instance, if you suspect that your program is overwriting the bounds of a large array or structure, you can specify a range that includes the size of the array or structure, **Figure 48**. (Note that specifying 0 as the upper limit means the upper bound is unlimited.) For example, if you have an array that is 1500 bytes, you

can define the range to be from 1000 to 2000 bytes, and Red Zones are applied only to allocations within this range. This limits the additional overhead in memory consumption and targets your array or structure.

Figure 48: Red Zone Size Ranges



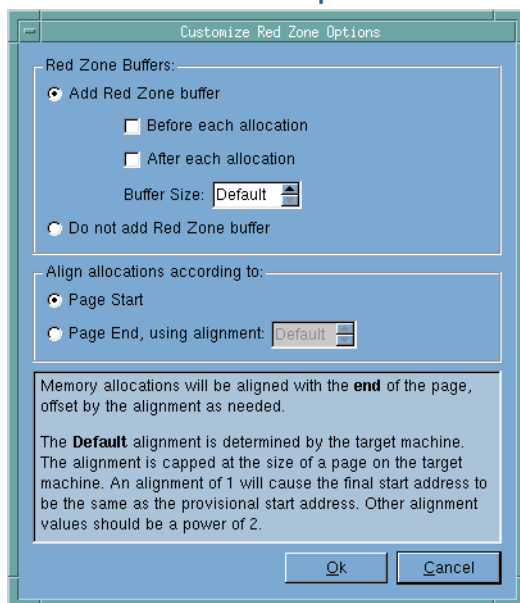
Customizing Red Zones

You can customize Red Zones in several ways using the screen in **Figure 49**.

- Buffer placement: the Red Zone buffer may be placed before, after, or *both* before and after the allocated block. Use caution when adding Red Zones both before and after your allocations, as this doubles the Red Zone overhead.
- Buffer size: the default is the target machine page size, but you can modify this. The number of bytes entered is rounded to the nearest alignment specified by the machine.

- You can elect not to use buffers. This setting detects whether a block is used after having been freed, with much lower memory consumption.
- Allocation alignment according to page start: this setting is useful for catching underruns.
- Allocation alignment according to page end: the default alignment for this setting is determined by the target machine and capped at the machine's page size. An alignment of 1 results in a final start address the same as the provisional start address. Other alignment values should be to a power of 2.

Figure 49: Custom Red Zone Options



Paint memory

When your program reads a data value, you are assuming that the program has already set the memory to some value. If your program hasn't seen a value or if it tries to read data from the block after you've deallocated it, an error has occurred. MemoryScape can help you detect these problems by setting the value of allocated or deallocated blocks to a value. When your program reads these painted values, it may be able to detect that there is a problem. In some cases, your program may even stop executing.

Enable painting either by selecting a level of **Extreme** in the Basic Options, or selecting the check box in the **Advanced** screen. [Figure 50](#) shows painting controls

Figure 50: Paint memory Option



These controls let you separately enable allocations and deallocations (you might want to use one and not the other if memory is tight) and set the pattern that MemoryScape writes. Notice that the default patterns resemble the words “allocate” and “deallocate”.

For information on painting, see [Task 16: “Painting Memory”](#) on page 119.

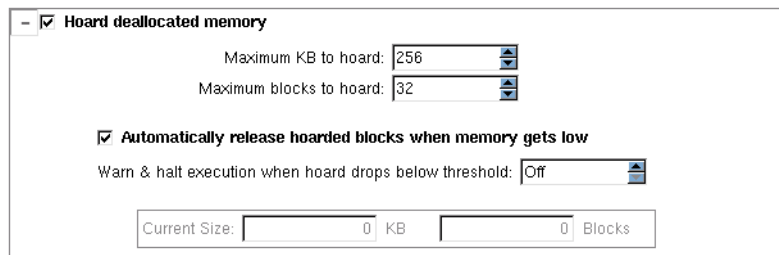
Hoard deallocated memory

After a program deallocates memory, it is quite possible that a pointer still points into the deallocated block. If the program uses that pointer to access information in this block, this data can be corrupt. Having MemoryScape hold on to deallocated blocks for awhile helps to keep this data correct. Holding on to deallocated blocks is called *hoarding*. By retaining this memory,

you reduce program failures. More importantly, because the program continues to execute, additional and often related memory errors occur—errors that can help you diagnose the problem.

Enable hoarding either by selecting a level of **Extreme** in the Basic Options, or by selecting the check box in the **Advanced** screen. [Figure 51](#) shows the portion of that screen that controls hoarding.

Figure 51: Hoard deallocated memory Option



- ☒ **Hoard deallocated memory**

Maximum KB to hoard: 256

Maximum blocks to hoard: 32

☒ **Automatically release hoarded blocks when memory gets low**

Warn & halt execution when hoard drops below threshold: Off

Current Size: 0 KB 0 Blocks

This dialog displays the hoard size and the number of blocks to be hoarded, which you can customize.

Hoarding deallocated memory may increase the risk of running out of available memory sooner than expected because deallocated memory is not released back to the heap manager. Reduce this risk by automatically releasing the hoarded memory when available memory gets low. You can also receive an event alerting you when the hoard size drops below the defined threshold. At that point, you know your program is getting close to running out of memory.

For more information, see [“Hoarding”](#) on page 56 and [Task 15: “Hoarding Deallocated Memory”](#) on page 118.

Where to Go Next

Now that MemoryScape is set up, you are ready to start your program executing under MemoryScape control. For more information, see [Task 4: “Controlling Program Execution”](#) on page 82.

Task 4: Controlling Program Execution

This task discusses how to control a program's execution from within MemoryScape. Once you've added programs to MemoryScape, you are ready to execute your program(s). During execution, MemoryScape collects memory information. For long-running programs, you will want to start execution, stop it, look at memory information, and then continue execution.

MemoryScape allows you to halt your program's execution at any time. However, it does not allow you to select the exact code location for your program to stop, and the program may stop inside a malloc or new call. If this happens, you may see an odd corrupt guard block or leak in your reports. When your process resumes execution, it will clear up the odd result.

If you require fine program control using breakpoints, or you need thread control, use TotalView with MemoryScape. See ["Debugging with TotalView"](#) on page 58.

Before reading this task, you should be familiar with the following:

[Locating Memory Problems,](#) on page 1

An overview of memory concepts and MemoryScape.

[Task 1: "Getting Started"](#) on page 61

How to start MemoryScape and an overview of the kinds of information you can obtain.

[Task 3: "Setting MemoryScape Options"](#) on page 71

How to configure MemoryScape so that it performs the activities you want it to perform.

The controls for starting and stopping program execution are on the **Home | Summary** screen as well as the lower left corner of many screens. Additional controls are in the **Manage Process and Files** screen, [Figure 52](#) on page 84.

Figure 52: Execution Controls

The screenshot displays the MemoryScope 3X.0.0-5 application interface, which is divided into several panels for memory debugging and process management.

Memory Debugging Session Panel:

- Summary:** June 23, 2009
- Add Programs:**
 - Add new program
 - Add parallel program
 - Attach to running program
 - Add core file
 - Add memory debugging file
- Generate Reports:**
 - Heap Status Reports
 - Leak Detection Reports
 - Memory Usage Reports
 - Corrupted Memory Report
 - Compare Memory Usage
- Other Tasks:** Export Memory Data
- Memory Debugging Guide:**
 - Analyze your memory debugging data: Memory debugging data is available for analysis. Select a report to view memory debugging data.
 - [Leak detection source report](#)
 - [Heap graphical report](#)
 - [Heap status source report](#)
 - [Memory debugging reports summary](#)
- Process Status and Control:**

Process	Rank	Event
Parallel Job random.0 (MPI)		
[-] MPI_COMM_WORLD		
[-] random.15	15	
- Heap Usage Monitor:** A line graph showing memory usage in MB over time. The y-axis ranges from 0.85 to 3.85 MB. The x-axis shows time points from 3.4 to 6.4. The graph displays a fluctuating line with green and blue markers.

Manage Processes and Files Panel:

- Summary:** June 23, 2009
- Other Tasks:** Add Program, Export Memory Data
- Manage Processes and Files:** You can control your processes or unload files using these controls.
- Processes:** Select processes from the list below and use these controls to change their execution state.

Process	Event	Event Time	Status	Host	System ID
Parallel Job fork_loopLinux (Fork/Exec)					
[-] fork_loopLinux			Running	<local>	14637
[-] fork_loopLinux.1			Running	<local>	14645
[-] fork_loopLinux.1.1			Running	<local>	14647
[-] fork_loopLinux.1.1.1			Running	<local>	14649
[-] fork_loopLinux.1.2			Running	<local>	14648

Buttons: Run, Halt, Kill, Restart, Detach
- Memory Debugging Files and Core Files:** Select files from the list below and press Unload to remove the file from this session.

Process	Event	Event Time	Status	Host	System ID
[-] random.3 (File: random-3.mdbg)			-	<local>	-

Buttons: Unload

Topics in this task are:

- [“Controlling Program Execution from the Home | Summary Screen”](#) on page 85
- [“Controlling Program Execution from the Manage Processes Screen”](#) on page 85
- [“Controlling Program Execution from a Context Menu”](#) on page 85
- [“Where to Go Next”](#) on page 85

Controlling Program Execution from the Home | Summary Screen

The controls on the **Home | Summary** screen start and halt execution of all your processes ([Figure 52](#) on page 84). For additional controls or to control processes separately, use the **Manage Processes | Manage Processes and Files** screen.

MemoryScape cannot generate information while a process is running, so it automatically halts the program for you. You are later asked if you want to resume execution.

While your program executes, MemoryScape graphically displays the memory usage, reflecting a pattern of memory use. If you see that the graph's shape is different from one run to another, you may learn when to halt the program and inspect memory. Or, if memory use grows sharply and unexpectedly, you may want to stop the program and see why.

Controlling Program Execution from the Manage Processes Screen

The controls on the **Manage Process and Files** screen include the ability to run, halt, kill, and restart execution. In addition, you can detach a program from MemoryScape, which means remove it from MemoryScape.

You can control which processes this command affects by selecting the processes from the list on this screen.

Controlling Program Execution from a Context Menu

When you right-click on a process in the bottom left part of most screens, MemoryScape displays a context menu that also includes execution controls. These controls operate in the same way as those in other screens.

Where to Go Next

- If MemoryScape detects a problem while your program is executing, it can stop execution, letting you know the kind of problem that just occurred. For more information, see [Task 6: “Using Runtime Events”](#) on page 90.
- You can display charts of your memory use. For more information, see [Task 5: “Seeing Memory Usage”](#) on page 86.
- After you stop execution, you will want to obtain reports on memory activity. You will find a list of these tasks in the introduction to this chapter ([“Memory Tasks”](#) on page 60).

Task 5: Seeing Memory Usage

This task generates charts that visually display memory usage information, along with detailed tables that numerically describe this information.

Before reading this task, you should be familiar with the following information:

[Locating Memory Problems](#),” on page 1

An overview of memory concepts and MemoryScape.

[Task 1: “Getting Started”](#) on page 61

How to start MemoryScape and an overview of the kinds of information you can obtain.

[Task 3: “Setting MemoryScape Options”](#) on page 71

Describes how to configure MemoryScape so that it performs the activities you want it to perform.

The totals shown in the Memory Usage reports may differ slightly from those in the Heap Status reports, because heap status is generated from monitoring program requests for memory (**malloc** or **new**) and program release of memory (**free** or **delete**), while memory usage data is obtained from the operating system facilities. Depending on the operating system, memory usage totals may include anonymous memory regions that the program or one of its libraries may have mapped into its address space. The totals also include a small amount of overhead from MemoryScape itself.

The Memory Usage report is intended to be a quick check of your program’s memory usage from the system’s perspective. For detailed information on your program’s use of the heap, see the Heap Status reports.

To display memory usage data, select **Memory | Memory Usage**. Select these reports:

- High-level process report
- Detailed program and library report
- Chart Report

Topics in this task are:

- [“Information Types”](#) on page 86
- [“Process and Library Reports”](#) on page 87
- [“Chart Report”](#) on page 87
- [“Where to Go Next”](#) on page 89

Information Types

While all information can be useful, the data in the heap column is the most interesting as it contains information that you can control. Memory Usage reports display the amount of memory:

Text Your program uses to store your program’s machine code instructions.

Data Your program uses to store uninitialized and initialized data.

Heap Your program is currently using for data created at runtime.

Stack Used by the currently executing routine and all the routines in its backtrace.

If you are looking at a multithreaded process, MemoryScape only shows information for the main thread’s stack.

The stack size of some threads does not change over time on some architectures. On some systems, the space allocated for a thread is considered part of the heap.

Stack Virtual Memory

The logical size of the stack. This value is the difference between the current value of the stack pointer and the value reported in the **Stack** column. Also, this value can differ from the size of the virtual memory mapping in which the stack resides.

Total Virtual Memory

The sum of the sizes of the mappings in the process's address space.

Process and Library Reports

The Process report (the bottom snapshot in [Figure 53](#)) lists all your processes and information about them. To sort, click on a column header.

Figure 53: Memory Usage Library and Process Reports

Process	Text	Data	Heap	Stack	Stack Virtual Mem
Process 35: random.15	1091.50KB	93.33KB	1343.65KB	39.55KB	44.00KB
libc.so.6	1161.38KB	21.63KB			
random	289.89KB	37.95KB			
libtheap.so	119.56KB	6.82KB			
ld-linux.so.2	81.91KB	2.08KB			
libnsl.so.1	69.56KB	10.37KB			
libresolv.so.2	59.56KB	11.53KB			

Process	Text	Data	Heap	Stack	Stack Virtual Mem	Total V
Process 35: random.15	1091.50KB	93.33KB	1343.65KB	39.55KB	44.00KB	
Process 34: random.14	1091.50KB	93.33KB	2.10MB	44.89KB	48.00KB	
Process 33: random.13	1091.50KB	93.33KB	635.65KB	29.75KB	32.00KB	
Process 32: random.12	1091.50KB	93.33KB	1115.65KB	35.10KB	40.00KB	
Process 31: random.11	1091.50KB	93.33KB	2.43MB	48.46KB	52.00KB	
Process 30: random.10	1091.50KB	93.33KB	1033.65KB	34.21KB	40.00KB	
Process 2: random.0	1091.50KB	93.33KB	827.65KB	27.94KB	32.00KB	
Process 29: random.9	1091.50KB	93.33KB	347.65KB	23.52KB	28.00KB	
Process 28: random.8	1091.50KB	93.33KB	1935.65KB	42.22KB	48.00KB	
Process 27: random.7	1091.50KB	93.33KB	1363.65KB	39.55KB	44.00KB	
Process 26: random.6	1091.50KB	93.33KB	2.04MB	40.88KB	52.00KB	

In a similar manner, a **Detailed program and library report** of your memory usage shows this information for your program and all libraries it uses.

Not shown in this figure are the process controls on the left. By selecting (or unselecting) processes and threads, you can control which are shown in these tables.

It can be difficult to know what to do if you identify a problem here. The only component over which you have direct control is the one you've written. Fortunately, this is usually where the problem is. If, however, the problem is with a library that your program loaded, you do have options. If you have control over the library, you can investigate that library's behavior. For example, MemoryScape does identify leaks within libraries.

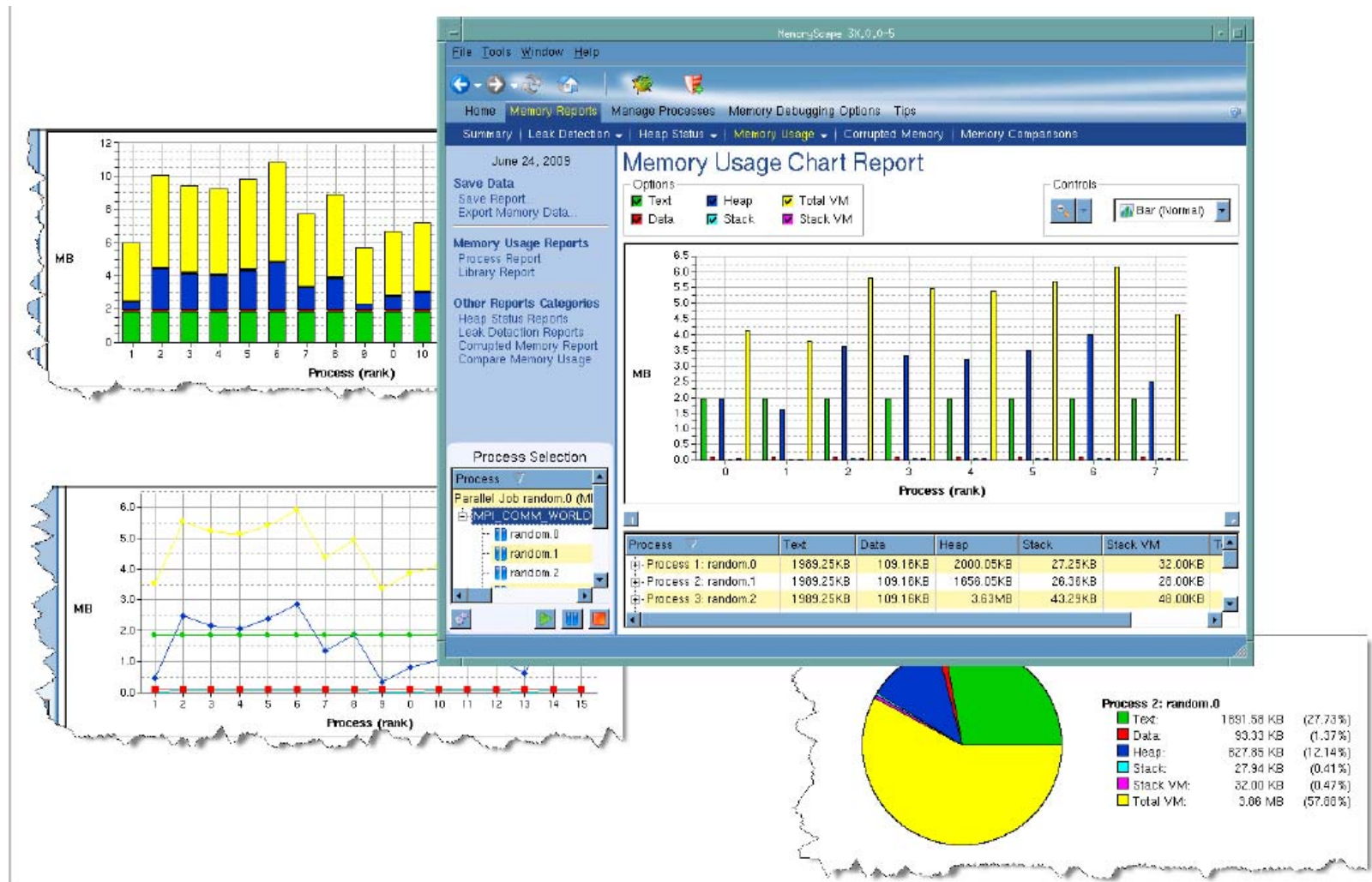
If the problem is one you can't control, your choices are limited. You could attempt to obtain a different version of the library. Or, if you can identify the cause of the problem, write a substitute function.

If, however, the problem is in your code, MemoryScape offers many ways to drill down and obtain information.

Chart Report

While the information tables in the Process and Library reports are useful, the best place to start is the Chart report, which offers a graphic look at your program's memory data, [Figure 54](#) on page 88.

Figure 54: Memory Usage



MemoryScape can display your information using a range of charts: Bar, Stack Bar, Line, and Pie, available from the pulldown list in the **Controls** area. In addition, you can zoom in or out to control the view.

The **Options** area at the top controls which of the six types of memory information are displayed.

The process area below the charts control which processes and threads to chart.

Where to Go Next

- MemoryScape can stop execution when an error occurs or when you want notification for a block being allocated or reallocated. See [Task 6: "Using Runtime Events"](#) on page 90 for more information.
- After you stop execution, you will want to obtain reports on memory activity. You will find a list of these tasks in the introduction to this chapter ("[Memory Tasks](#)" on page 60).

Task 6: Using Runtime Events

This section describes how to tell MemoryScape to stop execution either when an event or an error occurs. Telling you that a problem has occurred is called *notification*. Notification requires that the **Halt execution on memory event or error** option be enabled, which is the default. (This option is located on the advanced display of the **Memory Debugging Options** screen.

In addition, You can tell MemoryScape to notify you when a block is allocated or deallocated by setting a notification within the **Block Properties** window.

Before reading this task, you should be familiar with the following information:

[Locating Memory Problems,](#)” on page 1

Contains an overview of memory concepts and MemoryScape.

Task 1: “Getting Started” on page 61

Tells you how to start MemoryScape. It also contains an overview of the kinds of information you can obtain.

Task 3: “Setting MemoryScape Options” on page 71

Describes how to configure MemoryScape so that it performs the activities you want it to perform.

Topics within this task are:

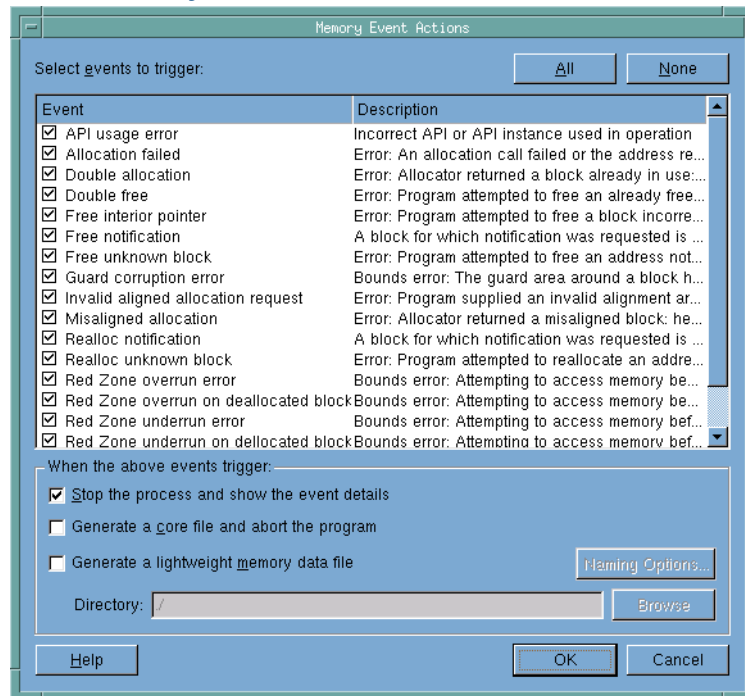
- [“Error Notifications”](#) on page 90
- [“Deallocation and Reuse Notifications”](#) on page 92
- [“Where to Go Next”](#) on page 93

Error Notifications

When your program uses a function from the malloc library, MemoryScape intercepts the call using a process called *interposition* (see [“Behind the Scenes”](#) on page 8). If the action is an allocation, MemoryScape records information about it. If your program is deallocating a block, MemoryScape looks for the block in its table. Based on what it finds, it can stop execution and notify you that a problem is about to occur. For example, if the block was previously deallocated, MemoryScape can stop execution and tell you about the problem.

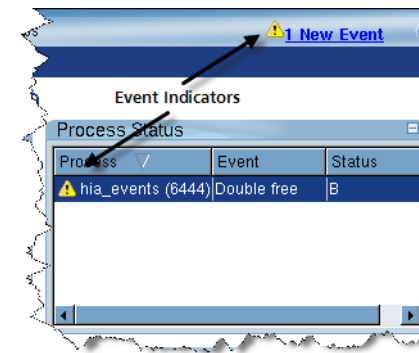
For information on which events stop execution, see the help or examine the contents of the dialog box displayed when you click the **Advanced** button on the **Memory Debugging Options** Advanced screen.

Figure 55: Memory Event Notification



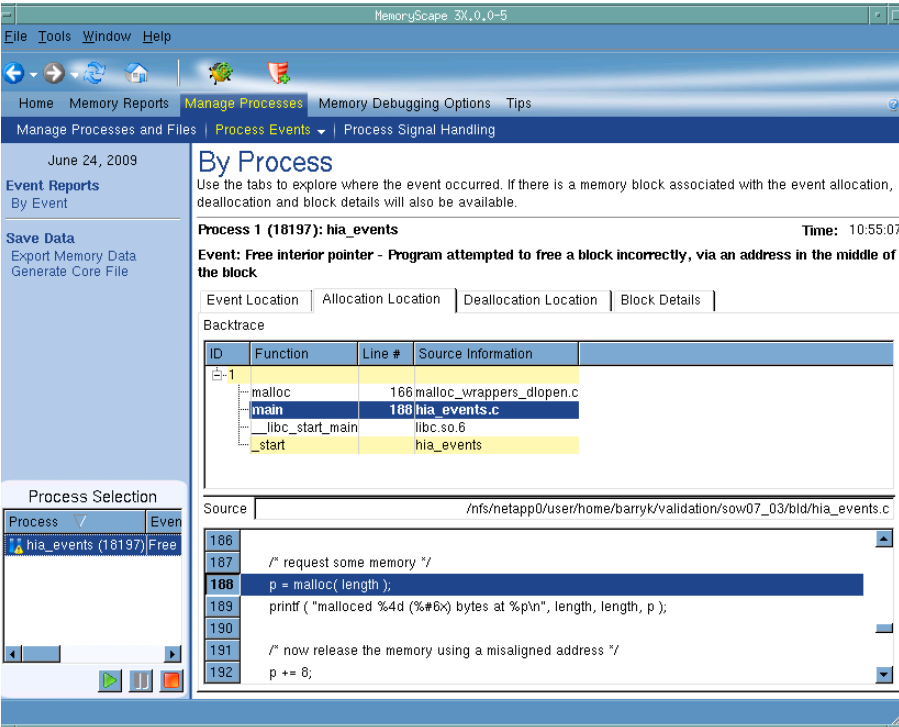
If an event occurs, MemoryScope stops execution and places an event indicator on the screen, [Figure 56](#).

Figure 56: Event Indicator



You can now display the **Manage Processes | Process Event** screen. MemoryScope will take you directly there if you click on the event link, [Figure 57](#).

Figure 57: Process Events Screen



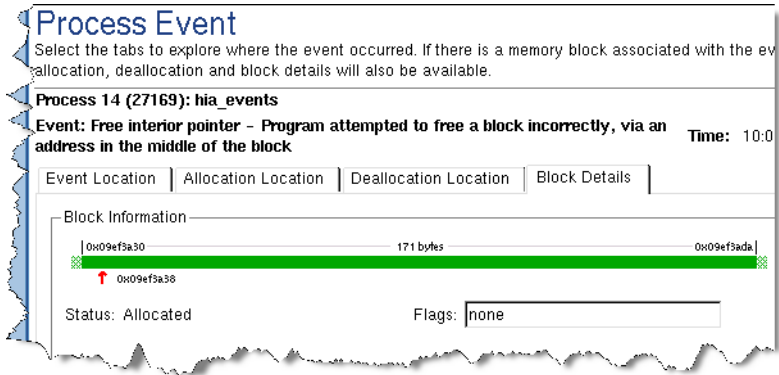
You are now ready to look for detailed information about the notification event. The top of the window (the text in bold) describes the event. Immediately below this text are four tabs: **Event Location**, **Allocation Location**, **Deallocation Location**, and **Block Details**. The information in the first three is of the same kind as that shown in Figure 57.

When MemoryScope intercepts a call to the malloc library, it also records the backtrace associated with the function call (that is, it records the function's call stack). Depending on the event, one or more of these four tabs will have data. For example, if you try to free stack memory, the only location that MemoryScope knows about is the place where the event occurred.

To assist you in locating a problem, MemoryScope saves some of the code surrounding the line in your program that caused the event.

The Block Details tab presents information about this event in a different form. This tab is very useful for obtaining additional information about the block, Figure 58.

Figure 58: Block Details Tab



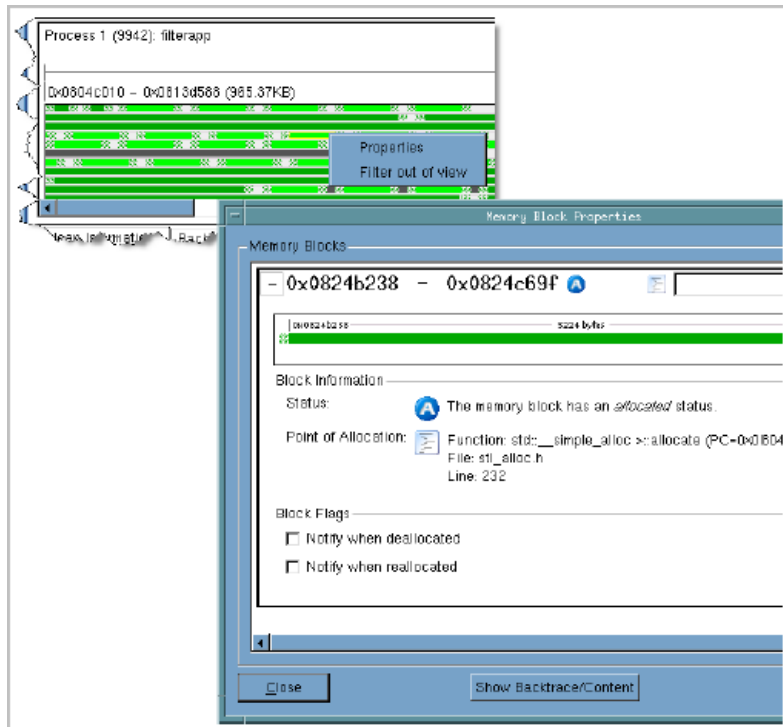
Deallocation and Reuse Notifications

You can configure MemoryScope to notify you when an already allocated block is deallocated or reallocated, like this:

1. Start, then stop execution.

2. Display the **Memory Reports | Heap Status | Heap Status Graphical Report** screen.
3. Locate the block you want to be notified about and right-click on it.
4. In the pop-up, select **Properties**, [Figure 59](#).

Figure 59: Block Notifications



5. Select one or both of the **Notify when deallocated** and **Notify when reallocated** check boxes at the bottom of the window.

If an event occurs while the program is executing (see [Task 4: “Controlling Program Execution”](#) on page 82), MemoryScape stops execution and displays its indicator symbol.

You can now display the **Manage Processes | Process Event** screen ([Figure 57](#) on page 92). This is described in the previous section of this task.

Where to Go Next

- To visually display information about your memory use, see [Task 7: “Graphically Viewing the Heap”](#) on page 94.
- After you stop execution, you will want to obtain reports on memory activity. You will find a list of these tasks in the introduction to this chapter (“[Memory Tasks](#)” on page 60).

Task 7: Graphically Viewing the Heap

This task is the first of a set of tasks that explain how to explore and examine your program's memory. Other important tasks for examining memory are [Task 9: "Seeing Leaks"](#) on page 102 and [Task 11: "Viewing Corrupted Memory"](#) on page 108.

Before reading this task, you should be familiar with:

[Locating Memory Problems,"](#) on page 1

An overview of memory concepts and MemoryScape.

[Task 1: "Getting Started"](#) on page 61

How to start MemoryScape and an overview of the kinds of information you can obtain.

[Task 3: "Setting MemoryScape Options"](#) on page 71

Describes how to configure MemoryScape so that it performs the activities you want it to perform.

[Task 4: "Controlling Program Execution"](#) on page 82

Shows how to start and stop program execution under MemoryScape control.

To display a grid showing how your program is using the heap, select **Memory Reports | Heap Status**, and then click **Heap Status Graphical Report**. (See [Figure 61](#) on page 95.)

If your program has more than one process or thread, you should select the thread or process of interest in the **Current Processes** area on the left side of the screen.

Topics in this task are:

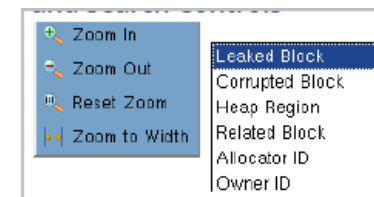
- ["Window Sections"](#) on page 94
- ["Block Information"](#) on page 96
- ["Bottom Tabbed Areas"](#) on page 96
- ["Where to Go Next"](#) on page 96

Window Sections

The **Heap Status Graphical Report screen** has three sections. The top has controls for viewing and filtering the graph. The middle contains a set of blocks that represent your memory allocations, and the bottom section provides information about blocks that you select in the middle. This bottom section can contain either heap information or a Backtrace report. (For information on Backtrace reports, see [Task 8: "Obtaining Detailed Heap Information"](#) on page 97.)

The grid is this screen's most prominent feature. As most programs use more memory than can easily be displayed, you can use the zoom controls (identified by the magnifying glass in the top area) to either zoom out to see more information or zoom in to get a better view.

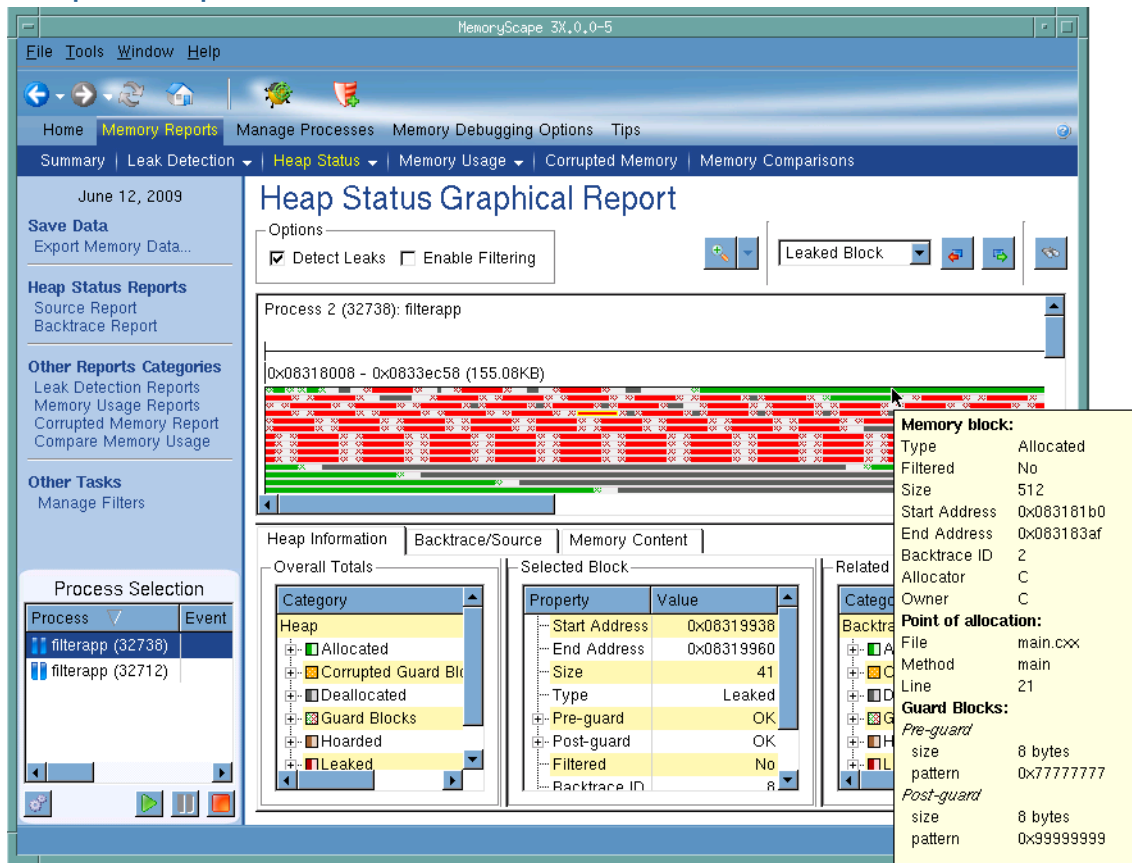
Figure 60: Zoom and Search Controls



You can also use the search controls to move from memory block to memory block using the curved arrow buttons in the control area. The search button tells MemoryScape what kind of block to look for. There are four choices, shown in [Figure 60](#).

By default, leaks are not marked. If, however, you select the **Detect Leaks** check box, MemoryScape takes a moment to analyze the heap. It then displays leaked memory blocks in red, [Figure 61](#).

Figure 61: Heap Status Graphical Report Screen



If you select the **Enable Filtering** check box, MemoryScape applies filters to the display. Unlike other reports where filtered information is removed, this information is displayed in gray. For information on filtering, see [Task 10: "Fil-](#)

[tering Reports"](#) on page 103.

Block Information

If you place the cursor over a block, MemoryScape opens a pop-up containing information about the block, [Figure 61](#). The information in this window is, of course, specific to the block. For example, this pop-up shows that guard blocks were being used when the block was allocated.

When you select a block, MemoryScape highlights all other blocks that have the same backtrace. So, if 100 different blocks share the same backtrace—which means that the execution path within the program is the same, and they were allocated from the same place in your program—they'll be highlighted. (Full details can be found in [Task 8: “Obtaining Detailed Heap Information”](#) on page 97.)

Bottom Tabbed Areas

The left most block within the **Heap Information** tab at the bottom of the screen summarizes of the type of information that can be displayed as well as explains the color coding used for these blocks, [Figure 62](#).

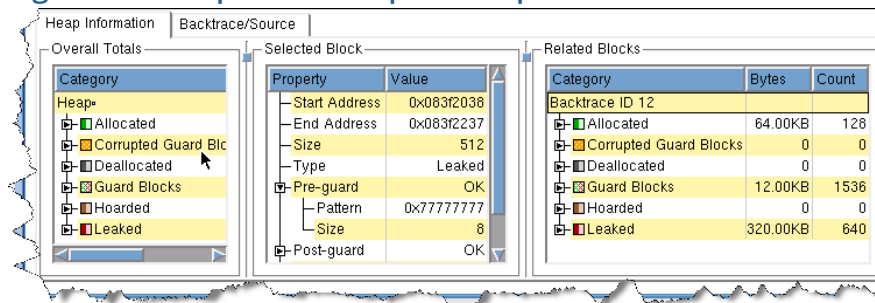
When you select a block, MemoryScape places information in the center and right boxes within the **Heap Information** tab. This information in the center list is the same as in the pop-up displayed when you place the mouse over a block. The information on the right is summary information about related blocks.

The **Memory Content** tab displays the bytes stored in memory. For more information, see [“Viewing Memory Contents”](#) on page 110.

Where to Go Next

- Typically, the **Heap Status Graphical Report** is a starting point for viewing other reports. For a list of reports, see to the introduction of this chapter, [“Memory Tasks”](#).
- If you want to continue execution, use the execution controls described in [Task 4: “Controlling Program Execution”](#) on page 82.
- You may want to set notifications that tell you when a block is deallocated or reused. For more information, see [“Deallocation and Reuse Notifications”](#) on page 92.

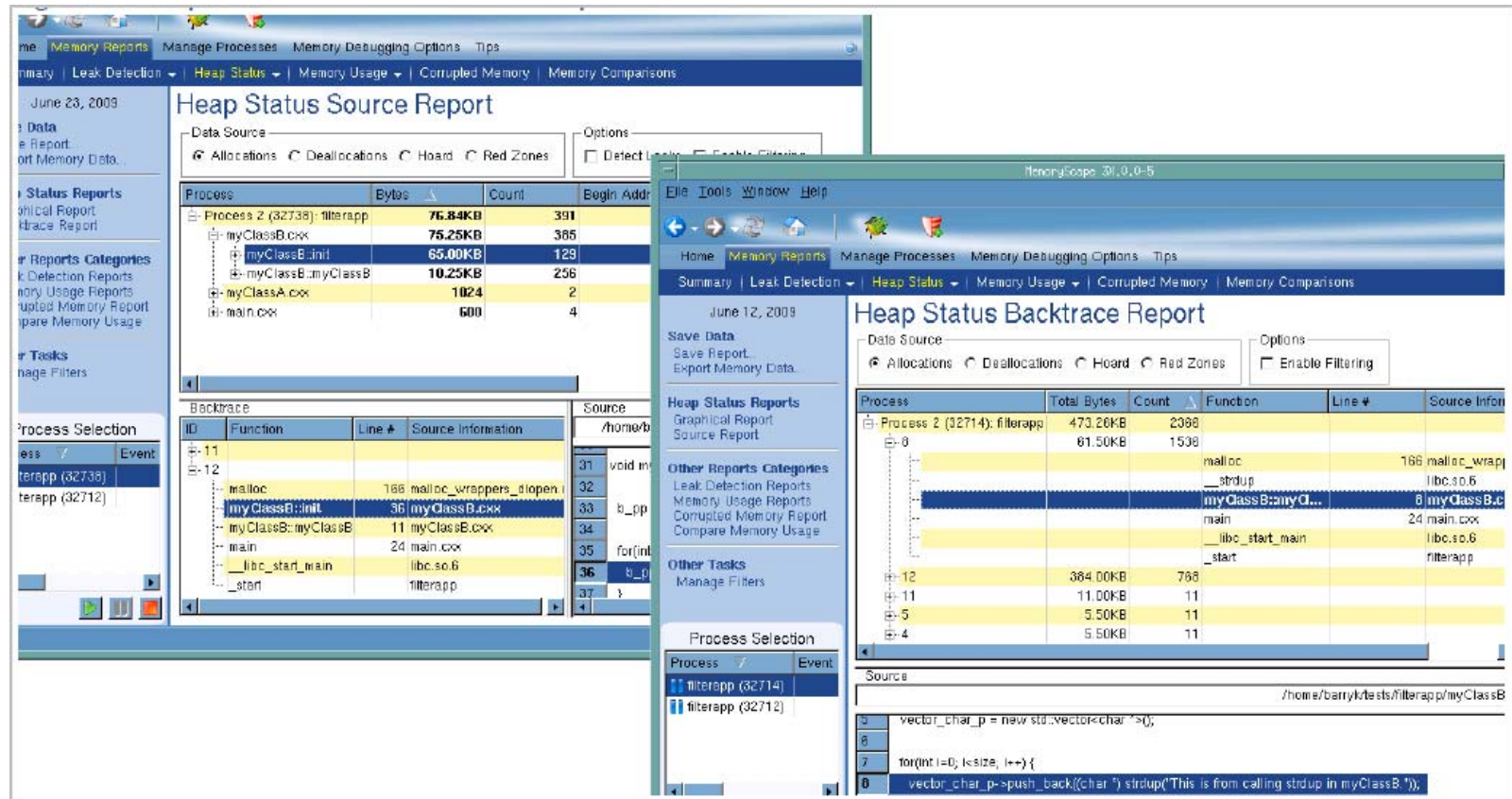
Figure 62: Heap Status Graphical Reports: Tabbed Area



Task 8: Obtaining Detailed Heap Information

This task discusses the Source and Backtrace reports, Figure 63.

Figure 63: Heap Status Source and Backtrace Reports



These reports are the most important sources of information within MemoryScape. While these reports relate directly to the Heap Status reports, they are available throughout MemoryScape as separate reports or as areas within a report.

The totals shown in the Heap Status reports may differ slightly from those in the Memory Usage reports, because heap status is generated from monitoring program requests for memory (**malloc** or **new**) and program release of memory (**free** or **delete**), while memory usage data is obtained from the operating system facilities. Depending on the operating system, memory usage totals may include anonymous memory regions that the program or one of its libraries may have mapped into its address space. The totals also include a small amount of overhead from MemoryScape itself.

The Memory Usage report is intended to be a quick check of your program's memory usage from the system's perspective. For detailed information on your program's use of the heap, use the Heap Status reports.

Which report to use, or when to use it, depends on how you want to approach the data. In general, most users prefer to approach this kind of information through Source reports. However, some like to see memory information organized on the way the program creates blocks.

Before reading this task, you should be familiar with the following information:

[Locating Memory Problems,](#) on page 1

An overview of memory concepts and MemoryScape.

[Task 1: "Getting Started"](#) on page 61

How to start MemoryScape and a summary of the kinds of information you can obtain.

[Task 4: "Controlling Program Execution"](#) on page 82

How to start and stop program execution.

[Task 7: "Graphically Viewing the Heap"](#) on page 94

How to obtain a graphic overview of the way this heap is laid out, as well as how to get more detailed information about individual blocks.

Topics in this task are:

- ["Heap Status Source Report"](#) on page 98
- ["Heap Status Source Backtrace Report"](#) on page 101
- ["Where to Go Next"](#) on page 101

Heap Status Source Report

The Source report (the top screen in [Figure 63](#) on page 97) contains three scrolling areas as well as a top control area. The area **Data Source** specifies the information to display: allocations, deallocations, or the hoard. In most cases, you'll want to see allocations. You cannot combine these kinds of information into one report.

By default, MemoryScape does not show leaks. To see them, select the **Detect Leaks** check box. If you have created a filter, apply it to the display by using the **Enable Filtering** check box. (For information on filtering, see [Task 10: "Filtering Reports"](#) on page 103.)

The top area organizes information by your program's source files. The **Bytes** column contains the amount of memory associated with a line in each file. Sorting on the **Bytes** column while displaying leaks helps to focus on those that waste the most memory; why focus on a leak that is under 1K when there's a 10M leak?

By repeatedly clicking, you'll soon get to the line in your program from which memory was allocated. The information here shows each block allocated from this line and the backtrace ID associated with it.

There is a distinction between backtraces associated with the statement and the statement in your program that allocates memory. Suppose you have a function called **create_list()**. This function could be called from many different places in your code, and each location will have a separate backtrace. For example, if **create_list()** is called from eight different places and each was called five times, there would be eight different backtraces (and eight different backtrace IDs) associated with it. Each individual backtrace would have five items associated with it.

As you click on lines in the top portion, the bottom right area shows lines from your source code. The line mentioned in the top area is highlighted here, Figure 64.

Figure 64: Uncovering Information

The figure consists of three screenshots of the Valgrind Memcheck tool interface, illustrating how to uncover information about memory allocations.

Top Left Screenshot: Shows the 'Data Source' panel with 'Allocations' selected. The 'Options' panel has 'Detect Leaks' checked and 'Enable Filtering' unchecked. The main table lists memory allocations for 'Process 1 (18413): filterapp':

Process	Bytes	Count	Begin Address	End Address
Process 1 (18413): filterapp	1266.11KB	6316		
myClassB.cxx	1219.00KB	6175		
myClassA.cxx	31.00KB	62		
main.cxx	16.11KB	79		

Top Right Screenshot: Shows a detailed view of a specific memory block. The 'Data Source' panel is the same. The 'Options' panel is the same. The main table shows details for 'myClassB.cxx':

Process	Bytes	Count	Begin Address	End Address
myClassB.cxx	1219.00KB	6175		
myClassB::init	1055.00KB	2079		
Line 36	1024.00KB	2048		
Line 33	31.00KB	31		
Block 11.31	1024	1	0x09ce7868	0x09ce786f
Block 11.30	1024	1	0x09ce7460	0x09ce746f
Block 11.29	1024	1	0x09cd2b58	0x09cd2b5f
Block 11.28	1024	1	0x09cd2750	0x09cd275f
Block 11.27	1024	1	0x09cbe050	0x09cbe05f

The 'Backtrace' panel shows the following information:

ID	Function	Line #	Source Information
11	myClassB::init	33	myClassB.cxx
11	myClassB::myClassB	11	myClassB.cxx
11	main	24	main.cxx
11	__libc_start_main		libc.so.6
11	_start		filterapp

Bottom Left Screenshot: Shows a detailed view of another memory block. The 'Data Source' panel is the same. The 'Options' panel is the same. The main table shows details for 'myClassB.cxx':

Process	Bytes	Count	Begin Address	End Address
Process 1 (18413): filterapp	1266.11KB	6316		
myClassB.cxx	1219.00KB	6175		
myClassB::init	1055.00KB	2079		
Line 36	1024.00KB	2048		
Line 33	31.00KB	31		
myClassB::myClassB	164.00KB	4096		
myClassA.cxx	31.00KB	62		
main.cxx	16.11KB	79		

The 'Backtrace' panel shows the following information:

ID	Function	Line #	Source Information
11	myClassB::init	33	myClassB.cxx
11	myClassB::myClassB	11	myClassB.cxx
11	main	24	main.cxx
11	__libc_start_main		libc.so.6
11	_start		filterapp

The 'Source' panel shows the source code snippet for 'myClassB.cxx':

```

28
29
30
31 void myClassB::init(void) {
32
33     b_pp = (int **) malloc (size * sizeof(int *));
34

```

When you need to see the backtrace, just click on the backtrace ID at the bottom left. As you select different levels in the stack in the backtrace, the display in the source area changes.

Heap Status Source Backtrace Report

The Backtrace Report (the bottom screen in [Figure 63](#) on page 97) contains similar information, organized by backtraces rather than source files. The backtrace ID displayed here is just a number that MemoryScape creates, useful in helping you coordinate information in different screens and tabs as it doesn't change from report to report.

Clicking on a line in the top portion of the backtrace displays a source line.

Where to Go Next

- These reports can contain too much information, which can be simplified using filters. See [Task 10: “Filtering Reports”](#) on page 103 for more information.
- [Task 14: “Saving Memory Information as HTML”](#) on page 116 describes one method of saving memory information. [Task 12: “Saving and Restoring Memory State Information”](#) on page 111 describes a second.
- [Task 13: “Comparing Memory”](#) on page 113 contains information on comparing more than one memory state.

Task 9: Seeing Leaks

The Leak reports are essentially the same as the **Heap Status Source** and **Heap Status Backtrace** reports. Honing in on the information you want is simpler in these reports as there are fewer controls and less data. For example, there is no **Data Source** area. The real difference between a **Heap Status Source** report and a **Leak Source** report is that a **Leak Source** report does not show allocations that are not leaked. That is, these two reports focus only on leaked memory.

As using these Leak reports is identical to using Heap Status reports, see [Task 8: “Obtaining Detailed Heap Information”](#) on page 97.

Task 10: Filtering Reports

The amount of information displayed in a Leak Detection or Heap Status report can be considerable. In addition, this information includes memory blocks allocated within all libraries, shared or otherwise, that your program uses. In other cases, your program may be allocating memory in many different ways, and you want to focus on only a few of them. This task shows you how to use filters to eliminate information from reports.

Before reading this task, you should be familiar with the following information:

- [Locating Memory Problems,](#)” on page 1
An overview of memory concepts and MemoryScape.
- [Task 1: “Getting Started”](#) on page 61
How to start MemoryScape with a summary of the kinds of information you can obtain.
- [Task 3: “Setting MemoryScape Options”](#) on page 71
How to configure MemoryScape so that it performs the activities you want it to perform.
- [Task 4: “Controlling Program Execution”](#) on page 82
How to start and stop program execution.

Creating Reports

How to create reports, discussed in the introduction to this chapter (“[Memory Tasks](#)” on page 60).

When filtering is enabled, MemoryScape considers each enabled filter and applies it to the report’s data. Filters can have any number of actions associated with them. Enable a filter by selecting the check box at the top of a report.

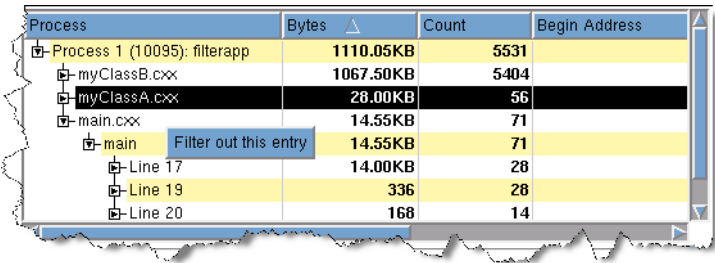
Topics in this task are:

- [“Adding, Deleting, Enabling and Disabling Filters”](#) on page 103
- [“Where to Go Next”](#) on page 107

Adding, Deleting, Enabling and Disabling Filters

To begin filtering data, right-click on a routine name or line number in the top pane of a Leak Detection, Heap Status Source or Backtrace report, or in a Heap Status Graphical report, and select **Filter out this entry** from the context menu.

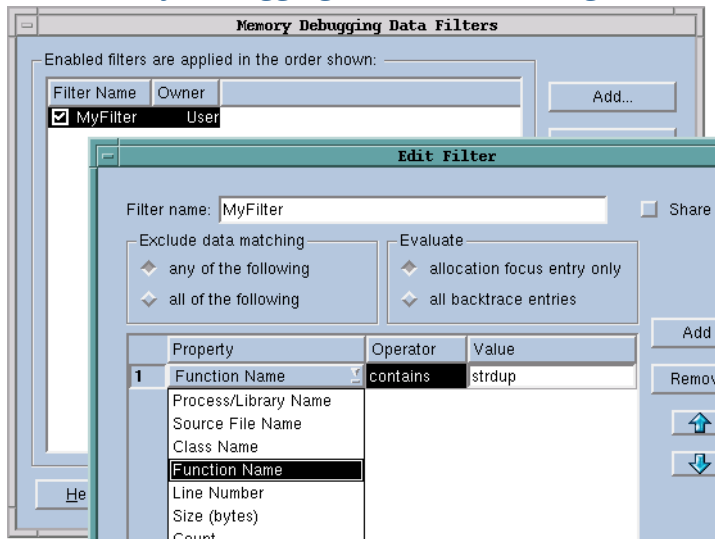
Figure 65: Filter out this entry Context Menu



This command adds the routine to those that can be filtered. It does not actually enable filtering; enabling is done by selecting the **Enable Filtering** check box in the command area of many screens.

To apply the filter, select **Manage Filters**, which is in the **Operations** area on the left side of the screen, [Figure 66](#).

Figure 66: Memory Debugging Data Filters Dialog Box



The controls in this dialog box:

☒ **Enable/Disable**

When checked, MemoryScape enables the filter.

Add Displays the **Add Filter** dialog box where you define a filter (discussed later in this section in [Adding and Editing Filters](#)).

Edit Displays a dialog box where you can change the selected filter's definition. The displayed **Edit Filter** dialog box is identical to the **Add Filter** dialog box.

Remove

Deletes the selected filter.



Up and Down

Moves a filter up or down in the filter list. Filters are applied in the order in which they appear, so you should place filters that remove the most entries at the top of the list. Filtering can be a time-consuming operation, so the right order can increase performance.

Enable All

Enables (checks) all filters in the list.

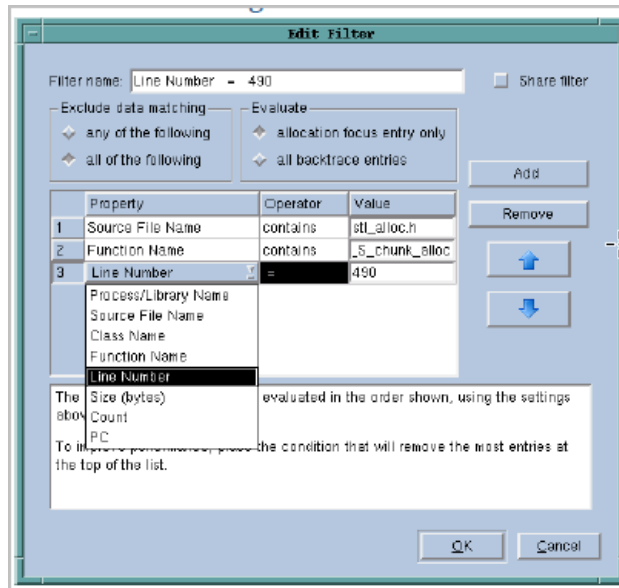
Disable All

Disables (unchecks) all filters in the list.

Adding and Editing Filters

Selecting the **Add** button in the **Memory Debugging Data Filters** dialog ([Figure 66](#)), launches the **Add Filter** dialog, [Figure 67](#).) Similarly, clicking the **Edit** button launches a nearly identical window.)

Figure 67: Add Filter Dialog Box



The controls in this window are:

Filter name

The name of the filter. This name will appear in the **Memory Debugging Data Filters** dialog box.

In [Figure 67](#), notice that one filter is named “**Function Name contains _S_chunk_alloc**”. This is the name created by MemoryScape when you use the context menu to add a function name. Similarly, you’ll see a filter named “**Line Number = 490**”.

Share filter

Creates a shared filter. *Shared* means that anyone using MemoryScape can use the filter.

NOTE >> This button appears only if you have write permissions for the MemoryScape lib directory.

Add Adds a blank line beneath the last criterion in the list. You can now enter information defining another criterion for this filter in this new line.

Remove

Deletes the selected criterion. To select a criterion, select the number to the left of the definition.

Up and Down

Changes the order in which criteria appear in the list. Criteria are applied in the order in which they appear, so you should place criteria that remove the most entries at the top of the list. Filtering can be a time-consuming operation, so this can increase performance.

Exclude data matching

For more than one criterion, the selected radio button indicates if *any* or *all* of the criteria have to be met.

any of the following

Removes an entry when the entry matches any of the criteria in the list.

all of the following

Removes a memory entry only if it fulfills all of the criteria.

Evaluate

Limits which backtraces MemoryScape looks at.

allocation focus entry only

Removes an entry only if the criteria is valid on an entry that is also the allocation focus.

The allocation focus is the point in the backtrace where MemoryScape believes your code called **malloc()**.

For example, if you define a filter condition that says **Function Name contains my_malloc** and set this entry to **allocation focus entry only**, MemoryScape removes entries only whose allocation focus contains **my_malloc**. That is, it removes only allocations that originated from **my_malloc**.

In contrast, if you set this entry to **all backtrace entries**, MemoryScape removes all blocks that contain **my_malloc** anywhere in their backtrace.

all backtrace entries

Applies filter criteria to all function names in the backtrace.

Criteria

A filter is made up of criteria. Each criterion specifies what to eliminate from the list. Each criterion has three parts: a property, an operator, and a value. For example, you can look for a Process/Library Name (the *property*) that contains (the *operator*) **strdup** (the *value*).

Property

When evaluating an entry, MemoryScape can look at one of eight properties for one criterion, [Figure 68](#). Select one of the items from the pulldown list. These items are:

Backtrace ID	PC
Class Name	Process/Library Name
Count	Size (bytes)
Function Name	Source File Name
Line Number	

Operator

Indicates the relationship the *value* has to the *property*, [Figure 68](#). Select one of the items from the pulldown list. If the property is a string, MemoryScape displays the following list:

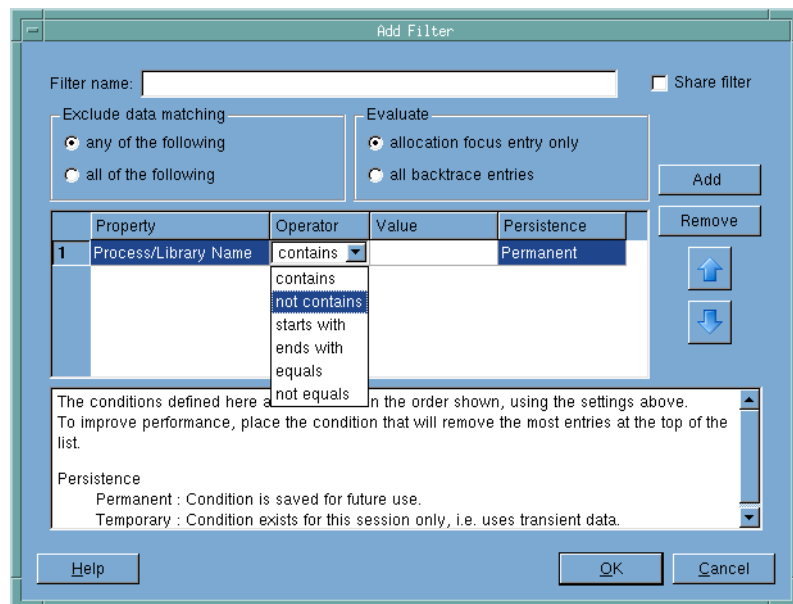
contains	not contains
ends with	starts with
equals	not equals

If the item is numeric, it displays the following list:

<=	>=
<	>
=	!=

Value A string or a number that indicates what to compare.

Figure 68: Add Filter Dialog Box



The "Add Filter" dialog box is shown. It has a title bar "Add Filter". Inside, there is a "Filter name:" text box and a "Share filter" checkbox. Below these are two sections: "Exclude data matching" with radio buttons for "any of the following" (selected) and "all of the following"; and "Evaluate" with radio buttons for "allocation focus entry only" (selected) and "all backtrace entries". To the right of these are "Add" and "Remove" buttons. Below is a table with columns: Property, Operator, Value, and Persistence. The first row has "1" in the first column, "Process/Library Name" in the second, "contains" in the third (with a dropdown menu open showing "contains", "not contains", "starts with", "ends with", "equals", and "not equals"), and "Permanent" in the fourth. To the right of the table are up and down arrow buttons. At the bottom left is a "Help" button, and at the bottom right are "OK" and "Cancel" buttons. A text area at the bottom contains instructions: "The conditions defined here are evaluated in the order shown, using the settings above. To improve performance, place the condition that will remove the most entries at the top of the list." and a "Persistence" section with "Permanent : Condition is saved for future use." and "Temporary : Condition exists for this session only, i.e. uses transient data."

	Property	Operator	Value	Persistence
1	Process/Library Name	contains		Permanent

Where to Go Next

After creating a filter, you'll want to use it. You'll find a list of tasks associated with reports at the beginning of this chapter ("[Memory Tasks](#)" on page 60).

Task 11: Viewing Corrupted Memory

If your program writes data either immediately before or immediately after an allocated block, it can alter data that other parts of the program will use. That is, this error means that these values are not what they are expected to be. This task shows how MemoryScape can help you with this problem.

Before reading this task, you should be familiar with the following information:

[Locating Memory Problems,](#)” on page 1

An overview of memory concepts and MemoryScape.

[Task 1: “Getting Started”](#) on page 61

How to start MemoryScape and a summary of the kinds of information you can obtain.

[Task 3: “Setting MemoryScape Options”](#) on page 71

How to configure MemoryScape so that it performs the activities you want it to perform.

[Task 10: “Filtering Reports”](#) on page 103

How you can remove information from the report that you do not need or want to see.

Topics in this task are:

- [“Examining Corrupted Memory Blocks”](#) on page 108
- [“Viewing Memory Contents”](#) on page 110

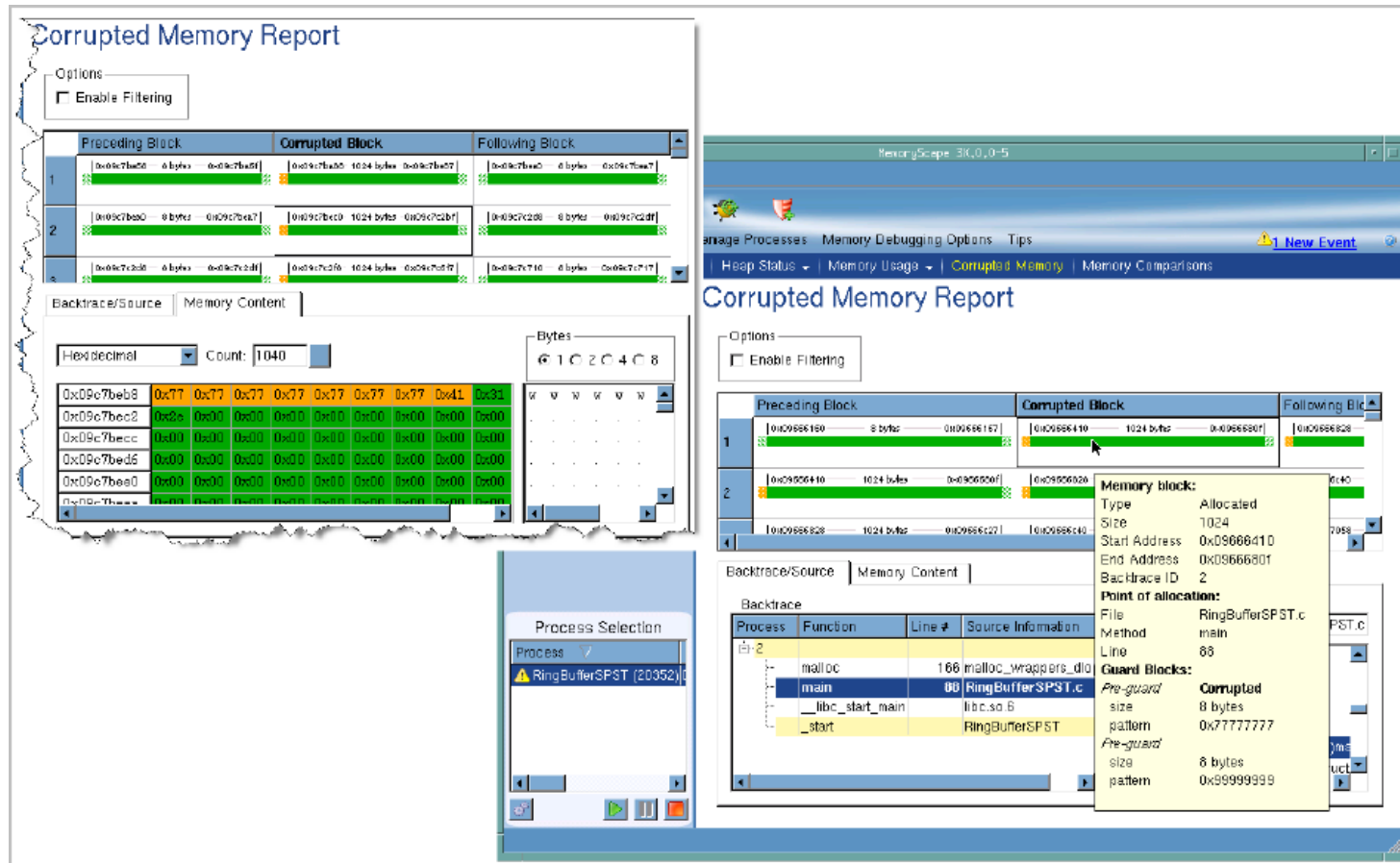
Examining Corrupted Memory Blocks

When your program uses functions in the malloc library, MemoryScape intercepts them, recording the action you have requested. If you enable guard blocks—by selecting **Medium** in the **Memory Options** screen—the MemoryScape agent also writes small blocks of information before and after all blocks that your program allocates. These blocks are called *guard blocks*. The guard block preceding the allocated block is initialized to one value (by default, this value is 0x77777777). The guard block following the allocated block is initialized to a second value (by default, 0x99999999). Because they are different, MemoryScape can determine which block was the source of the problem.

If your program writes data into either of the guard blocks, it changes this pattern. MemoryScape can detect this change in two ways:

- When your program deallocates a memory block, MemoryScape checks the guard block. If your program altered a guard block, MemoryScape stops execution and notifies you about the data corruption. ([Task 6: “Using Runtime Events”](#) on page 90.)
- You can halt execution (see [Task 4: “Controlling Program Execution”](#) on page 82) and display the **Memory Reports | Corrupted Memory** screen. This setting has MemoryScape examine all guard blocks and check for changes. If any are found, it displays a report similar to that shown on the right side of [Figure 69](#).

Figure 69: Corrupted Memory



The Corrupted Memory report contains two sections: a top section graphically displaying each corruption, and a bottom section containing a backtrace and the allocation source line for the allocated block. This is the same kind of information that is displayed in other reports. For information on these reports, see [Task 8: "Obtaining Detailed Heap Information"](#) on page 97.

If you place your cursor over a block, MemoryScope displays additional information about the block. In addition, if you right-click on a block and select **Properties** from the context menu, MemoryScope displays its **Block Properties** window. This window contains all the information that appears in the

pop-up display and it may contain additional information. For more on using the **Block Properties** window, see [Task 6: “Using Runtime Events”](#) on page 90.

Viewing Memory Contents

The Memory Contents tab directly displays the information in memory. See [Task 11: “Viewing Corrupted Memory”](#) on page 108. This information is presented in a manner similar to such shell tools as **od**. Controls in this tab specify how MemoryScape should display this information. For example, you can choose hexadecimal, octal, and character. (The default is hexadecimal) You can also change the number of bytes shown. The right area displays an ASCII representation of this information. The **Bytes** area lets you specify how many bytes are contained within each cell.

In this figure, notice that bytes are displayed using the same color as in the selected block at the top of the figure.

- Orange indicates a corrupted guard block.
- Dark green indicates an allocated data block
- Light green indicates an uncorrupted guard block.

Now look at the data. When MemoryScape created a *pre* guard block, it used its default setting, which set bytes to **0x77**. However, the eighth byte in the corrupted guard block has a value of **0x41**. All other values are **0x77**.

Notice also that the boundary between the data in the corrupted block and the following block has two guard blocks: the following guard from one data allocation and the preceding guard of a second.

When the program that generated this program was run, guard block notification was set. This insured that execution was halted when the memory block containing the guard block was deallocated. This is a good starting point for trying to locate the cause of the problem.

Task 12: Saving and Restoring Memory State Information

In many cases, obtaining data from an executing program doesn't give you the information you need. What you may really need is to examine a past state or compare memory states over time. This task describes saving state information and bringing it back into MemoryScape so that it can be examined.

Before reading this task, you should be familiar with the following information:

[Locating Memory Problems,](#) on page 1

An overview of memory concepts and MemoryScape.

[Task 1: "Getting Started"](#) on page 61

How to start MemoryScape with an overview of the kinds of information you can obtain.

[Task 3: "Setting MemoryScape Options"](#) on page 71

How to configure MemoryScape so that it performs the activities you want it to perform.

[Task 13: "Comparing Memory"](#) on page 113

The procedure for comparing two memory states.

Topics in this task are:

- ["Procedures for Exporting and Adding Memory Data"](#) on page 111
- ["Using Saved State Information"](#) on page 111
- ["Where to Go Next"](#) on page 112

Procedures for Exporting and Adding Memory Data

To save state information:

1. Select **Export Memory Data**, located on the left side of most reports.
2. Enter a file name into the dialog box.

If you are exporting a multiprocess program, MemoryScape places information for each process in its own file. For example, if the file name you enter is **monte_carlo**, MemoryScape appends a process number to each file so that it writes files named **monte_carlo0**, **monte_carlo1**, and so on.

The procedure for adding saved information is similar to adding a program to MemoryScape:


1. From the **Home | Add Program** screen, select **Add Memory debugging file**.
2. Enter the name of the file containing the saved state information.

Using Saved State Information

You can use saved state information in two ways:

- In many cases, you'll begin using saved state information in a **Compare Memory Report**.
- You can use saved state information in exactly the same way as state information for the currently executing program. That is, generate reports from the saved state information in exactly the same way as you would from an executing program.

MemoryScape saves a great deal of information in the saved state file. This means that you can compare the saved state information against an executing program or against other saved and recalled state information. You can also generate any report that you can generate from live state information. For example, you can display a **Leak Source** report based on the saved state information.

The sole difference between a live program and data from saved stated information is that MemoryScape displays a disk icon  next to the file's name.

Where to Go Next

- For information on using saved state information in comparisons, see [Task 13: “Comparing Memory”](#) on page 113.
- You can also save reports as HTML. See [Task 14: “Saving Memory Information as HTML”](#) on page 116.

Task 13: Comparing Memory

Previous tasks have shown you how to locate explicit memory problems. These problems are often the most obvious, even though they can be difficult to locate. A more difficult task is tracking down problems related to using too much memory. For these problems, you need to understand how your program is using memory over time.

Before reading this task, you should be familiar with the following information:

[Locating Memory Problems,](#) on page 1

An overview of memory concepts and MemoryScape.

[Task 1: “Getting Started”](#) on page 61

How to start MemoryScape with an overview of the kinds of information you can obtain.

[Task 12: “Saving and Restoring Memory State Information”](#) on page 111

How to save state information to disk and read it back into MemoryScape.

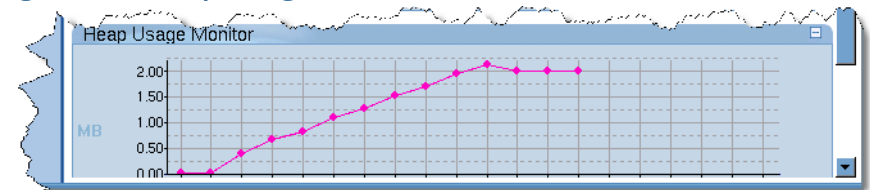
Topics in this task are:

- [“Overview”](#) on page 113
- [“Obtaining a Comparison”](#) on page 113
- [“Memory Comparison Report”](#) on page 114
- [“Where to Go Next”](#) on page 115

Overview

You would compare memory states after you notice that your program is using memory in a way that you don’t understand. For example, you are periodically displaying memory usage charts and you notice something unusual. Or, you see something unexpected on the **Home | Summary** screen’s Heap Usage Monitor.

Figure 70: Heap Usage Monitor



This graph shows that memory increases, then decreases slightly before remaining steady. This could be what you expect. However, it could be that you believe that your program should not have acquired so much memory.

Obtaining a Comparison

To create a comparison:

1. Stop your program while it is executing within the first plateau.
2. Save the memory state.
3. Run your program for a short time. Stop it at a later time.
4. Add the saved state back into MemoryScape by selecting **Add memory debugging file** from **Home | Add Program**. On the next screen, specify the previously saved state file.

5. Display the **Memory Reports | Memory Comparisons** screen.


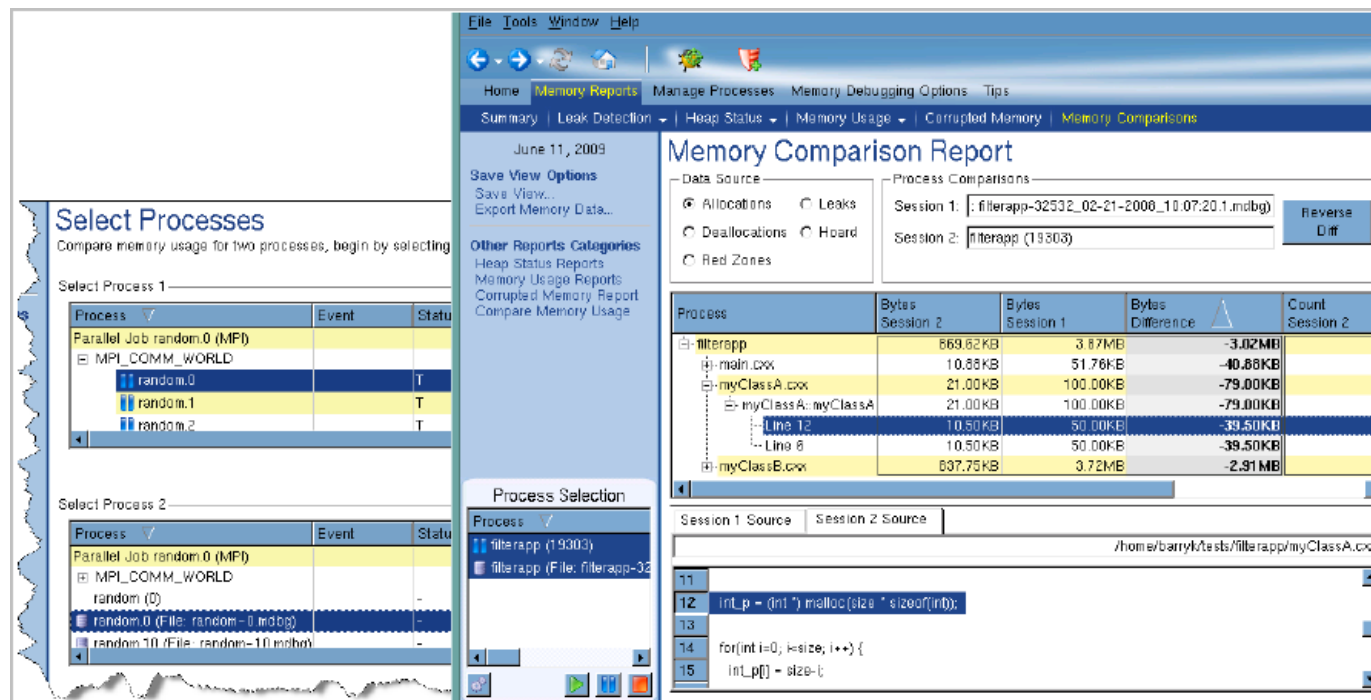
6. Select the two processes being compared. Note that the saved state has a small disk icon () next to it. **Figure 71** shows part of the screen containing controls for selecting the processes. The screen containing information is also displayed.

Figure 71: Memory Use Comparisons



Memory Comparison Report

Not unexpectedly, the information displayed in the **Memory Comparison Report** closely resembles the information in other reports. The buttons in the **Data Source** area control the kind of information being displayed, which

can be allocations, deallocations, leaks, or the hoard.

The comparison area displays the number of bytes allocated in each of the processes, and the difference between these values. In some cases, you may want to reverse the order in which MemoryScope compares information.

That is, MemoryScape compares processes in the order in which you selected the processes. Change this order by clicking the **Reverse Diff**



button.

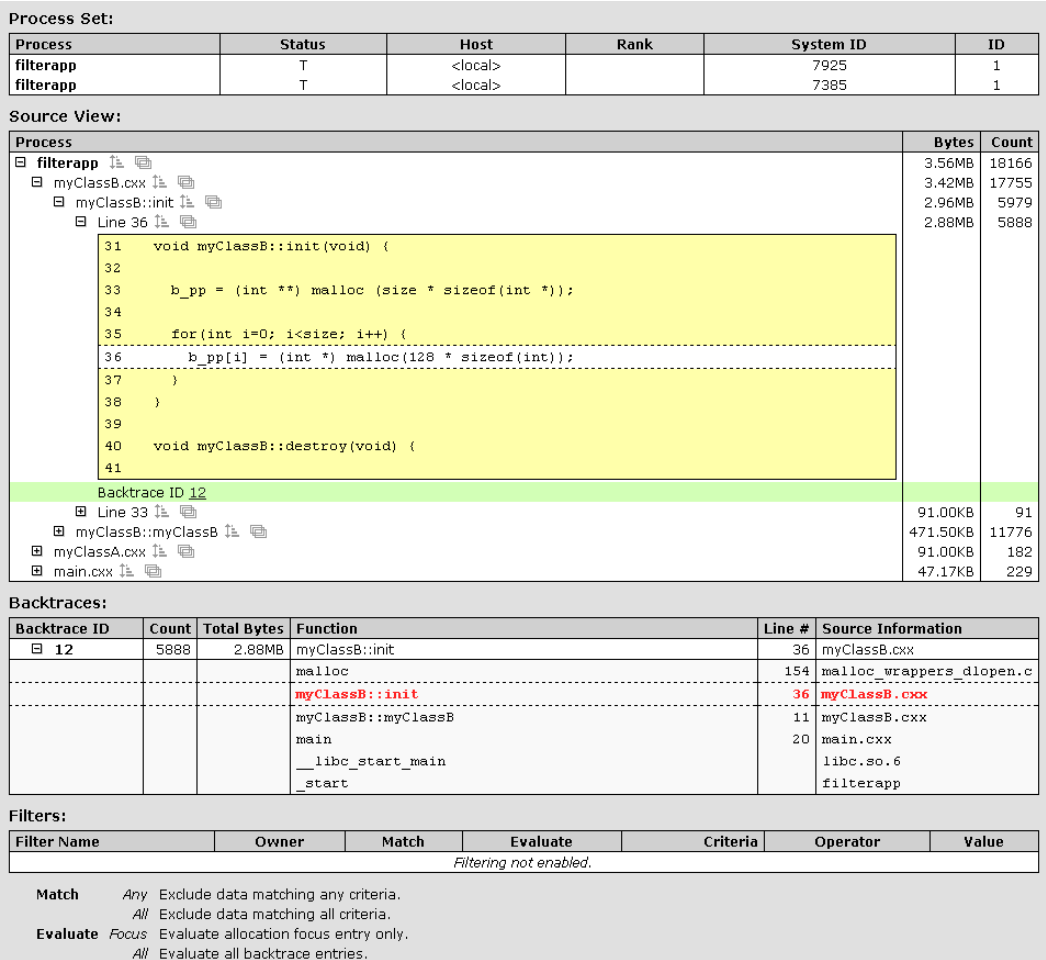
When you select a line in the table, MemoryScape displays the line in your program associated with it. Depending on what is being displayed and your program, these lines can differ. In many cases, however, they're identical.

Where to Go Next

Now that you've seen comparisons, you will probably want to obtain more information on using other MemoryScape reports. You'll find a summary at the beginning of this chapter ("[Memory Tasks](#)" on page 60).

Task 14: Saving Memory Information as HTML

Figure 72: Saved HTML File



Saving memory state and bringing it back into MemoryScape is one way for you to save memory state information. MemoryScape provides an alternative. You can save report information as an HTML file. (See [Figure 72](#) for an example.)

Before reading this task, you should be familiar with the following information:

[Locating Memory Problems,](#) on page 1

An overview of memory concepts and MemoryScape.

[Task 1: “Getting Started”](#) on page 61

How to start MemoryScape with an overview of the kinds of information you can obtain.

[Task 3: “Setting MemoryScape Options”](#) on page 71

How to configure MemoryScape so that it performs the activities you want it to perform.

[Task 12: “Saving and Restoring Memory State Information”](#) on page 111

How to save all memory data in a way that allows it to be brought back into MemoryScape.

Topics in this task are:

- [“Saving Report Information”](#) on page 117

Saving Report Information

To save report information:

1. Select **Save Report**, which is on the left side of most report screens.
2. Type a filename in the displayed dialog box.

There are many advantages and one disadvantage to saving information as HTML instead of exporting it. The disadvantage is that you need to decide what reports you want to save. That is, MemoryScape writes a set of HTML files for each report you want saved; if you don't save a report, there's no method for deriving another report from this saved information.

On the other hand, there are a number of advantages. The most important is that you can view the HTML report outside of MemoryScape in a browser. Another advantage is that you can share this information with others no matter where they are located, and these people need not have a MemoryScape license. One last advantage—and we're sure you'll have others—is that this makes it really easy to compare what you've done over time so that you can evaluate if you're making progress in solving memory problems.

Task 15: Hoarding Deallocated Memory

Hoarding is not an often-used feature. Its primary use is to prevent problems in which memory is deallocated by one part of the program but another part is using this memory, not knowing it is deallocated. For two examples of how hoarding is used, see “[Hoarding](#)” on page 56.

Before reading this task, you should be familiar with the following information:

[Locating Memory Problems,](#)” on page 1

An overview of memory concepts and MemoryScape.

[Task 1: “Getting Started”](#) on page 61

How to start MemoryScape with an overview of the kinds of information you can obtain.

[Task 3: “Setting MemoryScape Options”](#) on page 71

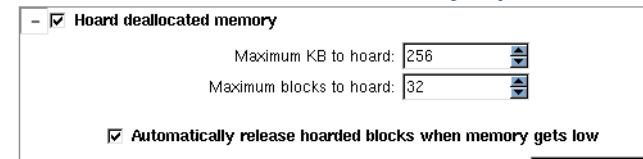
How to configure MemoryScape so that it performs the activities you want it to perform.

[Task 4: “Controlling Program Execution”](#) on page 82

How to start and stop program execution.

To enable memory hoarding, select **Extreme** from the **Memory Debugging Options** screen. (described in [Task 3: “Setting MemoryScape Options”](#) on page 71.) If you need to tune how MemoryScape hoards memory, select **Advanced Options**. [Figure 73](#) shows the portion of the options page where you set hoarding options.

Figure 73: Hoard deallocated memory Option



The screenshot shows a section of the MemoryScape Advanced Options dialog titled "Hoard deallocated memory". It contains two settings: "Maximum KB to hoard:" with a value of 256, and "Maximum blocks to hoard:" with a value of 32. Both values are in input fields with up and down arrows. Below these is a checkbox labeled "Automatically release hoarded blocks when memory gets low" which is checked.

Hoarding is actually a very simple process. When your program deallocates memory, MemoryScape stashes the deallocation request away. It also records information on the size of the block that would have been deallocated. And then it ignores the deallocation request.

Ignoring the deallocation request means that other parts of the program can continue to access this valid memory instead of using data that may have incorrect values. Said differently, the program can keep executing with the correct information a while longer. This improves your chances of identifying the location of the problem.

MemoryScape doesn't ignore your program's deallocations request indefinitely. By default, MemoryScape hoards 32 requests. Also by default, it defers only the deallocation of up to 256 KB of memory. You can, of course, change either of these values by using MemoryScape options.

To hoard all deallocated memory, set the maximum KB and blocks to **unlimited** by entering **0** in the hoarding control fields. To prevent or delay your program from running out of memory when using this setting, use the advanced option to set MemoryScape to automatically release hoarded memory when available memory gets low.

You can also set a threshold for the hoard size so MemoryScape can warn you when available memory is getting low. If the hoard size drops below the threshold, MemoryScape halts execution and notifies you. You can then view a Heap Status or Leak report to see where your memory is being allocated.

Task 16: Painting Memory

Your program may be using memory either before it is initialized or after it is deallocated. MemoryScape can help you identify these kinds of problems by initializing allocated or deallocated memory to a bit pattern. This is called *painting*. If you can recognize this bit pattern, you'll be taking a large step toward identifying the problem.

Before reading this task, you should be familiar with the following information:

[Locating Memory Problems,](#) on page 1

An overview of memory concepts and MemoryScape.

[Task 1: "Getting Started"](#) on page 61

How to start MemoryScape with an overview of the kinds of information you can obtain.

[Task 3: "Setting MemoryScape Options"](#) on page 71

How to configure MemoryScape so that it performs the activities you want it to perform.

[Task 4: "Controlling Program Execution"](#) on page 82

How to start and stop program execution.

To enable painting memory, select **Extreme** from the **Memory Debugging Options** screen (described in [Task 3: "Setting MemoryScape Options"](#) on page 71.) If you need to tune how MemoryScape paints memory, select **Advanced Options**. Here is the portion of the options screen for setting painting options:

Figure 74: Paint memory option



Here are some of the ways in which you use painting:

- If you paint memory and your application displays this memory, you will have proof that you are using uninitialized or deallocated memory.
- Painting memory provides consistency for uninitialized memory in that you should not experience situations where the program works for some users and doesn't for others.
- Painting memory will change your program's behavior if it is not using memory correctly. This may aid in identifying the problem. In addition, it may *correct* the problem so that the program doesn't appear to fail.
- Some painting patterns can force an error such as a crash to occur. (Crashes during debugging sessions are useful, allowing you to identify where problems are occurring.)
- After painting memory, your program can look for the pattern in order to check if something was not initialized.

Using Remote Display

Using the TotalView Remote Display client, you can start and then view TotalView and MemoryScape as they execute on another system, so that they need not be installed on your local machine.

Remote Display is currently bundled into all TotalView releases.

Supported platforms include:

- Linux x86-64
- Microsoft Windows
- Apple macOS Intel

No license is needed to run the Client, but TotalView running on any supported operating system must be a licensed version of TotalView 8.6 or greater.

Both MemoryScape and TotalView use the same client, discussed in the *Classic TotalView User Guide* in the chapter “Accessing TotalView Remotely.” No special configuration or use is required when using the client to access MemoryScape.

Creating Programs for Memory Debugging

MemoryScape tries to handle the details involved in starting your program, but differences in development and production environments may require customizations outside MemoryScape defaults.

This chapter contains information you'll need if our defaults do not meet your need. Topics in this chapter are:

- ["Compiling Programs"](#) on page 122
- ["Linking with the dbfork Library"](#) on page 123
- ["Ways to Start MemoryScape"](#) on page 125
- ["Attaching to Programs"](#) on page 126
- ["Setting Up MPI Debugging Sessions"](#) on page 127
- ["Linking Your Application with the Agent"](#) on page 136
- ["Using env to Insert the Agent"](#) on page 139
- ["Installing tvheap_mr.a on AIX"](#) on page 140
- ["Using MemoryScape in Selected Environments"](#) on page 142

Compiling Programs

The first step when preparing a program to load into MemoryScape is adding your compiler’s **-g** debugging command-line option to generate symbol table debugging information; for example:

```
cc -g -o executable source_program
```

You can also use MemoryScape on programs that you did not compile using the **-g** option, or programs for which you do not have source code. However, MemoryScape may not be able to provide source code information.

The following table presents some general considerations.

Compiler Option or Library	What It Does	When to Use It
Debugging symbols option (usually -g)	Generates debugging information in the symbol table.	Before debugging any program with MemoryScape.
Optimization option (usually -O)	<p>Rearranges code to optimize your program’s execution.</p> <p>Some compilers won’t let you use the -O option and the -g option at the same time.</p> <p>Even if your compiler lets you use the -O option, don’t use it when debugging your program, since unexpected results often occur.</p>	After you finish debugging your program.
Multiprocess programming library (usually dbfork)	<p>Uses special versions of the fork() and execve() system calls.</p> <p>In some cases, you need to use the -lpthread option.</p> <p>For more information about dbfork, see “Linking with the dbfork Library” on page 123.</p>	Before debugging a multiprocess program that explicitly calls fork() or execve().

Linking with the dbfork Library

If your program uses the **fork()** and **execve()** system calls, and you want to debug the child processes, you need to link programs with the **dbfork** library.

NOTE >> While you must link programs that use **fork()** and **execve()** with the **dbfork** library so that MemoryScape can automatically attach to them when your program creates them, programs that you attach to need not be linked with this library.

dbfork on IBM AIX on RS/6000 Systems

Add either the **-dbfork** or **-ldbfork_64** argument to the command that you use to link your programs. If you are compiling 32-bit code, use the following arguments:

- `/memscape_install_dir/lib/libdbfork.a \`
`-bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a`
- `-L/memscape_install_dir/lib \`
`-ldbfork -bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a`

For example:

```
cc -o program program.c \
-L/usr/totalview/rs6000/lib/ -ldbfork \
-bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a
```

If you are compiling 64-bit code, use the following arguments:

- `/memscape_install_dir/lib/libdbfork_64.a \`
`-bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a`

- `-L/memscape_install_dir/lib -ldbfork_64 \`
`-bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a`

For example:

```
cc -o program program.c \
-L/usr/totalview/rs6000/lib -ldbfork \
-bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a
```

When you use **gcc** or **g++**, use the **-Wl, -bkeepfile** option instead of the **-bkeepfile** option, which will pass the same option to the binder. For example:

```
gcc -o program program.c \
-L/usr/totalview/rs6000/lib -ldbfork -Wl, \
-bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a
```

Linking C++ Programs with dbfork

You cannot use the **-bkeepfile** binder option with the IBM xLC C++ compiler. The compiler passes all binder options to an additional pass called **munch**, which will not handle the **-bkeepfile** option.

To work around this problem, MemoryScape provides the C++ header file **libdbfork.h**. You must include this file somewhere in your C++ program. This forces the components of the **dbfork** library to be kept in your executable. The file **libdbfork.h** is included only with the RS/6000 version of MemoryScape. This means that if you are creating a program that will run on more than one platform, place the **include** within an **#ifdef** statement's range. For example:

```
#ifdef _AIX
#include "/usr/totalview/include/libdbfork.h"
#endif
int main (int argc, char *argv[])
{
}
```


In this case, you would not use the **-bkeepfile** option and would instead link your program using one of the following options:

- **/usr/totalview/include/libdbfork.a**
- **-L/usr/totalview/include -ldbfork**

dbfork and Linux or Mac OS X

Add one of the following arguments or command-line options to the command that you use to link your programs:

- **-L/usr/totalview/platform/lib**
or
-L/usr/totalview/platform/lib -ldbfork_64

where *platform* is one of the following: **darwin-x86**, **linux-x86-64**, **linux-arm64**, or **linux-powerle**.

For example:

```
cc -o program program.c \  
-L/usr/totalview/linux-x86-64/lib -ldbfork_64
```

dbfork and SunOS 5 SPARC

Add one of the following command line arguments or options to the command that you use to link your programs:

- **/opt/totalview/sun5/lib/libdbfork.a**
- **-L/opt/totalview/sun5/lib -ldbfork**

For example:

```
cc -o program program.c \  
-L/opt/totalview/sun5/lib -ldbfork
```

```
-L/opt/totalview/sun5/lib -ldbfork
```

As an alternative, you can set the **LD_LIBRARY_PATH** environment variable and omit the **-L** option on the command line:

```
setenv LD_LIBRARY_PATH /opt/totalview/sun5/lib
```

Ways to Start MemoryScape

MemoryScape can debug programs that run in many different computing environments and which use many different parallel processing modes and systems. This section looks at few of the ways you can start MemoryScape.

In most cases, the command for starting MemoryScape looks like this:

```
memscape [ executable [ corefile ] ] [ options ]
```

where *executable* is the name of the executable file and *corefile* is the name of the core file that you want to examine.

Your environment may require you to start MemoryScape in another way. For example, if you are debugging an MPI program, you may need to invoke MemoryScape on **mpirun**. For an example, see “[Debugging an MPI Program](#)” on page 126 and several other discussions in this chapter.

The following examples show different ways of starting MemoryScape:

Starting MemoryScape

memscape Starts MemoryScape without loading a program or core file. You now select **Add new program** or **Add parallel program** to load a program.

Starting MemoryScape and Naming a program

memscape executable

Starts MemoryScape and loads the *executable* program.

Starting MemoryScape from TotalView

From TotalView, select the **Tools** menu item from the Root Window or the **Debug** menu item from the Process Window. Select the **Open MemoryScape** option. At launch, MemoryScape tries to interpret the state of TotalView and opens to the appropriate page, most likely the Home page. You can select the **Memory Debugging Options** page to turn memory debugging on or off, but if your program is running, you must kill it before the settings take effect.

Using core files

memscape executable corefile

Starts MemoryScape and loads the *executable* program and the *corefile* core file.

Passing arguments to the program

memscape executable -a args

Starts MemoryScape and passes all the arguments following the **-a** option to the *executable* program. When you use the **-a** option, you must enter it as the last MemoryScape option on the command line.

If you don't use the **-a** option and you want to add arguments after MemoryScape loads your program, right click on the executable and select Properties.

Debugging a program that runs on another computer

memscape executable -remote hostname_or_address[:port]

Starts MemoryScape on your local host and the TotalView Debugger Server (**tvdsrv**) on a remote host. After MemoryScape begins executing, it loads the program specified by *executable* for remote debugging. You can specify a host name or a TCP/IP address. If you need to, you can also enter the TCP/IP port number.

If MemoryScape fails to automatically load a remote executable, you may need to disable *autolaunching* for this connection and manually start the Debugger Server (**tvdsvr**). (*Autolaunching* is the process of automatically launching **tvdsvr** processes.) You can disable autolaunching by adding the **hostname:portnumber** suffix to the name you type in the **Host** field of the **Add new program** or **Add parallel program** screens. As always, the **portnumber** is the TCP/IP port number on which our server is communicating with MemoryScape. For more information on how to disable autolaunching, see “Starting the TotalView Server Manually” in the *Classic TotalView User Guide*.

Debugging an MPI Program

memscape *executable*

(*method 1*) In many cases, you start an MPI program in much the same way as you would any other program, but you need to set its properties. One way is to select the executable's name from within MemoryScape, right-click for the context menu, and choose **Properties**. In the displayed dialog box, select the MPI version, in addition to other options.

mpirun -np *count* -tv *executable*

(*method 2*) The MPI **mpirun** command starts the MemoryScape pointed to by the **TOTALVIEW** environment variable. MemoryScape then starts your program. This program will run using **count** processes.

Attaching to Programs

If a program you're testing is using too much memory, you can attach to it while it is running. You can attach to single and multiprocess programs, and these programs can be running remotely.

NOTE >>MemoryScape requires that all programs use the MemoryScape agent. In most cases, it does this behind the scenes before the program begins executing. However, it cannot do this for an already executing program or for a core file. So, just attaching to an already running program will not provide the information you need as the agent won't be used. In some cases, you may want to add it by starting the program using the shell `env` command. However, the best alternative is to link the MemoryScape agent. For details, see “[Linking Your Application with the Agent](#)” on page 136.

To attach to a process, select the **Attach to running program** item from the **Home | Add Program** page.

NOTE >>When you exit from MemoryScape, it kills all programs and processes that it started. However, programs and processes that were executing before you brought them under MemoryScape's control continue to execute.

If you want MemoryScape to automatically attach to programs that use **fork()** and **execve()**, you must use the MemoryScape dbfork library. However, programs that you manually attach to need not be linked with this library.

Setting Up MPI Debugging Sessions

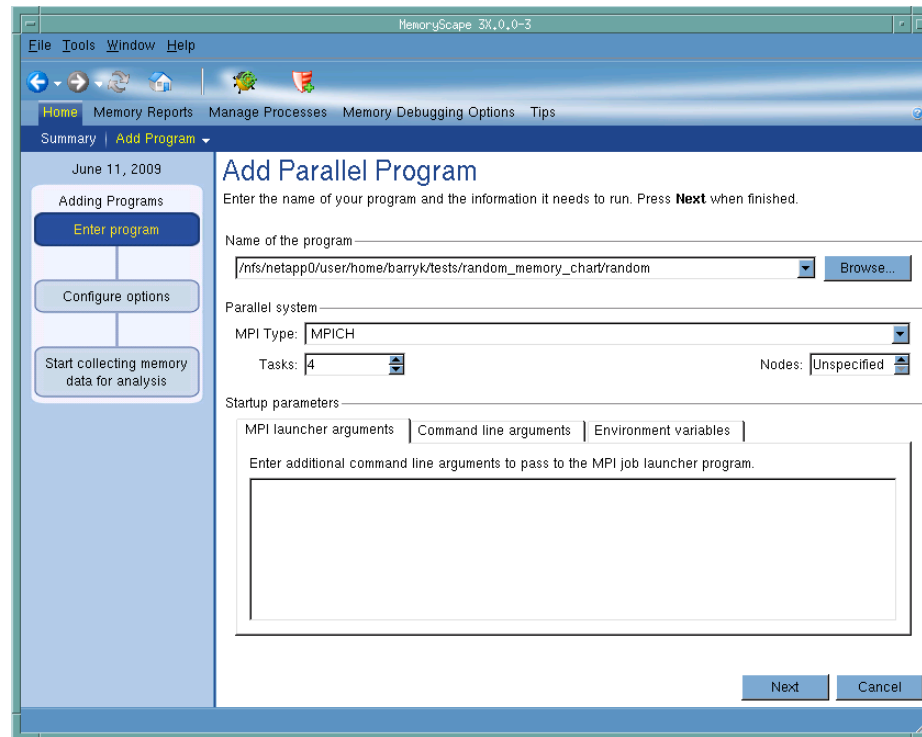
This section explains how to set up MemoryScape MPI debugging sessions. This section includes:

- “[Debugging MPI Programs](#)” on page 127
- “[Debugging MPICH Applications](#)” on page 129
- “[Debugging IBM MPI Parallel Environment \(PE\) Applications](#)” on page 131
- “[Debugging LAM/MPI Applications](#)” on page 133
- “[Debugging QSW RMS Applications](#)” on page 134
- “[Debugging Sun MPI Applications](#)” on page 134

Debugging MPI Programs

In many cases, the way in which you invoke an MPI program within MemoryScape control differs little from discipline to discipline. If you invoke MemoryScape from the command line without an argument, MemoryScape displays its **Add Programs to Your MemoryScape Session** screen. Select **Add parallel program**, and then enter the required information.

Figure 75: Adding a Parallel program



For example, you should select the **Parallel system**, the number of **Tasks**, and possibly **Nodes**. If there are additional arguments to send to the starter process, enter them in the **MPI launcher arguments** area. Note that these arguments are those sent to a starter process such as **mpirun** or **poe**; arguments sent to your program are instead entered in the **Command line arguments** area. If you need to add environment variables, enter them within the **Environment variables** tab.

In most cases, MemoryScope remembers your entries between invocations of MemoryScope and pre-populates the relevant fields.

Debugging MPICH Applications

NOTE >> In many cases, you can bypass the procedure described in this section. For more information, see “[Debugging MPI Programs](#)” on page 127.

To examine Message Passing Interface/Chameleon Standard (MPICH) applications, you must use MPICH version 1.2.5 or later on a homogenous collection of computers. If you need a copy of MPICH, you can obtain it at no cost from Argonne National Laboratory at <https://www.mcs.anl.gov/mpi>. (We strongly urge that you use a later version of MPICH.)

The MPICH library should use the **ch_p4**, **ch_p4mpd**, **ch_shmem**, **ch_Ifshmem**, or **ch_mpl** devices.

- For networks of workstations, the default MPICH library is **ch_p4**.
- For shared-memory SMP computers, use **ch_shmem**.
- On an IBM SP computer, use the **ch_mpl** device.

The MPICH source distribution includes all these devices. Choose the device that best fits your environment when you configure and build MPICH.

For more information, see:

- [Starting MemoryScape on an MPICH Job](#), below
- “[Attaching to an MPICH Job](#)” on page 130
- “[Using MPICH P4 procgroup Files](#)” on page 130

Starting MemoryScape on an MPICH Job

Before you can bring an MPICH job under MemoryScape’s control, both MemoryScape and the Debugger Server must be in your path, performed through either a login or shell startup script.

The *official* syntax for starting MemoryScape is as follows:

mpirun -tv [*other_mpic_args*] program [*program_args*]

For example:

```
mpirun -tv -np 4 sendrecv
```

The **-tv** option tells **mpirun** that it should obtain information from the **TOTALVIEW** environment variable.

For example, the following is the C shell command that sets the **TOTALVIEW** environment variable so that **mpirun** passes the **-verbosity** option to MemoryScape:

```
setenv TOTALVIEW "memscape -verbosity 1"
```

In this example, the **memscape** command must be in your path. If it isn’t, you need to specify either an absolute or relative path to the **memscape** command. MemoryScape begins by starting the first process of your job, the master process, under its control.

On the IBM SP computer with the **ch_mpl** device, **mpirun** uses the **poe** command to start an MPI job. While you still must use the MPICH **mpirun** (and its **-tv** option) command to start an MPICH job, the way you start MPICH differs. For details on using MemoryScape with **poe**, see “[Starting MemoryScape on a PE Program](#)” on page 132.

Starting MemoryScape using the **ch_p4mpd** device is similar to starting MemoryScape using **poe** on an IBM computer or other methods you might use on Sun platforms. In general, you start MemoryScape using the **memscape** command, with the following syntax;

```
memscape mpirun [ memscape_args ] -a [ mpich_args ] \  
program [ program-args ]
```

As your program executes, MemoryScape automatically acquires the processes that are part of your parallel job as your program creates them. MemoryScape automatically copies memory configuration information to the slave processes.

Attaching to an MPICH Job

You can attach to an MPICH application even if it was not started under MemoryScape's control. These processes, however, must have previously been linked with the MemoryScape agent. See [“Linking Your Application with the Agent”](#) on page 136.

To attach to an MPICH application:

1. Start MemoryScape.
Select **Attach to running program** from the **Add Programs to Your MemoryScape Session** screen. You are now shown processes that are not yet owned.
2. Attach to the first MPICH process in your workstation cluster by selecting it, then clicking Next.
3. On an IBM SP with the **ch_mpi** device, attach to the **poe** process that started your job. For details, see [“Starting MemoryScape on a PE Program”](#) on page 132.

Normally, the first MPICH process is the highest process with the correct program name in the process list. Other instances of the same executable can be:

- The **p4** listener processes if MPICH was configured with **ch_p4**.
- Additional slave processes if MPICH was configured with **ch_shmem** or **ch_lfshmem**.
- Additional slave processes if MPICH was configured with **ch_p4** and has a file that places multiple processes on the same computer.

4. After you attach to your program's processes, you are prompted to also attach to slave MPICH processes. To do so, press **Return** or choose **Yes**. If you choose **Yes**, MemoryScape starts the server processes and acquires all MPICH processes.

In some situations, the processes you expect to see might not exist (for example, they may crash or exit). MemoryScape acquires all the processes it can and then warns you if it can not attach to some of them. If you attempt to dive into a process that no longer exists (for example, using a message queue display), MemoryScape reports that the process no longer exists.

Using MPICH P4 procgroup Files

If you're using MPICH with a P4 **procgroup** file (by using the **-p4pg** option), you must use the *same* absolute path name in your **procgroup** file and on the **mpirun** command line. For example, if your **procgroup** file contains a different path name than that used in the **mpirun** command, even though this name resolves to the same executable, MemoryScape assumes that it is a different executable, which causes debugging problems.

The following example uses the same absolute path name on the MemoryScape command line and in the **procgroup** file:

```
% cat p4group
local 1 /users/smith/mympichexe
bigiron 2 /users/smith/mympichexe
% mpirun -p4pg p4group -tv /users/smith/mympichexe
```

In this example, MemoryScape does the following:

1. Reads the symbols from **mympichexe** only once.
2. Places MPICH processes in the same MemoryScape share group.
3. Names the processes **mympichexe.0**, **mympichexe.1**, **mympichexe.2**, and **mympichexe.3**.

If MemoryScape assigns names such as **mympichexe<mympichexe>.0**, a problem occurred and you need to compare the contents of your **proc-group** file and **mpirun** command line.

Starting MPI Issues

NOTE >> In many cases, you can bypass the procedure described in this section. For more information, see [“Debugging MPI Programs”](#) on page 127.

If you can't successfully start MemoryScape on MPI programs, check the following:

- Can you successfully start MPICH programs without MemoryScape?
The MPICH code contains some useful scripts that help verify that you can start remote processes on all computers in your computers file. (See **tst-machines** in **mpich/util**.)

- Does the TotalView Server (**tvdsrv**) fail to start?

Remember that MemoryScape uses **ssh -x** to start the server, and that this command doesn't pass your current environment to remotely started processes.

- Under some circumstances, MPICH kills MemoryScape with the **SIGINT** signal. You can see this behavior when you use the **Kill** command as the first step in restarting an MPICH job.

Debugging IBM MPI Parallel Environment (PE) Applications

NOTE >> In many cases, you can bypass the procedure described in this section. For more information, see [“Debugging MPI Programs”](#) on page 127.

You can debug IBM MPI Parallel Environment (PE) applications on the IBM RS/6000 and SP platforms.

To take advantage of MemoryScape's ability to automatically acquire processes, you must be using release 3.1 or later of the Parallel Environment for AIX.

Topics in this section are:

- [“Using Switch-Based Communications”](#) on page 132
- [“Performing a Remote Login”](#) on page 132
- [“Starting MemoryScape on a PE Program”](#) on page 132
- [“Attaching to a PE Job”](#) on page 133

The following sections describe what you must do before MemoryScape can debug a PE application.

NOTE >> You must link the MemoryScape agent into your IBM MPI Parallel Environment program before running it. For details, see “[Installing tvheap_mr.a on AIX](#)” on page 140.

Using Switch-Based Communications

If you're using switch-based communications (either *IP over the switch* or *user space*) on an SP computer, you must configure your PE debugging session so that MemoryScape can use *IP over the switch* for communicating with the TotalView Server (**tvdsvr**). Do this by setting the **-adapter_use** option to **shared** and the **-cpu_use** option to **multiple**, as follows:

- If you're using a PE host file, add **shared multiple** after all host names or pool IDs in the host file.
- Always use the following arguments on the **poe** command line:

```
-adapter_use shared -cpu_use multiple
```

If you don't want to set these arguments on the **poe** command line, set the following environment variables before starting **poe**:

```
setenv MP_ADAPTER_USE shared  
setenv MP_CPU_USE multiple
```

When using *IP over the switch*, the default is usually **shared adapter use** and **multiple cpu use**; we recommend that you set them explicitly using one of these techniques. You must run MemoryScape on an SP or SP2 node. Since MemoryScape will be using *IP over the switch* in this case, you cannot run MemoryScape on an RS/6000 workstation.

Performing a Remote Login

You must be able to perform a remote login using the **ssh -x** command. You also need to enable remote logins by adding the host name of the remote node to the **/etc/hosts.equiv** file or to your **.rhosts** file.

When the program is using switch-based communications, MemoryScape tries to start the TotalView Server by using the **ssh -x** command with the switch host name of the node.

Setting Timeouts

If you receive communications timeouts, you can set the value of the **MP_TIMEOUT** environment variable; for example:

```
setenv MP_TIMEOUT 1200
```

If this variable isn't set, the default **timeout** value is 600 seconds.

Starting MemoryScape on a PE Program

The following is the syntax for running Parallel Environment (PE) programs from the command line:

```
program [ arguments ] [ pe_arguments ]
```

You can also use the **poe** command to run programs as follows:

```
poe program [ arguments ] [ pe_arguments ]
```

If, however, you start MemoryScape on a PE application, you must start **poe** as MemoryScape's target using the following syntax:

```
memscape poe -a program [ arguments ] [ PE_arguments ]
```

For example:

```
memscape poe -a sendrecv 500 -rmpool 1
```

You should start all of your parallel tasks using the **Run** command.

Attaching to a PE Job

To take full advantage of MemoryScape's **poe**-specific automation, you need to attach to **poe** itself, and let MemoryScape automatically acquire the **poe** processes on all its nodes. In this way, MemoryScape acquires the processes you want to debug.

Attaching from a Node Running poe

To attach MemoryScape to **poe** from the node running **poe**:

1. Start MemoryScape in the directory of the debug target.
If you can't start MemoryScape in the debug target directory, you can start MemoryScape by editing the Debugger Server (**tvdsvr**) command line before attaching to **poe**.
2. In the **Home | Add Program** screen, select **Attach to running program**, then find the **poe** process list, select it, and hit the Next button.
When necessary, MemoryScape launches **tvdsvr** processes.
3. Locate the process you want to debug and dive on it.

If your source code files are not displayed in the Source Pane, you might not have told MemoryScape where these files reside. You can fix this by invoking the **File > Search Path** command to add directories to your search path.

Attaching from a Node Not Running poe

The procedure for attaching MemoryScape to **poe** from a node that is not running **poe** is essentially the same as the procedure for attaching from a node that is running **poe**.

To place **poe** in this list:

1. Connect MemoryScape to the startup node.
2. From the Home | Add Programs to Your MemoryScape Session page, select **Attach to running program**.
3. Look for the process named **poe** and continue as if attaching from a node that is running **poe**.

Debugging LAM/MPI Applications

NOTE >> In many cases, you can bypass the procedure described in this section. For more information, see [“Debugging MPI Programs”](#) on page 127.

You debug a LAM/MPI program in a similar way to how you debug most MPI programs. Use the following syntax if MemoryScape is in your path:

```
mpirun -tv mpirun args prog prog_args
```

As an alternative, you can invoke MemoryScape on **mpirun**:

```
memscape mpirun -a prog prog_args
```

Debugging QSW RMS Applications

NOTE >> In many cases, you can bypass the procedure described in this section. For more information, see “[Debugging MPI Programs](#)” on page 127.

Starting MemoryScape on an RMS Job

To start a parallel job under MemoryScape’s control, use MemoryScape as if you were debugging **prun**:

```
memscape prun -a prun-command-line
```

MemoryScape starts and shows you the machine code for RMS **prun**. Since you’re not usually interested in debugging this code, use the **Run** command to let the program run.

The RMS **prun** command executes and starts all MPI processes.

Attaching to an RMS Job

To attach to a running RMS job, attach to the RMS **prun** process that started the job.

You attach to the **prun** process the same way you attach to other processes.

Debugging Sun MPI Applications

NOTE >> In many cases, you can bypass the procedure described in this section. For more information, see “[Debugging MPI Programs](#)” on page 127.

MemoryScape can debug a Sun MPI program and display Sun MPI message queues. This section describes how to perform *job startup* and *job attach* operations.

To start a Sun MPI application, use the following command:

```
memscape mprun [ totalview_args ] -a [ mpi_args ]
```

For example:

```
memscape mprun -g blue -a -np 4 /usr/bin/mpl/conn.x
```

When the MemoryScape Window appears, select the **Go** button.

Attaching to a Sun MPI Job

To attach to an already running **mprun** job:

1. Find the host name and process identifier (PID) of the **mprun** job by typing **mpps -b**. For more information, see the **mpps(1M)** manual page.

The following is sample output from this command:

JOBNAME	MPRUN_PID	MPRUN_HOST
cre.99	12345	hpc-u2-9
cre.100	12601	hpc-u2-8

2. Go to **Attach to Running Program** and look for the process.

3. If MemoryScape is running on a different node than the **mprun** job, enter the host name in the **Remote Host** field.

NOTE >> You must link the MemoryScape agent into your Sun MPI Parallel Environment program before running it. For details, see “[Linking Your Application with the Agent](#)” on page 136.

Linking Your Application with the Agent

MemoryScape puts its heap agent between your program and its heap library which allows the agent to intercept the calls that your program makes to this library. After it intercepts the call, it checks the call for errors and then sends it on to the library so that it can be processed. The MemoryScape agent does not replace standard memory functions; it just monitors what they do. For more information, see “[Behind the Scenes](#)” on page 8.

In most cases, MemoryScape arranges for the heap agent to be loaded automatically when it starts your program. In some cases, however, special steps must be taken to ensure the agent loads. One example is when you are starting an MPI program using a launcher that does not propagate environment variables. (If you start your MPI program in MemoryScape using the **Add Parallel Program** page, MemoryScape propagates the information for you.) Another is when you want to start your program outside, or independently of, TotalView, and want to attach to the program later after it has started.

There are two ways you can arrange for the heap agent to be loaded:

- Link the application with the agent, as described in this section.

- Request that the heap agent be preloaded by setting the runtime loader's preloading environment variable. See “[Using env to Insert the Agent](#)” on page 139.

Here is some important platform-specific information:

- On AIX, the `malloc` replacement code and heap agent application must be in directories searched by the dynamic loader. If they are not in any of the standard directories (you can check with your system administrator), you can set `LIBPATH` to search these directories when you run the program. Another option is to add the directories to the program's list of search directories when you link the program. To do this, use the `-L` option as described in the table below. If you are in doubt about the directories being searched, you can obtain a list of the searched directories with `dump -Hv <program-name>`.

For additional requirements with AIX, see “[Installing tvheap_mr.a on AIX](#)” on page 140.

- On Cray, TotalView supports both static and dynamic linking. See the table below for the link lines you need to use.
- On Apple Mac OSX, you cannot link the agent into your program.

The following table lists additional command-line linker options that you must use when you link your program:

Platform	Compiler	Binary Interface	Additional linker options
Cray XT, XE, XK CLE (dynamic)	-	64	<code>-dynamic -L<path> -ltvheap_64 -Wl,-rpath,<path></code>
Cray XT, XE, XK CLE (static)	-	64	<code>-L<path> -ltvheap_cnl</code>
IBM RS/6000 (all)	IBM/GCC	32/64	<code>-L<path_mr> -L<path></code>
AIX 5	IBM/GCC	32	<code>-L<path_mr> -L<path> <path>/aix_malloctype.o</code>

Platform	Compiler	Binary Interface	Additional linker options
Linux x86-64 ¹	GCC/Intel/PGI	64	<code>-L<path_mr> -L<path> <path>/aix_malloctype64_5.o</code>
		32	<code>-L<path> -ltvheap -Wl,-rpath,<path></code>
Linux PowerLE	GCC	64	<code>-L<path> -ltvheap_64 -Wl,-rpath,<path></code>
		64	<code>-L<path> -ltvheap_64 -Wl,-rpath,<path></code>
Linux ARM64	GCC	64	<code>-L<path> -ltvheap_64 -Wl,-rpath,<path></code>
Sun	Sun/Apogee	32	<code>-L<path> -ltvheap -R <path></code>
	Sun	64	<code>-L<path> -ltvheap_64 -R <path></code>
	GCC	32	<code>-L<path> -ltvheap -Wl,-R,<path></code>
		64	<code>-L<path> -ltvheap_64 -Wl,-R,<path></code>

- ¹ On Ubuntu platforms, if the link line fails to start MemoryScape, try adding the additional flag `-Wl,-no-as-needed`. This flag should occur before the linking of `tvheap`, so on 64-bit platforms the link line would be:
- `-L<path> -Wl,-no-as-needed -ltvheap_64 -Wl,-rpath,<path>`

The following list describes the options in this table:

<path>

The absolute path to the agent in the MemoryScape installation hierarchy. More precisely, this directory is:

```
<installdir>/toolworks/memoryscape.<version>/  
<platform>/lib
```

<installdir>

The installation base directory name.

<version>

The MemoryScape version number.

<platform>

The platform tag.

<path_mr>

The absolute path of the malloc replacement library. This value is determined by the person who installs the MemoryScape malloc replacement library.

NOTE >> The heap agent library path can be hard to determine. If you have access to the command line interface (CLI), you can use the following command to print out its path:

```
puts $TV::hia_local_dir
```

Using env to Insert the Agent

When MemoryScape attaches to a process that is already running, the agent must already be associated with it. You can do this in two ways:

- Manually link the agent as described in previous sections.
- Start the program using **env** (see [man env](#) on your system). This pushes the agent into your program.

NOTE >>Preloading cannot be used with Cray. For information on preloading with Cray, see the earlier section [Linking Your Application with the Agent](#).

The variables required by each platform are shown in the following table. The placeholder **<hia_dir>** represents the directory in which the agent is found. See the previous section for how to determine this location.

Platform	Variable
Apple Mac OS X	DYLD_INSERT_LIBRARIES=<hia_dir>/libtvheap.dylib Note: See “ Mac OS ” on page 143 for detail on how this environment variable works.
IBM AIX	MALLOCTYPE=user:tvheap_mr.a If you are already using MALLOCTYPE for another purpose, reassign its value to the variable TVHEAP_MALLOCTYPE and assign MALLOCTYPE as above; when the agent starts it will correctly pass on the options.
Linux	
32-bit	LD_PRELOAD=<hia_dir>/libtvheap.so
64-bit	LD_PRELOAD=<hia_dir>/libtvheap_64.so
Sun	
32-bit generic	LD_PRELOAD=<hia_dir>/libtvheap.so
32-bit specific	LD_PRELOAD_32=<hia_dir>/libtvheap.so
64-bit generic	LD_PRELOAD=<hia_dir>/libtvheap_64.so
64-bit specific	LD_PRELOAD_64=<hia_dir>/libtvheap_64.so If the agent is the only library you are preloading, use the generic variable. Otherwise, use whichever variable was used for the other preloaded libraries.

Installing tvheap_mr.a on AIX

NOTE >> Installing tvheap_mr.a on AIX requires that the system have the bos.adt.syscalls System Calls Application Toolkit page installed.

You must install the **tvheap_mr.a** library on each node on which you plan to run the MemoryScape agent. The **aix_install_tvheap_mr.sh** script contains most of the required setup, and is located in this directory:

toolworks/totalview.version/rs6000/lib/

For example, after you become **root**, enter the following commands:

```
cd toolworks/memscope.1.0.0-0/rs6000/lib
mkdir /usr/local/tvheap_mr \
./aix_install_tvheap_mr.sh ./tvheap_mr.tar \
/usr/local/tvheap_mr
```

Use **poe** to create **tvheap_mr.a** on multiple nodes.

The pathname for the **tvheap_mr.a** library must be the same on each node. This means that you cannot install this library on a shared file system. Instead, you must install it on a file system that is private to the node. For example, because **/usr/local** is usually accessible only from the node on which it is installed, you might want to install it there.

NOTE >> The tvheap_mr.a library depends heavily on the exact version of libc.a that is installed on a node. If libc.a changes, you must recreate tvheap_mr.a by re-executing the aix_install_tvheap_mr.sh script.

If this malloc replacement library changes (which is infrequent) you'll need to rerun this procedure. Any change will be noted among a release's new features.

LIBPATH and Linking

This section discusses compiling and linking your AIX programs. The following command adds **path_mr** and **path** to your program's libpath:

```
xlc -Lpath_mr -Lpath -o a.out foo.o
```

When **malloc()** dynamically loads **tvheap_mr.a**, it should find the library in **path_mr**. When **tvheap_mr.a** dynamically loads **tvheap.a**, it should find it in **path**.

The AIX linker supports relinking executables. This means that you can make an already complete application ready for the MemoryScape agent; for example:

```
cc a.out -Lpath_mr -Lpath -o a.out.new
```

Here's an example that does not link in the heap replacement library. Instead, it allows you to dynamically set **MALLOCTYPE**:

```
xlc -q32 -g \
-L/usr/local/tvheap_mr \
-L/home/memscope/interposition/lib prog.o -o prog
```

This next example shows how a program can be set up to access the MemoryScape agent by linking in the **aix_malloctype.o** module:

```
xlc -q32 -g \
-L/usr/local/tvheap_mr \
-L/home/memscope/interposition/lib prog.o \
/home/memscope/interposition/lib/aix_malloctype.o \
-o prog
```

You can check that the paths made it into the executable by running the **dump** command; for example:

```
% dump -Xany -Hv tx_memdebug_hello

tx_memdebug_hello:

***Loader Section***
      Loader Header Information
VERSION#    #SYMtableENT    #RELOCent    LENidSTR
0x00000001  0x0000001f    0x00000040    0x000000d3

#IMPfilID   OFFidSTR        LENstrTBL    OFFstrTBL
0x00000005  0x00000068    0x00000080    0x000000db

***Import File Strings***
INDEX  PATH                BASE                MEMBER
0      /.../lib:/usr/.../lib:/usr/lib:/lib
1      libC.a              shr.o
2      libC.a              shr.o
3      libpthreads.a       shr_comm.o
4      libpthreads.a       shr_xpg5.o
```

Index 0 in the **Import File Strings** section shows the search path the run-time loader uses when it dynamically loads a library. Some systems propagate the preload library environment to the processes they will run; others, do not. If they do not, you need to manually link them with the **tvheap** library.

In some circumstances, you might want to link your program instead of setting the **MALLOCTYPE** environment variable. If you set the **MALLOCTYPE** environment variable for your program and it uses **fork()/exec()** a program that is not linked with the agent, your program will terminate because it fails to find **malloc()**.

Using MemoryScape in Selected Environments

This topic describes using the Memory Debugger within various environments. The sections within this topic are:

- [MPICH](#)
- [IBM PE](#)
- [RMS MPI](#)
- [Mac OS](#)
- [Linux](#)

MPICH

Here's how to use MemoryScape with MPICH MPI codes. Rogue Wave Software has tested this only on Linux x86-64.

1. You must link your parallel application with the MemoryScape agent as described "[LIBPATH and Linking](#)" on page 140. On most Linux x86-64 systems, you'll type:

```
mpicc -g test.o -o test -Lpath \
-ltvheap -Wl,-rpath,path
```

2. Start MemoryScape using the **-tv** command-line option to the **mpirun** script in the usual way. For example:

```
mpirun -tv mpirun-args test args
```

MemoryScape will start up on the rank 0 process.

3. If you need to configure MemoryScape, you should do it now.
4. Run the rank 0 process.

IBM PE

Here's how to use MemoryScape with IBM PE MPI codes:

1. You must prepare your parallel application to use the MemoryScape agent in "[LIBPATH and Linking](#)" on page 140 and in "[Installing tvheap_mr.a on AIX](#)" on page 140. Here is an example that usually works:

```
mpcc_r -g test.o -o test -Lpath_mr -Lpath \
path/aix_malloctype.o
```

"[Installing tvheap_mr.a on AIX](#)" on page 140 contains additional information.

2. Start MemoryScape on **poe** as usual:

```
memscape poe -a test args
```

Because **tvheap_mr.a** is not in **poe**'s LIBPATH, enabling MemoryScape upon the **poe** process will cause problems because **poe** will not be able to locate the **tvheap_mr.a** malloc replacement library.

3. If you need to configure MemoryScape, you should do it now.
4. Run the **poe** process.

RMS MPI

Here's how to use MemoryScape with Quadrics RMS MPI codes. Rogue Wave Software has tested this only on Linux x86-64.

1. There is no need to link the application with MemoryScape because **prun** propagates environment variables to the rank processes. However, if you'd like to link the application with the agent, you can.
2. Start MemoryScape on **prun**. For example:

```
memscape prun -a prun-args test args
```
3. If you need to configure MemoryScape, you should do it now.
4. Run the **prun** process.

Mac OS

In most circumstances, memory debugging works seamlessly on the Mac OS.

From 10.11 El Capitan and onwards, however, the Mac OS introduced some changes that can affect some programs when memory debugging. While these should not affect how your program runs, in some rare cases you may want to fine-tune how the HIA behaves.

Background

In the Mac OS environment, interposition works only for preloaded DLLs, meaning that the Heap Interposition Agent (HIA) can only be preloaded rather than linked with the target as in some other operating systems. (See [“Behind the Scenes”](#) on page 8 for more information on interposition and the HIA.)

The HIA makes sure that any environment variables related to preloading are correctly propagated if your program calls **execve()** or **system()**. The required Mac OS environment variable is **DYLD_INSERT_LIBRARIES**.

For all Mac OS releases from El Capitan onwards, however, a new feature System Integrity Protection (SIP) implemented a protocol that disallows passing **DYLD_INSERT_LIBRARIES** to a protected program or a program that resides in a protected directory. Calls to **system()** are affected because it is defined as invoking **/bin/sh**, which is in a SIP-protected directory.

Calls to system() on Mac OS

To work around the Mac OS SIP feature, for every **system()** call, the HIA copies **bin/sh** to a temporary directory (in **/tmp**) and arranges for the copy to be used so that **DYLD_INSERT_LIBRARIES** is not filtered out during the call. Once the child process has completed, the parent deletes the temporary directory.

This is the default behavior. To modify this, enable the environment variable **TV_MACOS_SYSTEM**.

Setting the Environment Variable TV_MACOS_SYSTEM

The **TV_MACOS_SYSTEM** environment variable allows customization of memory debugging behavior for Mac OS programs that call **system()**, and includes the following options:

pass_through=boolean

If **true**, the call to **system ()** is passed through to the underlying implementation.

The default is **false**, in which case the HIA controls the call to **system()** as described in [“Calls to system\(\) on Mac OS”](#) on page 143. Setting this option to **true** may be useful if you have disabled SIP.

Example: **"TV_MACOS_SYSTEM=pass_through=true"**

shell=<pathname>

Defines the shell for the HIA to use instead of the default **bin/sh**.

If defined, be sure that the SIP does not control access to this shell so that the HIA has access to it. Note that the named shell is not deleted after the return from **system()**.

This setting may be useful to avoid any potential performance issues caused by copying the shell for each **system()** call, and then deleting it later.

Be aware, however, that using a previously stashed copy of **bin/sh** may require some maintenance, since the copy will not be updated when the operating system is updated.

The pathname must not contain commas or whitespace characters.

Example: "TV_MACOS_SYSTEM=shell=/path/to/some/copy/of/bin/sh"

tmpdir=<pathname>

Defines a temporary directory where the HIA will copy the shell (given the default setting of the option **pass_through=false**).

The HIA does not create the directory, assuming that it exists already. The HIA deletes the copy of the shell after processing is complete, but does not remove the directory.

If not set, the HIA creates a temporary directory in **/tmp**, which it removes after the call to **system ()** completes.

Example: "TV_MACOS_SYSTEM=tmpdir=/path/to/some/directory"

Linux

dlopen and RTLD_DEEPBIND

In most circumstances, memory debugging works seamlessly with programs that call **dlopen** to dynamically load shared objects (DSOs).

However, the Linux implementation of **dlopen** accepts **RTLD_DEEPBIND** in the **flags/mode** argument. **RTLD_DEEPBIND** affects how undefined references in a DSO are bound. By default, when **RTLD_DEEPBIND** is not set, the dynamic linker first looks up any symbols needed by a newly-loaded DSO in the global scope.

RTLD_DEEPBIND modifies this behavior. When set, the dynamic linker places the lookup scope of the DSO ahead of the global scope. This means that the dynamic linker seeks to bind any undefined references in the DSO to definitions in the DSO, or any of the DSOs on which it depends. Only after these have been searched and a symbol not found is the global scope examined.

RTLD_DEEPBIND can affect memory debugging because, in some circumstances, references to all or part of the heap manager interface in a DSO can become bound to definitions in the standard library directly, rather than to those in the HIA. As a result, the HIA may not see all the traffic between the program and heap manager. If this occurs, the information the HIA is able to collect for TotalView will be incomplete, reducing its usefulness. In some circumstances, memory debugging may even fail.

How the HIA Handles RTLD_DEEPBIND

The HIA deals with the challenges posed by **RTLD_DEEPBIND** by intercepting calls to **dlopen**. If the program specifies **RTLD_DEEPBIND**, the HIA inserts itself as one of the to-be-loaded DSO's dependents. It does this by creating a

new ELF wrapper file that lists the HIA and the DSO the program wants to **dlopen** as needed files. Instead of opening the DSO the program named, the HIA **dlopens** the new wrapper DSO it constructed. Since the DSO given by the program code is listed as a needed file, it too is opened.

As far as the program is concerned, the **dlopen** behaves as it would in the absence of the HIA. After the call to **dlopen**, the HIA cleans up and deletes the wrapper DSO that it created.

Modifying How the HIA Handles RTLD_DEEPBIND

The basic behavior described in [How the HIA Handles RTLD_DEEPBIND](#) can be modified by setting the **TVHEAP_DEEPBIND** environment variable. The following comma-separated settings are supported options:

pass_through=boolean

If **true**, the HIA does no special processing to handle **RTLD_DEEPBIND**. It does not create the ELF wrapper, and instead passes the operation through to the standard **dlopen**.

The default is **false**, in which case the HIA takes the steps described in [How the HIA Handles RTLD_DEEPBIND](#).

Example: "TVHEAP_DEEPBIND=pass_through=true"

keep_wrapper=boolean

If **true**, the HIA does not delete the ELF wrapper it creates after it has been used. The default is **false**, in which case the HIA deletes the ELF wrapper after it **dlopens** it.

Example: "TVHEAP_DEEPBIND=keep_wrapper=true"

tmpdir=<directory_name>

If defined, the HIA creates the ELF wrapper it generates in the directory specified by the **tmpdir** setting. The default is to use the setting of the environment variable **TMPDIR**. If **TMPDIR** is not defined, the ELF wrapper is created in **/tmp**.

MemoryScape Scripting

You can obtain information from MemoryScape by executing it in batch mode using the **memscript** command. Batch mode requires the use of command-line options, like so:

memscript *command_line_options*

display_specifiers Command-Line Option

The **-display_specifiers** command-line option controls how MemoryScape writes information to the log file.

-display_specifiers "*list_item*"

Specifies one or more items that can be added or excluded from the log file. Separate items with a comma.

list_item values are described in the following table. The word **no** in front of item suppresses its display.

Item	Controls display of ...
[no]show_allocator	The allocator for the address space
[no]show_backtrace	The backtrace for memory blocks
[no]show_backtrace_id	The backtrace ID for memory blocks

Item	Controls display of ...
[no]show_block_address	The start and end addresses for a memory block
[no]show_flags	Memory block flags
[no]show_guard_id	The guard ID for memory blocks
[no]show_guard_settings	The guard settings for memory blocks
[no]show_image	The process/library associated with a backtrace frame
[no]show_owner	The owner of the allocation
[no]show_pc	The backtrace frame PC
[no]show_pid	The process PID
[no]show_red_zones_settings	The Red Zone entries for allocations and deallocations in the entire address space

event_action Command-Line Option

The **-event_action** command-line option is the most complex of the command line options. Its format is as follows:

-event_action "*event=action list*"

Specifies one or more actions that the script should perform if an event occurs. The "*event=action list*" consists of comma-separated set *event=action* pairs. For example:

```
"alloc_null=save_memory_debugging_file, \
dealloc_notification=list_allocations"
```

event can be:

Event	Description
addr_not_at_start	A block is being freed, and the address is not at the beginning of the block.
alloc_not_in_heap	The block being freed is not in the heap.
alloc_null	The malloc() function returned a null block.
alloc_returned_bad_alignment	The block is misaligned.
any_memory_event	All memory notification events.
bad_alignment_argument	The block returned by the malloc library is not aligned on a byte boundary required by your operating system. The heap may be corrupted. (This is not a program error.)

Event	Description
double_alloc	Allocator returned a block already in use. The heap may be corrupted.
double_dealloc	Program is attempting to free a block already freed.
free_not_allocated	Program is attempting to free a block that was not allocated.
guard_corruption	Guard corruption was detected when program deallocated a block.
hoard_low_memory_threshold	Hoard low memory threshold is crossed.
realloc_not_allocated	Program attempted to reallocate a block that was not allocated.
rz_overflow	Program attempted to access memory beyond end of allocated block.
rz_underrun	Program attempted to access memory before start of allocated block.
rz_use_after_free	Program attempted to access block after it was deallocated.
rz_use_after_free_overflow	Program attempted to access memory beyond end of deallocated block.
rz_use_after_free_underrun	Program attempted to access memory before start of deallocated block.
termination_notification	Program is about to execute its _exit routine.

action is as follows:

Action	Description
check_guard_blocks	Check for guard blocks and generate a corruption list.
list_allocations	Create a list of all your program's allocations.
list_leaks	Create a list of all of your program's leaks.
save_html_heap_status_source_view	Save the Heap Status Source report as an HTML file.
save_memory_debugging_file	Save a memory debugging file; you can reload this file at a later time.
save_text_heap_status_source_view	Save the Heap Status Source report as a text file.

Other Command Line Options

The **memscript** command takes these additional options.

-guard_blocks

Turn on guard blocks.

-red_zones_overruns

Turn on testing for Red Zone overruns.

-red_zones_underruns

Turn on testing for Red Zone underruns.

-detect_use_after_free

Turn on testing for use after memory is freed.

-hoard_freed_memory

Turn on the hoarding of freed memory.

-hoard_low_memory_threshold *nnnn*

Specify the low memory threshold that will generate an event.

-detect_leaks

Turn on leak detection.

-red_zones_size_ranges *min:max,min:max,...*

Specify the memory allocation ranges for which Red Zones are in effect.

Ranges can be in the following formats:

x:y allocations from x to y

:y allocations from 1 to y

x: allocations of x and higher

x allocation of x

-maxruntime *hh:mm:ss*

Specify the maximum amount of time the script should run where:

hh: number hours

mm: number of minutes

ss: number of seconds

As a script begins running, MemoryScope adds information to the beginning of the log file. This information includes time stamps for both the file when processes start, the name of the program, and so on.

memscript Example

The example here performs these actions:

- Runs the **filterapp** program under MemoryScope control.
- Passes an argument of **2** to the **filterapp** program.
- Whenever any event occurs—an HIA event, SEGV, and the like—saves a memory debugging file.
- Allows the script to run for no longer than 5 seconds.
- Performs the following activities: use guard blocks, hoard freed memory, and detect memory leaks.

```
memscript -maxruntime "00:00:05" \
  -event_action "any_event=save_memory_debugging_file" \
  -guard_blocks -hoard_freed_memory -detect_leaks \
  ~/Work/filterapp -a 2
```

MemoryScape Command-Line Options

This chapter presents the commands used to invoke MemoryScape as well as the variables you can place in a **.memrc** file.

Invoking MemoryScape

Use the **memscape** command to invoke the MemoryScape GUI and the **memscript** command to invoke MemoryScape in batch mode.

Topics in this section are:

- “Syntax” on page 150
- “Options” on page 150

Syntax

{ **memscript** | **memscape** } [*filename* [*corefile*]] [*options*]

Arguments

filename

Specifies the pathname of the executable being debugged. This can be an absolute or relative pathname. The executable must be compiled with

debugging symbols turned on, normally the **-g** compiler option. Any multiprocess programs that call **fork()**, **vfork()**, or **execve()** should be linked with the **dbfork** library.

corefile

Specifies the name of a core file. Use this argument in addition to *filename* when you want to examine a core file with MemoryScape.

If you specify mutually exclusive options on the same command line (for example, **-ccq** and **-nccq**), MemoryScape uses the last option that you enter.

Options

-a *args*

Passes all subsequent arguments (specified by *args*) to the program specified by *filename*. This option must be the last on the command line.

-aix_use_fast_trap

Specifies use of the AIX fast trap mechanism. You must either set this option on the command line or place it within a **.memrc** file.

-bg *color*

Same as **-background**.

-compiler_vars

Some Fortran compilers (HP f90/f77, HP f90, SGI 7.2 compilers) output debugging information that describes variables the compiler itself has in-

vented for purposes such as passing the length of character*(*) variables. By default, MemoryScape suppresses the display of these compiler-generated variables.

However, you can specify the **-compiler_vars** option to display these variables. This is useful when you are looking for a corruption of a run-time descriptor or are writing a compiler.

-no_compiler_vars

(Default) Does not show variables created by the Fortran compiler.

-control_c_quick_shutdown

-ccq (Default) Kills attached processes and exits.

-no_control_c_quick_shutdown

-nccq

Invokes code that sometimes allows MemoryScape to better manage the way it kills parallel jobs when it works with management systems. This has only been tested with SLURM. It may not work with other systems.

-debug_file consoleoutputfile

Redirects MemoryScape console output to a file named *consoleoutputfile*.

Default: All MemoryScape console output is written to **stderr**.

-display displayname

Sets the name of the X Windows display to *displayname*. For example, **-display vinnie:0.0** displays MemoryScape on the machine named "vinnie."

Default: The value of your **DISPLAY** environment variable.

-dump_core

Dumps a MemoryScape core file when an internal error occurs. This is used to help Perforce Software debug MemoryScape problems.

-no_dumpcore

(Default) Does not dump a core file when it gets an internal error.

-env variable=value

Adds an environment variable to the environment variables passed to your program by the shell. If the variable already exists, this option replaces the previous value. You need to use this command for each variable being added; that is, you cannot add more than one variable with an **env** command.

-nptl_threads

Specifies the use of NPTL threads by your application. You need to use this option only if MemoryScape cannot determine which thread package your program is using.

-no_nptl_threads

Specifies that your application is *not* using the NPTL threads package. Use this option only if MemoryScape thinks your application is using it when it is not.

-pid pid filename

Attaches to process *pid* for executable *filename* after TotalView starts executing.

-search_path pathlist

Specifies a colon-separated list of directories to search for source files. For example:

memscape -search_path proj/bin:proj/util

-signal_handling_mode "action_list"

Modifies the way in which MemoryScape handles signals. You must enclose the *action_list* string in quotation marks to protect it from the shell.

An *action_list* consists of a list of *signal_action* descriptions separated by spaces:

signal_action [*signal_action*] ...

A signal action description consists of an action, an equal sign (=), and a list of signals:

action=signal_list

An *action* can be one of the following: **Error**, **Stop**, **Resend**, or **Discard**.

A *signal_specifier* can be a signal name (such as **SIGSEGV**), a signal number (such as 11), or a star (*), which specifies all signals. We recommend that you use the signal name rather than the number because number assignments vary across UNIX sessions.

The following rules apply when you are specifying an *action_list*:

- If you specify an action for a signal in an *action_list*, MemoryScape changes the default action for that signal.
- If you do not specify a signal in the *action_list*, MemoryScape does not change its default action for the signal.
- If you specify a signal that does not exist for the platform, MemoryScape ignores it.
- If you specify an action for a signal more than once, MemoryScape uses the last action specified.

If you need to revert the settings for signal handling to MemoryScape's built-in defaults, use the **Defaults** button in the **File > Signals** dialog box.

For example, here's how to set the default action for the **SIGTERM** signal to resend:

```
"Resend=SIGTERM"
```

Here's how to set the action for **SIGSEGV** and **SIGBUS** to error, the action for **SIGHUP** to resend, and all remaining signals to stop:

```
"Stop=* Error=SIGSEGV,SIGBUS \
  Resend=SIGHUP"
```

-shm "action_list"

Same as **-signal_handling_mode**.

-stderr pathname

Names the file to which MemoryScape writes the target program's **stderr** information while executing within MemoryScape. If the file exists, MemoryScape overwrites it. If the file does not exist, MemoryScape creates it.

-stderr_append

Tells MemoryScape to append the target program's **stderr** information to the file named in the **-stderr** command or specified in the GUI. If the file does not exist, MemoryScape creates it.

-stderr_is_stdout

Redirects the target program's **stderr** to **stdout**.

-stdin pathname

Names the file from which the target program reads information while executing within MemoryScape.

-stdout pathname

Names the file to which MemoryScape writes the target program's **stdout** information while executing within MemoryScape. If the file exists, MemoryScape overwrites it. If the file does not exist, MemoryScape creates it.

-stdout_append

Tells MemoryScape to append the target program's **stdout** information to the file named in the **-stdout** command or specified in the GUI. If the file does not exist, MemoryScape creates it.

-verbosity level

Sets the verbosity level of MemoryScape-generated messages to *level*, which can be one of **silent**, **error**, **warning**, or **info**.

Default: **info**

Index

Numerics

0xa110ca7f allocation pattern 38, 54
0xdea110cf deallocation pattern 38, 54

A

-a option 125
-a option to memscape command 150
Add Filter Dialog Box 104
Add Filter dialog box 104
Add memory debugging file command 19, 111
Add parallel program option 69
Add parallel program screen 127
Add Program screen 111
Add Programs to Your MemoryScape Session page 65
Add Programs to Your MemoryScape Session screen 127, 130
adding command-line arguments 65
adding core files 66
adding environment variables 65
adding files 65
adding parallel programs 69
adding programs and files 16
adding remote programs 65
Address not at start of block problems 13
address space 4
addr_not_at_start event 147

advanced memory debugging options 74
Advanced Options button 75
agent linking 136
agent, inseting with env 139
agent. See heap debugging.
agent's shared library 8
AIX
 compiling 64-bit code 123
 linking C++ to dbfork library 123
 linking to dbfork library 123
aix_install_tvheap_mr.sh script 140
aix_use_fast_trap command-line option 150
-aix_use_fast_trap option 150
allocation
 0xa110ca7f pattern 54
 adding guards 77
 block painting 2
 information 98
allocation focus 106
Allocation Location tab 23
allocation pattern 38
alloc_not_in_heap event 147
alloc_null event 147
alloc_returned_bad_alignment event 147
any_event event 147
arguments, remembering 128

Attach to running program screen 130
attaching to MPICH job 130
attaching to mprun job 134
attaching to PE jobs 133
attaching to programs 139
attaching to programss 126
attaching to running program 66
attaching to running program, limitation 66, 126
attaching to Sun MPI job 134
autolaunching 126
automatic variables 10
automatically halting program 85

B

backtrace
 deallocation 23
 recording 92
 reports 101
 which displayed 26
backtrace ID 98
backtrace ID, defined 101
-backtrace option 33
backtrace reports 97
backtrace_depth TV_HEAP_ARGS value 43
backtraces 22, 26, 33
 setting depth 33

- setting trim 33
- backtrace_trim TV_HEAP_ARGS value 43
- bad_alignment_ argument event 147
- Basic Options button 75
- bg command-line option 150
- bg option 150
- bit painting
 - 0xa110ca7f 54
 - 0xdea110cf 54
- bit pattern
 - writing 28
- bit pattern, writing 2
- bkeepfile option 123
- Block Allocation tab 92
- block color coding 96
- Block Deallocation tab 92
- Block Details tab 23, 92
- block display 94
- block highlighting 96
- block information 46, 96
- block painting 2, 14, 17, 28
 - changing pattern 54
 - defined 2
- Block Properties window 109
- Block Properties window. 90
- block tagging 28
- blocks, allocating guards 77
- bss data error 23
- byte display 110

C

- C++
 - including libdbfork.h 123

- ccq command-line option 151
- changing filter order 104
- changing guard block patterns 78
- changing guard block size 78
- chart report 7, 87
- check_guard_blocks action 148
- checking for problems 2, 61
- checking guard blocks 33
- check_interior option 37
- color coding of blocks 96
- Command line arguments area 128
- command-line arguments, entering 65
- command-line arguments, setting 61
- command-line options
 - aix_use_fast_trap 150
 - bg 150
 - ccq 151
 - control_c_quick_shutdown 151
 - debug_file 151
 - display 151
 - dump_core 151
 - env 151
 - nccq 151
 - no_compiler_vars 151
 - no_control_c_quick_shutdown 151
 - no_dumpcore 151
 - no_nptl_threads 151
 - nptl_threads 151
 - pid 151
 - search_path 151
 - shm 152
 - signal_handling_mode 151
 - stderr 152

- stderr_append 152
- stderr_is_stdout 152
- stdin 152
- stdout 152
- stdout_append 152
- verbosity 152
- commands
 - dheap -guards 33
- comparing memory states 70
- Compare Memory Report 111
- comparing memory states 19, 113
- comparing state information 111
- comparisons
 - process 115
- compiling 64-bit code on AIX 123
- compiling programs 122
- concealed allocation 15
- Configuration page 28
- configuring 64
- console output redirection 151
- context menu, controlling processes 85
- control_c_quick_shutdown command-line option 151
- controlling execution 85
- controlling processes
 - from context menu 85
 - individually 85
- controlling program execution task 82
- core
 - dumping for MemoryScape 151
- core files
 - writing 77
- core files, adding 66
- core files, starting within MemoryScape 125

- corrupt memory 17
 - notification 108
- Corrupted Guard Blocks Report 51
- Corrupted Memory report 67, 108
- criteria
 - changing order 105
 - filters 105
 - operators 106
 - property 106
 - removing 105
- Current Processes area 94
- custody changes 15

D

- dangling interior pointer 36
- dangling pointer
 - problems 28
- dangling pointers 12, 36
 - example 29
- dangling pointers and leaks
 - compared 12
- data section 5
- data segment memory 86
- Data Source area 114
- Data Source controls 98
- dbfork and Linux for Mac OS X 124
- dbfork library
 - linking with 123
- dbg files 77
- deallocate, defined 9
- deallocated memory, hoarding 118
- deallocated memory, retaining 56
- deallocation

- 0xdea110cf pattern 54
- backtrace 23
- block painting 2
- notifications 92
- deallocation information 98
- deallocation pattern 38
- Deallocation tab 23
- debug_file command-line option 151
- debug_file option 151
- debugging command-line option 16
- debugging MPICH applications 129
- debugging options 17
 - Painting memory 80
- depth, backtraces 33
- Detailed program and library report 87
- Detect Leaks checkbox 98
- detect_leaks 149
- dheap
 - compare 32
 - disable 31
 - ed_zones 32
 - enable 31
 - example 32
 - export 35
 - export, data option 35
 - export, output option 35
 - export, set show_backtraces option 35
 - export, set_show_code option 35
 - filter 32, 36
 - filter, enable filtering 35
 - filter, enabling a filter 35
 - hoard 32
 - info 31

- leaks 32
- nonotify 32
- notify 32
- paint 32
 - status of Memory Tracker 31
- dheap -guards 33
- disabling all filters 104
- discarding memory information, when 17
- disk icon, memory comparisons 114
- display command-line option 151
- display option 151
- display_allocations_on_exit 43
- display_specifiers command-line option 146
- dlopen
 - and RTLD_DEEPBIND on Linux 144
- double_dealloc event 147
- dump_core command-line option 151
- dynamically allocate space 13

E

- Edit Filter Dialog Box 104
- editing filters 104
- Enable Filtering check box 98, 103
- Enable Filtering checkbox 95
- enable guard blocks option 67
- enabling all filters 104
- enabling painting 80
- entering command-line arguments 65
- entering environment variables 65
- env command-line option 151
- env, inserting agent 139
- environment variables
 - LD_LIBRARY_PATH 124

- environment variables, entering 65
- environment variables, TV_HEAP_ARGS 43
- Environment variables tab 128
- error
 - freeing stack memory 23
- error notification 90
- error notifications 76
- event backtrace 22
- event indicator 21, 76
- Event Location tab 22, 92
- event notification options 90
- event_action command-line option 147
- events 90
 - writing file when occurring 77
- examining memory 45
- executing on remote hosts 65
- execution controls 85
- execution, stopping 66
- execve() 123
- exit events 17
- _exit routine 75
- exit, notification option 75
- Export Memory Data command 19, 66, 111
- exporting memory data 66
- extending a block 13

F

- fifo hoard queue 34
- File > Export command 77
- files, libdbfork.h 123
- filter criteria 105
- Filter out this entry 103
- filter properties 104

- filtering 95, 98, 103
 - heap information 35
 - Heap Status Graphical Report 51
 - overview 51
 - reports 45
- filters
 - changing criteria order 105
 - criteria 106
 - criteria operators 106
 - criteria properties 106
 - disabling all 104
 - editing 104
 - enabling all 104
 - naming 105
 - ordering 104
 - removing 104
 - removing criteria 105
- filters, editing 104
- finding deallocation problems 13
- finding differences 114
- finding heap heap allocation problems 13
- finding memory leaks 25
- fork() 123
- Fortran, tracking memory 8
- free not allocated problems 13
- free problems 2, 32
 - finding 21
- free stack memory error 92
- freeing bss data error 23
- freeing data section memory error 23
- freeing freed memory 23
- freeing memory that is already freed error 23
- freeing stack memory error 23

- freeing the wrong address 24
- freeing the wrong address error 24
- free_not_allocated event 147
- function backtrace, backtrace

G

- g 16
- Generate a core file and abort the program option 76
- Generate a lightweight memory data file option 76
- graphical heap display 23
- Graphical report 45, 47
- Graphical View 46
- graphically viewing the heap 94
- Guard allocated memory option 77
- guard block display 110
- guard blocks 2, 17
 - altering 108
 - changing patterns 78
 - changing size 78
 - checking 33
 - checking for errors 51
 - defined 108
 - enabling 108
 - explained 51
 - manually checking for 78
 - maximum size 78
 - medium debugging level 78
 - notification 17, 33, 51
 - notifications 108
 - notify on deallocation problem 77
 - overwriting 17
 - patterns 108
- guard_blocks command-line option 149

guard_corruption event 147

H

Halt execution at process exit 17

Halt execution at process exit option 75

Halt execution on memory event or error option 75,
90

handling signals 151

header section 6

heap

defined 13

status 19

viewing graphically 94

heap allocation problems 13

heap API 13

stopping allocation when misused 13

heap API problems 32

heap debugging 21

attaching to programs 139

environment variable 139

freeing bss data 23

freeing data section memory error 23

freeing memory that is already freed error 23

freeing the wrong address 24

functions tracked 21

IBM PE 142

incorporating agent 136

interposition

defined 8

linker command-line options 136

MPICH 142

preloading 8

realloc problems 24

RMS MPI 142

tvheap_mr.a

library 140

using 21

heap displays, simplifying 35

heap information 97

filtering 35

Heap Information tab 96

heap information, saving 35

heap interposition agent 8

heap library functions 8

heap memory 86

Heap Source Backtrace report 101

Heap Status Backtrace report 102

Heap Status Graphical Report 67, 93, 94, 96

color coding key 46

zoom controls 94

Heap Status Graphical report 45, 47

Heap Status Graphical Report screen 94

Heap Status Graphical View 46

Heap Status reports 94

heap status reports 67

Heap Status Source Report 98

Heap Status Source report 102

Heap Usage Monitor 113

heap usage monitor 85

HIA, linking 18

hoard capacity 34

hoard information 98

hoard size 81

-hoard_freed_memory command-line option 149

hoarding 18, 28, 34, 56

block maximum 34

deallocated memory 118

defined 2

enabling 34

finding a multithreading problem 56

finding dangling pointer references 56

KB size 34

option 118

status 34

host names, specifying specifying host names 125

hosts, remote 65

HTML, saving reports as 117

I

identifying leaks within libraries 87

importing memory state 111

individually controlling processes 85

interposition 8, 90

interposition defined 8

-is_dangling option 36

L

LAM and MemoryScape 133

LD_PRELOAD heap debugging environment vari-
able 139

leak consolidation 37

leak detection 37

checking interior 37

Leak Detection report 27

Leak Detection reports 25, 67

Leak Source report 102

leaks

concealed ownership 15

custody changes 15

defined 2, 14

-leaks option 37

listing 2

- orphaned ownership 14
 - showing 98
 - underwritten destructors
 - leaks 15
 - why they occur 14
- leaks and dangling pointers
 - compared 12
- leaks reports 102
- leightweight memory file
 - writing 77
- lib directory 105
- libdbfork.a 123
- libdbfork.h file 123
- LIBPATH and linking 140
- libraries
 - leaks within 87
- library report 87
- linking 5
- linking agent 136
- linking MemoryScape 18
- linking to dbfork library 123
 - AIX 123
 - C++ and dbfork 123
 - SunOS 5 124
- list_allocations action 148
- listing leaks 2
- list_leaks action 148
- load file 5
- loading programs and files 16

M

- Mac OS X, linking dbfork 124
- machine code section 6
- magnifying glass controls 94

- MALLOCTYPE heap debugging environment variable 139, 140
- Manage Filters option 103
- Manage Process and Files screen 82, 85
- managing processes 85
- maxruntime command-line option 149
- memalign_strict_alignment_even_multiple
 - TV_HEAP_ARGS value 44
- memory
 - data segment 86
 - examining 45
 - heap 86
 - maps 4
 - pages 4
 - stack 86
 - text segment 86
 - total virtual memory 87
 - virtual stack 87
- memory block painting 17
- Memory Comparison report 114
- memory comparisons 70, 113
- Memory Comparisons report 67
- Memory Comparisons screen 114
- Memory Content tab 96
- memory contents tab 110
- memory corruption 2
- memory data, exporting 66
- Memory Debugger
 - functions tracked 8
 - using 16
- Memory Debugging Data Filters Dialog Box 104
- memory debugging file 112
 - indicator 112
- memory debugging options 16

- advanced 74
 - basic 71
 - Guard allocated memory 77
 - Halt execution at process exit 75
 - Halt execution on memory event or error 75
 - high 74
 - low 74
 - medium 74
 - screen 71
- Memory Debugging Options screen 74, 90
- memory error notification 17
- Memory Event Details Window
 - Point of Deallocation tab 23
- memory hoarding 18
- memory leak
 - defined 14
- memory painting 78, 80
- Memory Reports area 67
- memory state
 - importing 111
 - restoring 111
 - saving 111
- memory states
 - comparing 19
- memory states, comparing 19
- memory usage
 - seeing 86
- Memory Usage reports 7, 19, 67, 86
 - chart 87
 - Library 87
 - Process 87
- memory, painting 119
- memory, reusing 34

- MemoryScape
 - command options 150
 - linking 18
 - starting 16, 61
- MemoryScape and LAM 133
- MemoryScape and MPICH 129
- MemoryScape and PE 131
- MemoryScape and Sun MPI 134
- memscape command 61, 64, 129
- memscript command 64
- merging object files 5
- MmeoryScape command line 125
- MPI
 - on Sun 134
- MPI debugging 70
- MPI issues 131
- MPI launcher arguments area 128
- MPI programs, starting 69, 126
- MPICH
 - and heap debugging 142
- MPICH and MemoryScape 129
- MPICH and poe 129
- MPICH applications, debugging 129
- MPICH library, selecting 129
- MPICH P4 procgroup 130
- MPICH, attaching to job 130
- mpirun 125
- mprun 134
- mprun command 134
- mprun job, attaching to 134

N

- Naming Options dialog box 77

- nccq option 151
- no_compiler_vars option 151
- no_control_c_quick_shutdown option 151
- Nodes, selecting number of 128
- no_dumpcore command-line option 151
- no_dump_core option 151
- non-invasive 8
- no_nptl_threads command-line option 151
- notification 32
- notification for guard blocks 33
- notifications 17, 90, 91
 - defined 17
 - error 76, 90
 - exit 17
 - guard blocks 17, 77
 - setting individual events 76
- notify option 32
- Notify when deallocated 93
- Notify when reallocated 93
- notifying at process exit option 75
- nptl_threads command-line option 151

O

- object files
 - merging 5
- od-like features 110
- options 17
 - advanced 74
 - aix_use_fast_trap 150
 - basic 71
 - Guard allocated memory 77
 - Halt execution at process exit 75
 - Halt execution on memory event or error 75

- high 74
- low 74
- medium 74
- painting 119
- Painting memory 80
- setting 71
- orphaned ownership 14
- output TV_HEAP_ARGS value 44
- Overall Totals area 46
- overwriting guard blocks 17

P

- Paint memory option 80
- paint option 37
- painting 37
 - allocation pattern 38
 - blocks 2
 - deallocated memory 56
 - deallocation pattern 38
 - enabling 37
 - memory 78, 80, 119
 - options for 119
 - pattern 119
 - patterns 78, 80
 - zero allocation 37
- painting. See block painting
- parallel programs, adding 69
- Parallel system, selecting 128
- parameter values
 - returning 10
- parameters
 - by reference 11
 - by value 11

- passing arguments to programs 125
- passing pointer to memory to lower frames 11
- passing pointers 11
- patterns for guard blocks 108
- patterns for painting 119
- PC, setting 13
- PE and MemoryScape 131
- PE jobs, attaching 133
- pid command-line option 151
- poe and MPICH 129
- poe, starting using 132
- Point of Deallocation tab 23
- pointers
 - dangling 12
 - passing 10
 - realloc problem 13
- pop-up with block information 96
- port number. 125
- preload variables, by platform 139
- preloading 8, 18
- preloading Memory Debugger agent 8
- process comparisons 115
- process control from context menu 85
- Process Event screen 93
- Process Events screen 21, 91
- process report 87
- Process Status and Control area 69
- processes, attaching to 126
- processes, individually controlling 85
- processes, managing 85
- program execution, controlling 82
- program, attaching 66
- program, attaching limitation 66, 126

- program, mapping to disk 4
- programs
 - compiling 122
- programs, attaching to 126
- programs, passing arguments to 125
- properties dialog box 61
- Properties window 93
- properties, filters 104
- prun, using 134

R

- reachable blocks 37
- reading saved memory state information 67
- realloc errors 24
- realloc not allocated problems 13
- realloc pointer problem 13
- realloc problems 13, 24
 - finding 21
- reallocation timing 12
- realloc_not_allocated event 147
- recording memory requests 74
- reference counting 15
- Related Blocks area 47
- remote 125
- remote computers, debugging on 125
- remote hosts 65
- remote login 132
- remote programs, adding 65
- remote programs, starting 125
- removing filters 104
- reports
 - filtering 45
 - when to create 17

- requests, recording 74
- restoring memory state 111
- retaining deallocated memory 56
- returning parameter values 10
- reuse notifications 92
- reusing memory 34
- Reverse Diff button 115
- RTLD_DEEPBIND
 - and dlopen on Linux 144
- running out of memory 14
- run-time events 90

S

- Save Report command 117
- saved state information, using 111
- save_html_heap_status_source_view action 148
- save_memory_debugging_file action 148
- save_text_heap_status_source_view action 148
- saving heap information 35
- saving memory state 111
- saving memory state information 67
- saving reports
 - as HTML 117
- search controls 95
- search_path command-line option 151
- sections
 - data 5
 - header 6
 - machine code 6
 - symbol table 5
- seeing memory usage 86
- Selected Block area 46
- selecting a block 96

- setting MemoryScape options 71
- setting the PC 13
- setting timeouts 132
- shm option 151, 152
- show_backtrace item 146
- show_backtrace_id item 146
- show_block_address item 146
- show_flags item 146
- show_guard_details item 146
- show_guard_id item 146
- show_guard_status item 146
- show_image item 146
- showing backtraces 33
- showing leaks 98
- show_leak_stats item 146
- show_pc item 146
- show_pid item 146
- signal_handling_mode command-line option 151
- signal_handling_mode option 151
- signals, handling in MemoryScape 151
- SLURM, control_c_quick_shutdown variable 151
- source report 97
- source reports 98
- space, dynamically allocating 13
- stack frames 10
 - arranging 9
- stack memory 9, 11, 86
- stack virtual memory 87
- stack, compared to data section 9
- stac1 frame
 - data blocok 10
- staring MPI programs 69
- starting MemoryScape 16, 61, 125

- memscript command 64
- starting MPI programs 126
- starting remove programs 125
- starting with poe 132
- state information
 - comparing 111
 - exporting 66
 - when discarded 66
- stderr command-line option 152
- stderr_append command-line option 152
- stderr_append command-line option 152
- stderr_is_stdout command-line option 152
- stdin command-line option 152
- stdout command-line option 152
- stdout_append command-line option 152
- stopping execution 66, 91
- stopping execution on heap API misuse 13
- stopping when free problems occur 2
- strdup allocating memory 13
- Summary screen 113
- Sun MPI 134
- Sun MPI and MemoryScape 134
- SunOS 5
 - linking to dbfork library 124
- switch-based communications 132
- symbol table section 5

T

- tag_alloc 42
- tagging 28, 37, 42
 - notify on dealloc 42
 - notify on realloc 42
- Tasks, selecting number of 128

- tasks, specifying 128
- termination_ notification event 147
- text segment memory 86
- timing of reallocations 12
- timeouts, setting 132
- tooltip displaying block data 46
- TotalView lib directory 105
- tracking memory problems 21
- tracking realloc problems 24
- trim, backtrace 33
- tv option 129
- tvdsrv 66, 125
- TVHEAP_ARGS 70
- TV_HEAP_ARGS environment variable 43
 - backtrace_depth 43
 - backtrace_trim 43
 - display_allocations_on_exit 43
 - memalign_strict_alignment_even_multiple 44
 - output 44
 - verbosity 44
- TV_HEAP_ARGS value 43
- TVHEAP_DEEPBIND
 - controlling RTLD_DEEPBIND on Linux 145
- tvheap_mr.a
 - aix_install_tvheap_mr.sh script 140
 - and aix_malloctype.o 140
 - creating using poe 140
 - dynamically loading 140
 - libc.a requirements 140
 - pathname requirements 140
 - relinking executables on AIX 140
- tvheap_mr.a library 140

U

- underwritten destructors 15
- using env to insert agent 139
- using prun 134
- using the Memory Debugger 16

V

- verbosity option 152
- verbosity TV_HEAP_ARGS value 44
- viewing memory contents 110

- viewing the heap graphically 94

- views

 - simplifying 35

- virtual memory 87

- virtual stack memory 87

- visual block display 94

W

- watchpoints 28

- WI,-bkeepfile option 123

- writing a core file 77

- writing a lightweight memory file 77

- writing core files 77

- writing lightweight memory files 77

- wrong address, freeing 24

Z

- zero allocation 37

- zero allocation painting 37

- zoom controls 94

- zooming 89