

Debug Malloc Library

Version 5.5.2
May 2007

Gray Watson

Copyright 1992 to 2007 by Gray Watson.

Published by Gray Watson

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the chapter entitled “Copying” are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the chapter entitled “Copying” may be included in a translation approved by the author instead of in the original English.

Debug Malloc Library

The debug memory allocation or *dmalloc* library has been designed as a drop in replacement for the system's `malloc`, `realloc`, `calloc`, `free` and other memory management routines while providing powerful debugging facilities configurable at runtime. These facilities include such things as memory-leak tracking, fence-post write detection, file/line number reporting, and general logging of statistics.

The library is reasonably portable having been run successfully on at least the following operating systems: AIX, DGUX, Free/Net/OpenBSD, GNU/Hurd, HPUX, Irix, Linux, Mac OSX, NeXT, OSF/DUX, SCO, Solaris, Ultrix, Unixware, MS Windows, and Unicos on a Cray T3E. It also provides support for the debugging of threaded programs. See Section 3.10 [Using With Threads], page 28.

The package includes the library, configuration scripts, debug utility application, test program, and documentation. Online documentation as well as the full source is available at URL <http://dmalloc.com/>. Details on the library's mailing list are available there as well.

Please use the forums at URL <http://dmalloc.com/> to discuss any problems or to request features. If you are still having problems, the author can be reached via his home page at URL <http://256.com/gray/> with questions or feedback. Please include the version number of the library that you are using, your machine and operating system types, and the value of the `DMALLOC_OPTIONS` environment variable.

Gray Watson.

1 Library Copying and Licensing Conditions

Copyright 1992 to 2007 by Gray Watson.

Permission to use, copy, modify, and distribute this software for any purpose and without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies, and that the name of Gray Watson not be used in advertising or publicity pertaining to distribution of the document or software without specific, written prior permission.

Gray Watson makes no representations about the suitability of the software described herein for any purpose. It is provided “as is” without express or implied warranty.

2 Description of Features and How to Get Started

2.1 How to Install the Library

To configure, compile, and install the library, follow these steps carefully.

1. Make sure you have downloaded the latest version of the library available from the home page at URL <http://dmalloc.com/>.
2. The release files have a `.tgz` file extension which means that they are a tar'd gzip'd directory of files. You will need to ungzip and then untar the release file into your source work directory. You may have to rename the file to `.tar.gz` to get some old zip programs to handle the file correctly.
3. You may want to edit or at least review the settings in `settings.dist` to tune specific features of the library. The `configure` script will copy this file to `settings.h` which is where you should be adding per-architecture settings.
4. Type `sh ./configure` to configure the library. You may want to first examine the `config.help` file for some information about configure. You may want to use the `--disable-cxx` option if you do not want the Makefile to build the C++ version of dmalloc. You may want to use the `--enable-threads` option to build the threaded version of dmalloc. You may want to use the `--enable-shlib` option to build the shared versions of the dmalloc libraries. `sh ./configure --help` lists the available options to configure. Configure should generate the Makefile and configuration files automatically.
5. You may want to examine the `Makefile` and `conf.h` files created by configure to make sure it did its job correctly.
6. You might want to tune the settings in `settings.h` file to tune the library to the local architecture. This file contains relevant settings if you are using pthreads or another thread library. See Section 3.10 [Using With Threads], page 28. The `configure` script created this file from the `settings.dist` file. Any permanent changes to these settings should made to the `settings.dist` file. You then can run `config.status` to re-create the `settings.h` file.
7. The `DMALLOC_SIZE` variable gets auto-configured in `dmalloc.h.2` but it may not generate correct settings for all systems. You may have to alter the definitions in this file to get things to stop complaining when you go to compile about the size arguments to malloc routines. Comments on this please.
8. Typing `make` should be enough to build `libdmalloc.a`, and `dmalloc` program. If it does not work, please see if there are any notes in the contrib directory about your system-type. If not and you figure your problem out, please send me some notes so future users can profit from your experiences.

NOTE: You may experience some errors compiling some of the `return.h` assembly macros which attempt to determine the callers address for logging purposes. See Section 5.3 [Portability], page 47. You may want to first try disabling any compiler optimization flags. If this doesn't work then you may need to disable the 'USE_RETURN_MACROS' variable in the `settings.h` file.

NOTE: The code is dependent on an ANSI-C compiler. If the configure script gives the 'WARNING' that you do not have an ANSI-C compiler, you may still be able to add

some sort of option to your compiler to make it ANSI. If there such is an option, please send it to the author so it can be added to the configure script.

9. If you use threads and did not add the `--enable-threads` argument to configure, typing `make threads` should be enough to build `libdmallocth.a` which is the threaded version of the library. This may or may not work depending on the configuration scripts ability to detect your local thread functionality. Feel free to send me mail with improvements.

See the section of the manual on threads for more information about the operation of the library with your threaded program. See Section 3.10 [Using With Threads], page 28.

10. If you have a C++ compiler installed, the library should have automatically built `libdmallocxx.a` which is the C++ version of the library. If it was not done automatically, you can build it by typing `make cxx`. You should link this library into your C++ programs instead of `libdmalloc.a`. See the `dmallocc.cc` C++ file which contains basic code to overload the `new`, `new[]`, `delete`, and `delete[]` C++ operators. My apologies on the minimal C++ support. I am still living in a mostly C world. Any help improving this interface without sacrificing portability would be appreciated.
11. Typing `make light` should build and run the `dmalloc_t` test program through a set of light trials. By default this will execute `dmalloc_t` 5 times – each time will execute 10,000 malloc operations in a very random manner. Anal folks can type `make heavy` to up the ante. Use `dmalloc_t --usage` for the list of all `dmalloc_t` options.
12. Typing `make install` should install the `libdmalloc.a` library in `/usr/local/lib`, the `dmalloc.h` include file in `/usr/local/include`, and the `dmalloc` utility in `/usr/local/bin`. You may also want to type `make installth` to install the thread library into place and/or `make installcc` to install the C++ library into place.

You may have specified a `--prefix=PATH` option to configure in which case `/usr/local` will have been replaced with `PATH`.

See the “Getting Started” section to get up and running with the library. See Section 2.2 [Getting Started], page 6.

2.2 Getting Started with the Library

This section should give you a quick idea on how to get going. Basically, you need to do the following things to make use of the library:

1. Make sure you have downloaded the latest version of the library available from the home page at URL <http://dmalloc.com/>.
2. Follow the installation instructions on how to configure, make, and install the library (i.e. type: `make install`). See Section 2.1 [Installation], page 5.
3. You need to make sure that the library configuration and build process above was able to locate one of the `on_exit` function, `atexit` function, or had compiler destructor support. If one of these functions or support is available then the `dmalloc` library should be able to automatically shut itself down when the program exits. This causes the memory statistics and unfreed information to be dumped to the log file. However, if your system has none of the above, then you will need to call `dmalloc_shutdown` yourself before your program exits.

4. To get the `dmalloc` utility to work you need to add an alias for `dmalloc` to your shell's runtime configuration file if supported. The idea is to have the shell capture the `dmalloc` program's output and adjust the environment.

After you add the alias to the shell config file you need to log out and log back in to have it take effect, or you can execute the appropriate command below on the command line directly. After you setup the alias, if you enter `dmalloc runtime` and see any output with `DMALLOC_OPTIONS` in it then the alias did not take effect.

Bash, ksh, and zsh (<http://www.zsh.org/>) users should add the following to their `.bashrc`, `.profile`, or `.zshrc` file respectively (notice the `-b` option for bourne shell output):

```
function dmalloc { eval 'command dmalloc -b $*'; }
```

If your shell does not support the `command` function then try:

```
function dmalloc { eval '\dmalloc -b $*'; }
```

or

```
function dmalloc { eval '/usr/local/bin/dmalloc -b $*'; }
```

If you are still using `csh` or `tsh`, you should add the following to your `.cshrc` file (notice the `-C` option for c-shell output):

```
alias dmalloc 'eval '\dmalloc -C \!*'
```

If you are using `rc` shell, you should add the following to your `.rcrc` file (notice the `-R` option for `rc`-shell output):

```
fn dmalloc {eval '{/usr/local/bin/dmalloc $*}}
```

5. Although not necessary, you may want to include `dmalloc.h` in your C files and recompile. This will allow the library to report the file/line numbers of calls that generate problems. See Section 3.1 [Allocation Macros], page 15. It should be inserted at the *bottom* of your include files as to not conflict with other includes. You may want to `ifdef` it as well and compile with `cc -DDMALLOC ...`:

```
/* other includes above ^^^ */

#ifdef DMALLOC
#include "dmalloc.h"
#endif
```

6. Another optional task is to compile all of your source with the `dmalloc.h` with the `DMALLOC_FUNC_CHECK` compilation flag. This will allow the library to check all of the arguments of a number of common string and utility routines. See Section 3.3 [Argument Checking], page 16.

```
cc -DDMALLOC -DDMALLOC_FUNC_CHECK file.c
```

7. Link the `dmalloc` library into your program. The `dmalloc` library should probably be placed at or near the end of the library list.
8. Enable the debugging features by typing `dmalloc -l logfile -i 100 low` (for example). You should not see any messages printed by the `dmalloc` utility (see NOTE below). This will:

- Set the malloc logfile name to `logfile` (`-l logfile`). For programs which change directories, you may want to specify the full path to your logfile.

- Have the library check itself every 100 iterations (`-i 100`). This controls how fast your program will run. Larger numbers check the heap less and so it will run faster. Lower numbers will be more likely to catch memory problems.
- Enable a number of debug features (`low`). You can also try `runtime` for minimal checking or `medium` or `high` for more extensive heap verification.
- By default, the low, medium, and high values enable the `error-abort` token which will cause the library to abort and usually dump core immediately upon seeing an error. See Section 3.4 [Dumping Core], page 16. You can disable this feature by entering `dmalloc -m error-abort` (-m for minus) to remove the `error-abort` token and your program will just log errors and continue.

`dmalloc --usage` will provide verbose usage info for the dmalloc program. See Chapter 4 [Dmalloc Program], page 35.

You may also want to install the `dmallocrc` file in your home directory as `.dmallocrc`. This allows you to add your own combination of debug tokens. See Section 4.5 [RC File], page 44.

NOTE: The output from the dmalloc utility should be captured by your shell. If you see a bunch of stuff which includes the string `DMALLOC_OPTIONS` then the alias you should have created above is not working and the environmental variables are not being set. Make sure you've logged out and back in to have the alias take effect.

9. Run your program, examine the logfile that should have been created by `dmalloc_shutdown`, and use its information to help debug your program.

2.3 Basic Description of Terms and Functions

2.3.1 General Memory Terms and Concepts

Any program can be divided into 2 logical parts: text and data. Text is the actual program code in machine-readable format and data is the information that the text operates on when it is executing. The data, in turn, can be divided into 3 logical parts according to where it is stored: *static*, *stack*, and *heap*.

Static data is the information whose storage space is compiled into the program.

```
/* global variables are allocated as static data */
int numbers[10];

main()
{
    ...
}
```

Stack data is data allocated at runtime to hold information used inside of functions. This data is managed by the system in the space called stack space.

```
void foo()
{
    /* this local variable is stored on the stack */
    float total;
    ...
}
```

```

}

main()
{
    foo();
}

```

Heap data is also allocated at runtime and provides a programmer with dynamic memory capabilities.

```

main()
{
    /* the address is stored on the stack */
    char * string;
    ...

    /*
     * Allocate a string of 10 bytes on the heap. Store the
     * address in string which is on the stack.
     */
    string = (char *)malloc(10);
    ...

    /* de-allocate the heap memory now that we're done with it */
    (void)free(string);
    ...
}

```

It is the heap data that is managed by this library.

Although the above is an example of how to use the malloc and free commands, it is not a good example of why using the heap for runtime storage is useful.

Consider this: You write a program that reads a file into memory, processes it, and displays results. You would like to handle files with arbitrary size (from 10 bytes to 1.2 megabytes and more). One problem, however, is that the entire file must be in memory at one time to do the calculations. You don't want to have to allocate 1.2 megabytes when you might only be reading in a 10 byte file because it is wasteful of system resources. Also, you are worried that your program might have to handle files of more than 1.2 megabytes.

A solution: first check out the file's size and then, using the heap-allocation routines, get enough storage to read the entire file into memory. The program will only be using the system resources necessary for the job and you will be guaranteed that your program can handle any sized file.

2.3.2 Functionality Supported by All Malloc Libraries

All malloc libraries support 4 basic memory allocation commands. These include *malloc*, *calloc*, *realloc*, and *free*. For more information about their capabilities, check your system's manual pages – in unix, do a `man 3 malloc`.

```

void *malloc ( unsigned int size ) [Function]
    Usage: pnt = (type *)malloc(size)

```

The malloc routine is the basic memory allocation routine. It allocates an area of `size` bytes. It will return a pointer to the space requested.

```
void *calloc ( unsigned int number, unsigned int size )           [Function]
Usage: pnt = (type *)calloc(number, size)
```

The calloc routine allocates a certain `number` of items, each of `size` bytes, and returns a pointer to the space. It is appropriate to pass in a `sizeof(type)` value as the size argument.

Also, calloc nulls the space that it returns, assuring that the memory is all zeros.

```
void *realloc ( void *old_pnt, unsigned int new_size )          [Function]
Usage: new_pnt = (type *)realloc(old_pnt, new_size)
```

The realloc function expands or shrinks the memory allocation in `old_pnt` to `new_size` number of bytes. Realloc copies as much of the information from `old_pnt` as it can into the `new_pnt` space it returns, up to `new_size` bytes. If there is a problem allocating this memory, 0L will be returned.

If the `old_pnt` is 0L then realloc will do the equivalent of a `malloc(new_size)`. If `new_size` is 0 and `old_pnt` is not 0L, then it will do the equivalent of `free(old_pnt)` and will return 0L.

```
void free ( void *pnt )                                         [Function]
Usage: free(pnt)
```

The free routine releases allocation in `pnt` which was returned by malloc, calloc, or realloc back to the heap. This allows other parts of the program to re-use memory that is not needed anymore. It guarantees that the process does not grow too big and swallow a large portion of the system resources.

WARNING: there is a quite common myth that all of the space that is returned by malloc libraries has already been cleared. *Only* the `calloc` routine will zero the memory space it returns.

2.4 General Features of the Library

The debugging features that are available in this debug malloc library can be divided into a couple basic classifications:

file and line number information

One of the nice things about a good debugger is its ability to provide the file and line number of an offending piece of code. This library attempts to give this functionality with the help of `cpp`, the C preprocessor. See Section 3.1 [Allocation Macros], page 15.

return-address information

To debug calls to the library from external sources (i.e. those files that could not use the allocation macros), some facilities have been provided to supply the caller's address. This address, with the help of a debugger, can help you locate the source of a problem. See Section 3.2 [Return Address], page 15.

fence-post (i.e. bounds) checking

Fence-post memory is the area immediately above or below memory allocations. It is all too easy to write code that accesses above or below an allocation – especially when dealing with arrays or strings. The library can write special values in the areas around every allocation so it will notice when these areas have been overwritten. See Section 3.9.3 [Fence-Post Overruns], page 27.

NOTE: The library cannot notice when the program *reads* from these areas, only when it writes values. Also, fence-post checking will increase the amount of memory the program allocates.

heap-constancy verification

The administration of the library is reasonably complex. If any of the heap-maintenance information is corrupted, the program will either crash or give unpredictable results.

By enabling heap-consistency checking, the library will run through its administrative structures to make sure all is in order. This will mean that problems will be caught faster and diagnosed better.

The drawback of this is, of course, that the library often takes quite a long time to do this. It is suitable to enable this only during development and debugging sessions.

NOTE: the heap checking routines cannot guarantee that the tests will not cause a segmentation-fault if the heap administration structures are properly (or improperly if you will) overwritten. In other words, the tests will verify that everything is okay but may not inform the user of problems in a graceful manner.

logging statistics

One of the reasons why the debug malloc library was initially developed was to track programs' memory usage – specifically to locate *memory leaks* which are places where allocated memory is never getting freed. See Section 3.9.2 [Memory Leaks], page 26.

The library has a number of logging capabilities that can track un-freed memory pointers as well as runtime memory usage, memory transactions, administrative actions, and final statistics.

examining freed memory

Another common problem happens when a program frees a memory pointer but goes on to use it again by mistake. This can lead to mysterious crashes and unexplained problems.

To combat this, the library can write special values into a block of memory after it has been freed. This serves two purposes: it will make sure that the program will get garbage data if it trying to access the area again, and it will allow the library to verify the area later for signs of overwriting.

If any of the above debugging features detect an error, the library will try to recover. If logging is enabled then an error will be logged with as much information as possible.

The error messages that the library displays are designed to give the most information for developers. If the error message is not understood, then it is most likely just trying to indicate that a part of the heap has been corrupted.

The library can be configured to quit immediately when an error is detected and to dump a core file or memory-image. This can be examined with a debugger to determine the source of the problem. The library can either stop after dumping core or continue running. See Section 3.4 [Dumping Core], page 16.

NOTE: do not be surprised if the library catches problems with your system's routines. It took me hours to finally come to the conclusion that the localtime call, included in SunOS release 4.1, overwrites one of its fence-post markers.

2.5 How the Library Checks Your Program

This is one of the newer sections of the library implying that it is incomplete. If you have any questions or issues that you'd like to see handled here, please let me know.

The dmalloc library replaces the heap library calls normally found in your system libraries with its own versions. When you make a call to malloc (for example), you are calling dmalloc's version of the memory allocation function. When you allocate memory with these functions, the dmalloc library keeps track of a number of pieces of debugging information about your pointer including: where it was allocated, exactly how much memory was requested, when the call was made, etc.. This information can then be verified when the pointer is freed or reallocated and the details can be logged on any errors.

Whenever you reallocate or free a memory address, the dmalloc library always performs a number of checks on the pointer to make sure that it is valid and has not been corrupted. You can configure the library to perform additional checks such as detected fence-post writing. The library can also be configured to overwrite memory with non-zeros (only if calloc is not called) when it is allocated and erase the memory when the pointers are freed.

In addition to per-pointer checks, you can configure the library to perform complete heap checks. These complete checks verify all internal heap structures and include walking all of the known allocated pointers to verify each one in turn. You need this level of checking to find random pointers in your program which got corrupted but that won't be freed for a while. To turn on these checks, you will need to enable the **check-heap** debug token. See Section 4.4 [Debug Tokens], page 41. By default this will cause the heap to be fully checked each and every time dmalloc is called whether it is a malloc, free, realloc, or another dmalloc overloaded function.

Performing a full heap check can take a good bit of CPU and it may be that you will want to run it sporadically. This can be accomplished in a couple different ways including the '-i' interval argument to the dmalloc utility. See Chapter 4 [Dmalloc Program], page 35. This will cause the check to be run every N-th time. For instance, 'dmalloc -i 3' will cause the heap to be checked before every 3rd call to a memory function. Values of 100 or even 1000 for high memory usage programs are more useful than smaller ones.

You can also cause the program to start doing detailed heap checking after a certain point. For instance, with 'dmalloc -s 1000' option, you can tell the dmalloc library to enable the heap checks after the 1000th memory call. Examine the dmalloc log file produced and use the iteration count if you have LOG_ITERATION_COUNT enabled in your **settings.h** file.

The `start` option can also have the format `'file:line'`. For instance, if it is set to `'dmalloc_t.c:126'`, `dmalloc` will start checking the heap after it sees a `dmalloc` call from the `dmalloc_t.c` file, line number 126. If you use `'dmalloc_t.c:0'`, with a 0 line number, then `dmalloc` will start checking the heap after it sees a call from anywhere in the `dmalloc_t.c` file.

3 How to Program with the Library

3.1 Macros Providing File and Line Information

By including `dmalloc.h` in your C files, your calls to `malloc`, `calloc`, `realloc`, `recalloc`, `memalign`, `posix_memalign`, `valloc`, `strdup`, and `free` are replaced with calls to `_dmalloc_malloc`, `_dmalloc_realloc`, and `_dmalloc_free` with various flags. Additionally the library replaces calls to `xmalloc`, `xcalloc`, `xrealloc`, `xrecalloc`, `xmemalign`, `xvalloc`, `xstrdup`, and `xfree` with associated calls.

These macros use the c-preprocessor `__FILE__` and `__LINE__` macros which get replaced at compilation time with the current file and line-number of the source code in question. The routines use this information to produce verbose reports on memory problems.

```
not freed: '0x38410' (22 bytes) from 'dmalloc_t.c:92'
```

This line from a log file shows that memory was not freed from file `dmalloc_t.c` line 92. See Section 3.9.2 [Memory Leaks], page 26.

You may notice some non standard memory allocation functions in the above list. `Recalloc` is a routine like `realloc` that reallocates previously allocated memory to a new size. If the new memory size is larger than the old, `recalloc` initializes the new space to all zeros. This may or may not be supported natively by your operating system. `Memalign` is like `malloc` but should insure that the returned pointer is aligned to a certain number of specified bytes. Currently, the `memalign` function is not supported by the library. It defaults to returning possibly non-aligned memory for alignment values less than a block-size. `Valloc` is like `malloc` but insures that the returned pointer will be aligned to a page boundary. This may or may not be supported natively by your operating system but is fully supported by the library. `Strdup` is a string duplicating routine which takes in a null terminated string pointer and returns an allocated copy of the string that will need to be passed to `free` later to deallocate.

The X versions of the standard memory functions (`xmalloc`, `xfree`, etc.) will print out an error message to standard error and will stop if the library is unable to allocate any additional memory. It is useful to use these routines instead of checking everywhere in your program for allocation routines returning `NULL` pointers.

WARNING: If you are including the `dmalloc.h` file in your sources, it is recommended that it be at the end of your include file list because `dmalloc` uses macros and may try to change declarations of the `malloc` functions if they come after it.

3.2 Getting Caller Address Information

Even though the allocation macros can provide file/line information for some of your code, there are still modules which either you can't include `dmalloc.h` (such as library routines) or you just don't want to. You can still get information about the routines that call `dmalloc` function from the return-address information. To accomplish this, you must be using this library on one of the supported architecture/compiler. See Section 5.3 [Portability], page 47.

The library attempts to use some assembly hacks to get the return-address or the address of the line that called the `dmalloc` function. If you have unfreed memory that does not have associated file and line information, you might see the following non-freed memory messages.

```
not freed: '0x38410' (22 bytes) from 'ra=0xdd2c'
not freed: '0x38600' (10232 bytes) from 'ra=0x10234d'
not freed: '0x38220' (137 bytes) from 'ra=0x82cc'
```

With the help of a debugger, these return-addresses (or ra) can then be identified. I've provided a `ra_info.pl` perl script in the `contrib/` directory with the `dmalloc` sources which seems to work well with `gdb`. You can also use manual methods for `gdb` to find the return-address location. See Section 3.9.4 [Translate Return Addresses], page 28.

3.3 Checking of Function Arguments

One potential problem with the library and its multitude of checks and diagnoses is that they only get performed when a `dmalloc` function is called. One solution this is to include `dmalloc.h` and compile your source code with the `DMALLOC_FUNC_CHECK` flag defined and enable the `check-funcs` token. See Section 4.4 [Debug Tokens], page 41.

```
cc -DDMALLOC -DDMALLOC_FUNC_CHECK file.c
```

NOTE: Once you have compiled your source with `DMALLOC_FUNC_CHECK` enabled, you will have to recompile with it off to disconnect the library. See Section 3.7 [Disabling the Library], page 24.

WARNING: You should be sure to have `dmalloc.h` included at the end of your include file list because `dmalloc` uses macros and may try to change declarations of the checked functions if they come after it.

When this is defined `dmalloc` will override a number of functions and will insert a routine which knows how to check its own arguments and then call the real function. `Dmalloc` can check such functions as `bcopy`, `index`, `strcat`, and `strcascmp`. For the full list see the end of `dmalloc.h`.

When you call `strlen`, for instance, `dmalloc` will make sure the string argument's fence-post areas have not been overwritten, its file and line number locations are good, etc. With `bcopy`, `dmalloc` will make sure that the destination string has enough space to store the number of bytes specified.

For all of the arguments checked, if the pointer is not in the heap then it is ignored since `dmalloc` does not know anything about it.

3.4 Generating a Core File on Errors

If the `error-abort` debug token has been enabled, when the library detects any problems with the heap memory, it will immediately attempt to dump a core file. See Section 4.4 [Debug Tokens], page 41. Core files are a complete copy of the program and its state and can be used by a debugger to see specifically what is going on when the error occurred. See Section 3.9 [Using With a Debugger], page 25. By default, the low, medium, and high arguments to the library utility enable the `error-abort` token. You can disable this feature by entering `dmalloc -m error-abort` (-m for minus) to remove the `error-abort` token and your program will just log errors and continue. You can also use the `error-dump` token which tries to dump core when it sees an error but still continue running. See Section 4.4 [Debug Tokens], page 41.

When a program dumps core, the system writes the program and all of its memory to a file on disk usually named `core`. If your program is called `foo` then your system may dump

core as `foo.core`. If you are not getting a `core` file, make sure that your program has not changed to a new directory meaning that it may have written the core file in a different location. Also insure that your program has write privileges over the directory that it is in otherwise it will not be able to dump a core file. Core dumps are often security problems since they contain all program memory so systems often block their being produced. You will want to check your user and system's core dump size `ulimit` settings.

The library by default uses the `abort` function to dump core which may or may not work depending on your operating system. If the following program does not dump core then this may be the problem. See `KILL_PROCESS` definition in `settings.dist`.

```
main()
{
    abort();
}
```

If `abort` does work then you may want to try the following setting in `settings.dist`. This code tries to generate a segmentation fault by dereferencing a `NULL` pointer.

```
#define KILL_PROCESS    { int *_int_p = 0L; *_int_p = 1; }
```

3.5 Additional Non-standard Routines

The library has a number of variables that are not a standard part of most malloc libraries:

`int dmalloc_errno` [Variable]

This variable stores the internal dmalloc library error number like `errno` does for the system calls. It can be passed to `dmalloc_strerror()` (see below) to get a string version of the error. It will have a value of zero if the library has not detected any problems.

`char* dmalloc_logpath` [Variable]

This variable can be used to set the dmalloc log filename. The env variable `DMALLOC_LOGFILE` overrides this variable.

Additionally the library provides a number of non-standard malloc routines:

`void dmalloc_shutdown (void)` [Function]

This function shuts the library down and logs the final statistics and information especially the non-freed memory pointers. The library has code to support auto-shutdown if your system has the `on_exit()` call, `atexit()` call, or compiler destructor support (see `conf.h`). If you do not have these, then `dmalloc_shutdown` should be called right before `exit()` or as the last function in `main()`.

```
main()
{
    ...
    dmalloc_shutdown();
    exit(0);
}
```

`int dmalloc_verify (char * pnt)` [Function]

This function verifies individual memory pointers that are suspect of memory problems. To check the entire heap pass in a NULL or 0 pointer. The routine returns `DMALLOC_VERIFY_ERROR` or `DMALLOC_VERIFY_NOERROR`.

NOTE: ‘`dmalloc_verify()`’ can only check the heap with the functions that have been enabled. For example, if fence-post checking is not enabled, ‘`dmalloc_verify()`’ cannot check the fence-post areas in the heap.

`unsigned-int dmalloc_debug (const unsigned int flags)` [Function]

This routine sets the debug functionality flags and returns the previous flag value. It is helpful in server or cgi-bin programs where environmental variables cannot be used. See Section 3.12 [Debugging A Server], page 30. For instance, if debugging should never be enabled for a program, a call to `dmalloc_debug(0)` as the first call in `main()` will disable all the memory debugging from that point on.

NOTE: you cannot add or remove certain flags such as signal handlers since they are setup at initialization time only.

NOTE: you can also use `dmalloc_debug_setup` below.

`unsigned-int dmalloc_debug_current (void)` [Function]

This routine returns the current debug functionality value. This allows you to save a copy of the debug dmalloc settings to be changed and then restored later.

`void dmalloc_debug_setup (const char * options_str)` [Function]

This routine sets the global debugging functionality as an option string. Normally this would be passed in in the `DMALLOC_OPTIONS` environmental variable. This is here to override the env or for circumstances where modifying the environment is not possible or does not apply such as servers or cgi-bin programs. See Section 3.12 [Debugging A Server], page 30.

Some examples:

```
/*
 * debug tokens high, threaded lock-on at 20,
 * log to dmalloc.%p (pid)
 */
dmalloc_debug_setup("debug=0x4f46d03,lockon=20,log=dmalloc.%p");
```

```
/*
 * turn on some debug tokens directly and log to the
 * file 'logfile'
 */
dmalloc_debug_setup(
    "log-stats,log-non-free,check-fence,log=logfile");
```

`int dmalloc_examine (const DMALLOC_PNT pnt, DMALLOC_SIZE * user_size_p, DMALLOC_SIZE * total_size_p, char ** file_p, int * line_p, DMALLOC_PNT * ret_addr_p, unsigned long * user_mark_p, unsigned long * seen_p)` [Function]

This function returns the size of a pointer’s allocation as well as the total size given including administrative overhead, file and line or the return-address from where it

was allocated, the last pointer when the pointer was "used", and the number of times the pointer has been "seen". It will return `DMALLOC_NOERROR` or `DMALLOC_ERROR` depending on whether `pnt` is good or not.

NOTE: This function is *certainly* not provided by most if not all other malloc libraries.

```
int dmalloc_tag ( const DMALLOC_PNT pnt, char *file, int          [Function]
                 line)
```

This macro sets a pointer's file and line information to the supplied values. *file must* be a constant string, it is not copied.

```
void dmalloc_track ( const dmalloc_track_t track_func )          [Function]
```

Register an allocation tracking function which will be called each time an allocation occurs. Pass in `NULL` to disable. To take a look at what information is provided, see the `dmalloc_track_t` function typedef in `dmalloc.h`.

```
unsigned-long dmalloc_mark ( void )                             [Function]
```

Return to the caller the current "mark" which can be used later to log the pointers which have changed since this mark with the `dmalloc_log_changed` function. Multiple marks can be saved and used.

This is very useful when using the library with a server which does not exit. You can then save a mark before a transaction or event happens and then check to see what has changed using the `dmalloc_log_changed` function below. See Section 3.12 [Debugging A Server], page 30.

If you `LOG_ITERATION` enabled in your `settings.h` file then the entries in the log file will be prepended with the number of memory transactions that the library has handled so far. You can also enable `LOG_PNT_ITERATION` in `settings.h` to store the memory transaction number with each pointer.

```
unsigned-long dmalloc_memory_allocated ( void )                 [Function]
```

Return to the caller the total number of bytes that have been allocated by the library. This is not the current in use but the total number of bytes returned by allocation functions.

```
unsigned-int dmalloc_page_size ( void )                        [Function]
```

Return to the caller the memory page-size being used by the library. This should be the same value as the one returned by the `getpagesize()` function, if available.

```
unsigned-long dmalloc_count_changed ( const unsigned long     [Function]
                                     mark, const int not_freed_b, const int free_b )
```

Count the pointers that have changed since the mark which was returned by `dmalloc_mark`. If `not_freed_b` is set to non-0 then count the pointers that have not been freed. If `free_b` is set to non-0 then count the pointers that have been freed.

This can be used in conjunction with the `dmalloc_mark()` function to help servers which never exit ensure that transactions or events are not leaking memory. See Section 3.12 [Debugging A Server], page 30.

```
    unsigned long mark = dmalloc_mark() ;
    ...
    assert(dmalloc_count_changed(mark, 1, 0) == 0) ;
```

`void dmalloc_log_stats (void)` [Function]

This routine outputs the current dmalloc statistics to the log file.

`void dmalloc_log_unfreed (void)` [Function]

This function logs the unfreed-memory information to the log file. This is also useful to log the currently allocated points to the log file to be compared against another dump later on.

`void dmalloc_log_changed (const unsigned long mark, const int not_freed_b, const int freed_b, const int details_b)` [Function]

Log the pointers that have changed since the mark which was returned by `dmalloc_mark`. If `not_freed_b` is set to non-0 then log the pointers that have not been freed. If `free_b` is set to non-0 then log the pointers that have been freed. If `details_b` set to non-0 then log the individual pointers that have changed otherwise just log the summaries.

This can be used in conjunction with the `dmalloc_mark()` function to help servers which never exit find transactions or events which are leaking memory. See Section 3.12 [Debugging A Server], page 30.

`void dmalloc_vmessage (const char * format, va_list args)` [Function]

Write a message into the dmalloc logfile using `vprintf`-like arguments.

`void dmalloc_message (const char * format, ...)` [Function]

Write a message into the dmalloc logfile using `printf`-like arguments.

`void dmalloc_get_stats (DMALLOC_PNT * heap_low_p, DMALLOC_PNT * heap_high_p, unsigned long * total_space_p, unsigned long * user_space_p, unsigned long * current_allocated_p, unsigned long * current_pnt_np, unsigned long * max_allocated_p, unsigned long * max_pnt_np, unsigned long * max_one_p)` [Function]

This function return a number of statistics about the current heap. The pointers `heap_low_p` and `heap_high_p` will be set to the low and high spots in the heap. `total_space_p` will be set to the total space in the heap including user space, administrative space, and overhead. `user_space_p` will be set to the space given to the user process (allocated and free space). `current_allocated_p` will be set to the current allocated space given to the user process. `current_pnt_np` will be set to the current number of pointers allocated by the user process. `max_allocated_p` will be set to the maximum allocated space given to the user process. `max_pnt_np` will be set to the maximum number of pointers allocated by the user process. `max_on_p` will be set to the maximum space allocated with one call by the user process.

`const char* dmalloc_strerror (const int error_number)` [Function]

This function returns the string representation of the error value in `error_number` (which probably should be `dmalloc_errno`). This allows the logging of more verbose memory error messages.

You can also display the string representation of an error value by a call to the `dmalloc` program with a `-e #` option. See Chapter 4 [Dmalloc Program], page 35.

3.6 Description of the Internal Error Codes

The following error codes are defined in `error_val.h`. They are used by the library to indicate a detected problem. They can be caused by the user (`ERROR_TOO_BIG`) or can indicate an internal library problem (`ERROR_SLOT_CORRUPT`). The `dmalloc` utility can give you the string version of the error with the `-e` argument:

```
$ dmalloc -e 60
dmalloc: dmalloc_errno value '60' =
    'pointer is not on block boundary'
```

Here are the error codes set by the library. They are non contiguous on purpose because I add and delete codes all of the time and there are sections for various error-code types.

1 (ERROR_NONE) no error

No error. It is good coding practice to set the no-error code to be non-0 value because it forces you to set it explicitly.

2 (INVALID_ERROR)

Invalid error number. If the library outputs this error then your `dmalloc` utility may be out of date with the library you linked against. This will be returned with all error codes not listed here.

10 (ERROR_BAD_SETUP) initialization and setup failed

Bad setup value. This is currently unused but it is intended to report on invalid setup configuration information.

11 (ERROR_IN_TWICE) malloc library has gone recursive

Library went recursive. This usually indicates that you are not using the threaded version of the library. Or if you are then you are not using the `'-o'` "lock-on" option. See Section 3.10 [Using With Threads], page 28.

13 (ERROR_LOCK_NOT_CONFIG) thread locking has not been configured

Thread locking has not been configured. This indicates that you attempted to use the `'-o'` "lock-on" option without linking with the thread version of the library. You should probably be using `-ldmalloca` *not* `-ldmalloc` when you are linking. Or you should include `.../lib/libdmalloca.a` on your compilation line.

20 (ERROR_IS_NULL) pointer is null

Pointer is null. The program passed a NULL (0L) pointer to `free` and you have the `error-free-null` token enabled.

21 (ERROR_NOT_IN_HEAP) pointer is not pointing to heap data space

Pointer is not pointing to heap data space. This means that the program passed an out-of-bounds pointer to `free` or `realloc`. This could be someone trying to work with a wild pointer or trying to free a pointer from a different source than `malloc`.

22 (ERROR_NOT_FOUND) cannot locate pointer in heap

Cannot locate pointer in heap. The user passed in a pointer which the heap did not know about. Either this pointer was allocated by some other mechanism (like `mmap` or `sbrk` directly) or it is a random invalid pointer.

In some rare circumstances, sometimes seen with shared libraries, there can be two separate copies of the dmalloc library in a program. Each one does not know about the pointers allocated by the other.

23 (ERROR_IS_FOUND) found pointer the user was looking for

This indicates that the pointer specified in the address part of the environmental variable was discovered by the library. See Section 4.3 [Environment Variable], page 40. This error is useful so you can put a breakpoint in a debugger to find where a particular address was allocated. See Section 3.9 [Using With a Debugger], page 25.

24 (ERROR_BAD_FILE) possibly bad .c filename pointer

A possibly invalid filename was discovered in the dmalloc administrative sections. This could indicate some corruption of the internal tables. It also could mean that you have a source file whose name is longer than 100 characters. See `MAX_FILE_LENGTH` in the `settings.dist` file.

25 (ERROR_BAD_LINE) possibly bad .c file line-number

A line-number was out-of-bounds in the dmalloc administrative sections. This could indicate some corruption of the internal tables. It also could mean that you have a source file containing more than 30000 lines of code. See `MAX_LINE_NUMBER` in the `settings.dist` file.

26 (ERROR_UNDER_FENCE) failed UNDER picket-fence magic-number check

This indicates that a pointer had its lower bound picket-fence magic number overwritten. If the `check-fence` token is enabled, the library writes magic values above and below allocations to protect against overflow. Most likely this is because a pointer below it went past its allocate and wrote into the next pointer's space.

27 (ERROR_OVER_FENCE) failed OVER picket-fence magic-number check

This indicates that a pointer had its upper bound picket-fence magic space overwritten. If the `check-fence` token is enabled, the library writes magic values above and below allocations to protect against overflow. Most likely this is because an array or string allocation wrote past the end of the allocation.

Check for improper usage of `strcat`, `sprintf`, `strcpy`, and any other functions which work with strings and do not protect themselves by tracking the size of the string. These functions should *always* be replaced with: `strncat`, `snprintf`, `strncpy`, and others.

28 (ERROR_WOULD_OVERWRITE) use of pointer would exceed allocation

This error is generated by the function pointer checking code usually enabled with the `check-funcs` token. Dmalloc overloads a number of string and memory copying functions and verifies that the buffers (if allocated in the heap) would not be overwritten by the function.

30 (ERROR_NOT_START_BLOCK) pointer is not to start of memory block

This indicates that the user passed in a pointer to be freed or reallocated that was not at the start of the allocation. You would get this error, for example, if you allocate and get pointer `X` but then try to free `X+1`.

40 (ERROR_BAD_SIZE) invalid allocation size

This error indicates that a size value in the internal structures of the library were corrupted. This could be a random pointer problem, pointer overflow, or some other corruption.

41 (ERROR_TOO_BIG) largest maximum allocation size exceeded

An allocation asked for memory larger than the configured maximum. This is a user configured setting. See `LARGEST_ALLOCATION` in the `settings.dist` file. It is used to protect against wild allocation sizes. If you have super large allocation sizes then you should tune the `LARGEST_ALLOCATION` value appropriately.

43 (ERROR_ALLOC_FAILED) could not grow heap by allocating memory

The library could not allocate more heap space and the program has run out of memory. This could indicate that you've overflowed some system imposed limit. On many operation systems, the `ulimit` call can tune system defaults. The library uses a lot more memory compared to the system `malloc` library because it stores a lot more information about the allocated pointers.

NOTE: This also may be due to an inability of your operating system to use the `mmap` system call to allocate memory. You may need to force the `USE_MMAP` setting to be 0. Please use the forums at URL <http://dmalloc.com/> to report issues with this.

45 (ERROR_OVER_LIMIT) over user specified allocation limit

The library has allocated more memory than was specified in the memory-limit environmental variable. See Section 4.3 [Environment Variable], page 40.

60 (ERROR_NOT_ON_BLOCK) pointer is not on block boundary

The user tried to free or realloc a pointer that was not pointing to a block boundary. You would get this error, for example, if you allocate and get pointer `X` but then try to free `X+1`.

61 (ERROR_ALREADY_FREE) tried to free previously freed pointer

The user tried to free a pointer than has already been freed. This is a very common mistake and can lead to serious problems. It can be because a destructor is being called twice for some reason. Although tracking down the specific source is highly recommended, it is good to set pointers to `NULL` (0L) after you free them as a rule.

67 (ERROR_FREE_OVERWRITTEN) free space has been overwritten

If either the `free-blank` or `check-blank` tokens are enabled then the library will overwrite memory when it is freed with the “`dmalloc-free`” byte (hex 0xdf, octal 0337, decimal 223). If the program writes into this space, then the library will detect the write and trigger this error. This could indicate that the program is using a pointer after it has been freed.

70 (ERROR_ADMIN_LIST) bad admin structure list

An internal corruption in the library's administrative structures has been detected. This could be a random pointer problem, pointer overflow, or some other corruption.

72 (ERROR_ADDRESS_LIST) internal address list corruption

An internal corruption in the library's administrative structures has been detected. This could be a random pointer problem, pointer overflow, or some other corruption.

73 (ERROR_SLOT_CORRUPT) internal memory slot corruption

An internal corruption in the library's administrative structures has been detected. This could be a random pointer problem, pointer overflow, or some other corruption.

3.7 How to Disable the library

If you would like to disable the library's detailed checking features during a particularly allocation intensive section of code, you can do something like the following:

```

unsigned int dmalloc_flags;
...
/* turn off all debug flags and save a copy of old value */
dmalloc_flags = dmalloc_debug(0);

/* section of a lot of allocations */
...
/* end of section */

/* restore the dmalloc flag setting */
dmalloc_debug(dmalloc_flags);

```

When you are finished with the development and debugging sessions, you may want to disable the dmalloc library and put in its place either the system's memory-allocation routines, gnu-malloc, or maybe your own. Attempts have been made to make this a reasonably painless process. The ease of the extraction depends heavily on how many of the library's features you made use of during your coding.

Reasonable suggestions are welcome as to how to improve this process while maintaining the effectiveness of the debugging.

- If you want to *totally* disable the dmalloc library then you will need to recompile all the C files that include `dmalloc.h` while defining `DMALLOC_DISABLE`. This will cause the dmalloc macros to not be applied. See Section 3.1 [Allocation Macros], page 15.

```
cc -g -DDMALLOC_DISABLE file.c
```

An alternative is to surround the `dmalloc.h` inclusion or any direct dmalloc references with an `#ifdef DMALLOC` and then just remove the `-DDMALLOC`.

```

#ifdef DMALLOC
#include "dmalloc.h"
#endif

main()
{
...

```

```

#ifdef DMALLOC
    dmalloc_verify(OL);
#endif
    return 0;
}

// to get dmalloc information
$ cc -DDMALLOC main.c

// without dmalloc information
$ cc main.c

```

- If you compiled any of your source modules with `DMALLOC_FUNC_CHECK` defined then you must first recompile all those modules without the flag enabled.

If you have disabled `dmalloc` with the `DMALLOC_DISABLED` flag or never included `dmalloc.h` in any of your C files, then you will not need to recompile your sources when you need to disable the library.

If you get unresolved references like `_dmalloc_malloc` or `_dmalloc_bcopy` then something was not disabled as it should have been.

3.8 Using the Library with C++

For those people using the C++ language, the library tries to configure and build `libdmallocxx.a` library. This library should be linked into your C++ programs instead of `libdmalloc.a`.

`Dmalloc` is not as good with C++ as C because the dynamic memory routines in C++ are `new()` and `delete()` as opposed to `malloc()` and `free()`. Since `new` and `delete` are usually not used as functions but rather as `x = new type`, there is no easy way for `dmalloc` to pass in file and line information unfortunately. The `libdmallocxx.a` library provides the file `dmallocc.cc` which effectively redirects `new` to the more familiar `malloc` and `delete` to the more familiar `free`.

NOTE: The author is not a C++ hacker so feedback in the form of other hints and ideas for C++ users would be much appreciated.

3.9 Using Dmalloc With a Debugger

Here are a number of possible scenarios for using the `dmalloc` library to track down problems with your program.

You should first enable a logfile filename and turn on a set of debug features. You can use `dmalloc -l logfile low` to accomplish this. If you are interested in having the error messages printed to your terminal as well, enable the `print-messages` token by typing `dmalloc -p print-messages` afterwards. See Chapter 4 [Dmalloc Program], page 35.

Now you can enter your debugger (I use the *excellent* GNU debugger `gdb`), and put a break-point in `dmalloc_error()` which is the internal error routine for the library. When your program is run, it will stop there if a memory problem is detected.

If you are using `GDB`, I would recommend adding the contents of `dmalloc.gdb` in the `contrib` subdirectory to your `.gdbinit` file in your home directory. This enables the

`dmalloc` command which will prompt you for the arguments to the `dmalloc` command and will set a break point in `dmalloc_error()` automatically.

If you are using shared libraries, you may want to execute the following commands initially to load in `dmalloc` and other library symbols:

```
(gdb) sharedlibrary
(gdb) add-shared-symbol-files
```

3.9.1 Diagnosing General Problems with a Debugger

If your program stops at the `dmalloc_error()` routine then one of a number of problems could be happening. Incorrect arguments could have been passed to a `malloc` call: asking for negative number of bytes, trying to realloc a non-heap pointer, etc.. There also could be a problem with the system's allocations: you've run out of memory, some other function in your program is using the heap allocation functions `mmap` or `sbrk`, etc.. However, it is most likely that some code that has been executed was naughty.

To get more information about the problem, first print via the debugger the `dmalloc_errno` variable to get the library's internal error code. You can suspend your debugger and run `dmalloc -e value-returned-from-print` to get an English translation of the error. A number of the error messages are designed to indicate specific problems with the library administrative structures and may not be user-friendly.

If the problem was due to the arguments or system allocations then the source of the problem has been found. However, if some code did something wrong, you may have some more work to do to locate the actual problem. The `check-heap` token should be enabled and the interval setting disabled or set to a low value so that the library can find the problem as close as possible to its source. The code that was execute right before the library halted, can then be examined closely for irregularities. See Section 4.4 [Debug Tokens], page 41, See Chapter 4 [Dmalloc Program], page 35.

You may also want to put calls to `dmalloc_verify(0)` in your code before the section which generated the error. This should locate the problem faster by checking the library's structures at that point. See Section 3.5 [Extensions], page 17.

3.9.2 Tracking Down Non-Freed Memory

So you've run your program, examined the log-file and discovered (to your horror) some un-freed memory. Memory leaks can become large problems since even the smallest and most insignificant leak can starve the program given the right circumstances.

```
not freed: '0x45008' (12 bytes) from 'ra=0x1f8f4'
not freed: '0x45028' (12 bytes) from 'unknown'
not freed: '0x45048' (10 bytes) from 'argv.c:1077'
  known memory not freed: 1 pointer, 10 bytes
 unknown memory not freed: 2 pointers, 24 bytes
```

Above you will see a sample of some non-freed memory messages from the logfile. In the first line the `'0x45008'` is the pointer that was not freed, the `'12 bytes'` is the size of the unfreed block, and the `'ra=0x1f8f4'` or return-address shows where the allocation originated from. See Section 3.9.4 [Translate Return Addresses], page 28.

The systems which cannot provide return-address information show `'unknown'` instead, as in the 2nd line in the sample above.

The ‘`argv.c:1077`’ information from the 3rd line shows the file and line number which allocated the memory which was not freed. This information comes from the calls from C files which included `dmalloc.h`. See Section 3.1 [Allocation Macros], page 15.

At the bottom of the sample it totals the memory for you and breaks it down to known memory (those calls which supplied the file/line information) and unknown (the rest).

Often, you may allocate memory in via `strdup()` or another routine, so the logfile listing where in the `strdup` routine the memory was allocated does not help locate the true source of the memory leak – the routine that called `strdup`. Without a mechanism to trace the calling stack, there is no way for the library to see who the caller of the caller (so to speak) was.

However, there is a way to track down unfreed memory in this circumstance. You need to compile the library with `STORE_SEEN_COUNT` defined in `conf.h`. The library will then record how many times a pointer has been allocated or freed. It will display the unfreed memory as:

```
not freed: '0x45008|s3' (12 bytes) from 'ra=0x1f8f4'
```

The `STORE_SEEN_COUNT` option adds a ‘`|s#`’ qualifier to the address. This means that the address in question was seen ‘`#`’ many times. In the above example, the address ‘`0x45008`’ was seen ‘`3`’ times. The last time it was allocated, it was not freed.

How can a pointer be “seen” 3 times? Let say you `strdup` a string of 12 characters and get address ‘`0x45008`’ – this is `#1` time the pointer is seen. You then free the pointer (seen `#2`) but later `strdup` another 12 character string and it gets the ‘`0x45008`’ address from the free list (seen `#3`).

So to find out who is allocating this particular 12 bytes the 3rd time, try `dmalloc -a 0x45008:3`. The library will stop the program the third time it sees the ‘`0x45008`’ address. You then enter a debugger and put a breakpoint at `dmalloc_error`. Run the program and when the breakpoint is reached you can examine the stack frame to determine who called `strdup` to allocate the pointer.

To not bother with the `STORE_SEEN_COUNT` feature, you can also run your program with the `never-reuse` token enabled. This token will cause the library to never reuse memory that has been freed. Unique addresses are always generated. This should be used with caution since it may cause your program to run out of memory.

3.9.3 Diagnosing Fence-Post Overwritten Memory

For a definition of fence-posts please see the “Features” section. See Section 2.4 [Features], page 10.

To detect fence-post overruns, you need to enable the ‘`check-fence`’ token. See Section 4.4 [Debug Tokens], page 41. This pads your allocations with some extra bytes at the front and the end and watches the space to make sure that they don’t get overwritten. *NOTE:* The library cannot detect if this space gets read, only written.

If you have encountered a fence-post memory error, the logfile should be able to tell you the offending address.

```
free: failed UNDER picket-fence magic-number checking:
pointer '0x1d008' from 'dmalloc_t.c:427'
Dump of proper fence-bottom bytes: '\e\253\300\300\e\253\300\300'
```

```
Dump of '0x1d008'-8: '\e\253\300\300WOW!\003\001pforger\023\001\123'
```

The above sample shows that the pointer '0x1d008' has had its lower fence-post area overwritten. This means that the code wrote below the bottom of the address or above the address right below this one. In the sample, the string that did it was 'WOW!'.

The library first shows you what the proper fence-post information should look like, and then shows what the pointer's bad information was. If it cannot print the character, it will display the value as '\ddd' where ddd are three octal digits.

By enabling the `check-heap` debugging token and assigning the interval setting to a low number, you should be able to locate approximately when this problem happened. See Section 4.4 [Debug Tokens], page 41, See Chapter 4 [Dmalloc Program], page 35.

3.9.4 Translating Return Addresses into Code Locations

The following gdb commands help you translate the return-addresses (ra=) entries in the logfile into locations in your code. I've provided a `ra_info.pl` perl script in the `contrib/` directory with the dmalloc sources which seems to work well with gdb. But, if you need to do it manually, here are the commands in gdb to use.

```
# you may need to add the following commands to load in shared libraries
(gdb) sharedlibrary
(gdb) add-shared-symbol-files

(gdb) x 0x10234d
0x10234d <_findbuf+132>: 0x7fffceb7

(gdb) info line *(0x82cc)
Line 1092 of argv.c starts at pc 0x7540 and ends at 0x7550.
```

In the above example, gdb was used to find that the two non-freed memory pointers were allocated in `_findbuf()` and in file `argv.c` line 1092 respectively. The 'x address' (for examine) can always be used on the return-addresses but the 'info line *(address)' will only work if that file was compiled using the `-g` option and has not been stripped. This limitation may not be true in later versions of gdb.

3.10 Using the Library with a Thread Package

Threads are special operating system facilities which allow your programs to have multiple threads of execution (hence the name). In effect your program can be doing a number of things "at the same time". This allows you to take full advantage of modern operating system scheduling and multi-processor hardware. If I've already lost you or if any of the terminology below does not make sense, see manuals about POSIX threads (pthreads) before going any further. O'Reilly publishes a pretty good pthreads manual for example.

To use dmalloc with your threaded program, you will first need to make sure that you are linking with `libdmallocth.a` which is the threaded version of the library. The support for threads in dmalloc should be adequate for most if not all testing scenarios. It provides support for mutex locking itself to protect against race conditions that result in multiple simultaneous execution. One of the major problems is that most thread libraries uses malloc themselves. Since all of dmalloc's initialization happens when a call to malloc is made, we

may be attempting to initialize or lock the mutex while the thread library is booting up. A very bad thing since thread libraries don't expect to recurse.

The solution to this problem is to have the library not initialize or lock its mutex variable until after a certain number of allocation calls have been completed. If the library does not wait before initializing the locks, the thread library will probably core dump. If it waits too long then it can't protect itself from multiple execution and it will abort or other bad things might happen. You adjust the number of times to wait at runtime with the 'lock-on' option to the `dmalloc` program (for example `dmalloc -o 20`). See Chapter 4 [Dmalloc Program], page 35. Times values between 5 and 30 are probably good although operating systems will vary significantly. You know its too low if your program immediately core dumps and too high if the `dmalloc` library says its gone recursive although with low values, you might get either problem.

An additional complexity is when we are initializing the lock before mutex locking around the library. As mentioned, the initialization itself may generate a malloc call causing the library to go recursive and the pthread library to possibly core dump. With the `THREAD_INIT_LOCK` setting defined in `settings.h`, you can tune how many times before we start locking to try and initialize the mutex lock. It defaults to 2 which seems to work for me. If people need to have this runtime configurable or would like to present an alternative default, please let me know.

So to use `dmalloc` with a threaded program, follow the following steps carefully.

1. Follow the installation instructions on how to configure, make, and install the library but make sure to add the `--enable-threads` argument to configure. See Section 2.1 [Installation], page 5.
2. Typing `make` should be enough to build the threaded versions of the libraries including `libdmallocth.a`.
3. Link the `dmalloc` threaded library into your program. The `dmalloc` library should probably be placed at or near the end of the library list.
4. Enable the debugging options that you need by typing `dmalloc -l logfile -i 100 low` (for example). `dmalloc --usage` will provide verbose usage info for the `dmalloc` program. See Chapter 4 [Dmalloc Program], page 35.
5. Enable the "lock-on" option (for example `dmalloc -o 20`). As explained above, you may have to try different values before getting it right. Values between 5 and 30 are probably good.
6. If you get a `dmalloc` error #13 'thread locking has not been configured' then you have not compiled your program with the threaded version of `dmalloc` or there was a problem building it.
7. If everything works, you should be able to run your program, have it not immediately crash, and the `dmalloc` library should not complain about recursion.

If you have any specific questions or would like addition information posted in this section, please let me know. Experienced thread programmers only please.

3.11 Using the library with Cygwin environment.

The Cygwin environment is a Linux-like environment for Windows. It provides Linux look and feel as well as a programming environment. See URL <http://www.cygwin.com/> for more details.

Cygwin uses the `GetEnvironmentVariableA` function to read in environmental variables instead of `getenv`. This functions are used to get the value of the 'DMALLOC_OPTIONS' variable which sets the debugging options. See Section 4.3 [Environment Variable], page 40.

As of right now, `dmalloc` is not detecting the `GetEnvironmentVariableA` function correctly so you may need to tune the `conf.h` file to get it to work. See the sections on `HAVE_GETENVIRONMENTVARIABLEA` and `GETENV_SAVE` settings. Feedback is welcome here.

If you still have problems reading in the environmental variables, you can work around this issue. You can add some code into the `main` function in your program to initialize the `dmalloc` flags yourself. Here is a code sample:

```
main(int argc, char **argv)
{
#ifdef DMALLOC
/*
 * Get environ variable DMALLOC_OPTIONS and pass the settings string
 * on to dmalloc_debug_setup to setup the dmalloc debugging flags.
 */
    dmalloc_debug_setup(getenv("DMALLOC_OPTIONS"));
#endif

    /* rest of code in main starts here */
    ...
}
```

The `#ifdef` is just a good idea. I means that when debugging with `dmalloc` you need to compile your code with `-DDMALLOC`. When you are done debugging you can remove the flag and the call to `dmalloc_debug_setup` will be removed.

Please let me know if there is a better way to do this.

3.12 Debugging Memory in a Server or Cgi-Bin Process

There are some specified challenges when trying to debug allocations in processes which do not startup, run, and then shutdown. Server processes (often called daemons) are those that are started (often at system boot time) and run perpetually. Other processes which are difficult to debug are CGI programs which are spawned by web servers or when you want to start debugging inside of a child process.

1. Build your server or cgi-bin program with the `dmalloc` library like any other program. See Section 2.2 [Getting Started], page 6.
2. Add code into your program to enable the library flags to perform the memory checks that you require. Since these programs often do not run from the command line, you cannot use the `dmalloc` utility program and modify the process environment. See Chapter 4 [Dmalloc Program], page 35. The library provides a couple of functions to set the debugging flags when a program is running.

3. To set the memory debugging flags, use the `dmalloc_debug_setup` function which takes a string in the same format of the 'DMALLOC_OPTIONS' environmental variable. See Section 4.3 [Environment Variable], page 40. Use the `dmalloc` utility with the `-n no-changes` argument to see the appropriate settings for the 'DMALLOC_OPTIONS' environmental variable.

```
> dmalloc -n -l logfile high
Outputed:
DMALLOC_OPTIONS=debug=0x4f4ed03,log=logfile
export DMALLOC_OPTIONS
```

So if you want to turn on *high* debugging and log to the file `logfile` then you would copy the above 'DMALLOC_OPTIONS' value into a call to `dmalloc_debug_setup`. Notice that I have surrounded the `dmalloc` code with an `#ifdef DMALLOC` so you'll have to compile using the `-DDMALLOC` flag.

```
main()
{
#ifdef DMALLOC
    /* set the 'high' flags */
    dmalloc_debug_setup("debug=0x4f47d03,log=logfile");
#endif
    ...
}
```

Please note that the `dmalloc_debug_setup` function does not know about `high`, `low`, or other debug tokens but needs the actual flag values.

4. For earlier versions of the library (before 5.0.0) without `dmalloc_debug_setup`, the `dmalloc_debug` function is available to set the flags directly, but it cannot adjust the logfile name and the other environment settings. You can use the `dmalloc` utility program to see what the numerical equivalent of the *high* token.

```
> dmalloc -n high
Outputed:
DMALLOC_OPTIONS=debug=0x4f4ed03
export DMALLOC_OPTIONS
```

You can then take the `0x4f4ed03` hexadecimal number and call `dmalloc_debug` with that number.

```
main()
{
#ifdef DMALLOC
    /* set the 'high' flags */
    dmalloc_debug(0x4f4ed03);
#endif
    ...
}
```

5. Even with the settings enabled, you may have problems getting the logfile to be written if your program is running as 'nobody' or another user without permissions for security reasons. This is especially true for `cgi-bin` programs. In this case you should specify a full path to your malloc logfile in a world writable directory (ex. `dmalloc_`

`debug_setup("debug=0x4f47d03,log=/var/tmp/malloc");`). Watch for programs which change into other directories and which may cause logfiles specified as relative or local paths to be dropped in other locations. You may always want to use a full path logfile.

6. Once you have your settings enabled and your log is being generated, you may now want to check out how your process is doing in terms of unfreed memory. Since it is not shutting down, the automatic unfreed log entries are not being dropped to the logfile. By using the `dmalloc_mark` and `dmalloc_log_changed` functions, you can set a mark point at a certain place inside of your program, and then later see whether there are any unfreed pointers since the mark.

```

main()
{
#ifdef DMALLOC
    /* set the 'high' flags */
    dmalloc_debug_setup("debug=0x4f47d03,log=logfile");
#endif

    while (1) {
        /* accept a connection from a client */
        accept_connection();

        while (1) {
#ifdef DMALLOC
            unsigned long mark;
            /* get the current dmalloc position */
            mark = dmalloc_mark() ;
#endif
            /* process the connection */
            if (process_connection() != PROCESS_OK) {
                break;
            }
#ifdef DMALLOC
            /*
             * log unfreed pointers that have been added to
             * the heap since mark
             */
            dmalloc_log_changed(mark,
                                1 /* log unfreed pointers */,
                                0 /* do not log freed pointers */,
                                1 /* log each pnt otherwise summary */);
#endif
        }
        /* close the connection with the client */
        close_connection();
    }
    ...
}

```

```
}

```

Usually you would set the mark after the initializations and before each transaction is processed. Then for each transaction you can use `dmalloc_log_changed` to show the unfreed memory. See Section 3.5 [Extensions], page 17.

7. You can also use the `dmalloc_log_stats` function to dump general information about the heap. Also, remember that you can use the `dmalloc_message` and `dmalloc_vmessage` routines to annotate the dmalloc logfile with details to help you debug memory problems. See Section 3.5 [Extensions], page 17.

3.13 Explanation of the Logfile Output

Most of time you will be using the logfile output from library as the sole information source for diagnosing problems in and getting statistics for your program.

```
1098918225: 3: Dmalloc version 'Version 5.5.2'
1098918225: 3: flags = 0x4f4e503, logfile '/tmp/dmalloc.log'
1098918225: 3: interval = 500, addr = 0, seen # = 0, limit = 0
1098918225: 3: starting time = 1098918225
1098918225: 3: process pid = 32406
1098918226: 4: WARNING: tried to free(0) from foo.c:708'
1098918228: 20: *** free: at 'unknown' pnt '0xed310080|s2': \
      size 12, alloced at 'bar.c:102'
1098918230: 50: ERROR: heap_check: free space was overwritten (err 67)
1098918230: 50: error details: checking free pointer
1098918230: 50: pointer '0x291c5' from 'unknown' prev access 'foo.c:787'
```

Here is a short example of some logfile information. Each of the lines are prefixed by the time (in epoch seconds since 1/1/1970) and the iteration or call count which is the number of times the library has been called from malloc, free, verify, etc.. In the above example, the first 5 log entries were written at epoch 1098918225 or 'Wed Oct 27 19:03:45 2004 EST' and they were generated by the 3rd call to the library. See the `settings.dist` file entries to tune what elements appear on each line: `LOG_TIME_NUMBER`, `LOG_ITERATION`, `LOG_PID`, etc.. You can convert the epoch seconds to a date from the command line with the following perl code: `perl -e 'print localtime($ARGV[0])."\n";' epoch-seconds-number`

The first 5 lines of the sample logfile contain header information for all logfiles. They show the version number and URL for the library as well as all of the settings that the library is currently using. These settings are tuned using the dmalloc utility program. See Chapter 4 [Dmalloc Program], page 35. The 5th line of is the process-id that generated the logfile.

The 6th line in the above example is what causes the logfile to be opened and the header to be written. It is a warning that tells you that you tried to free a 0L pointer at a certain location. You can disable these warnings by setting '`ALLOW_FREE_NULL_MESSAGE`' to 0 in `settings.dist`.

Line 7 is an example of a transaction log that you get when you enable the `log-trans` debug token. See Section 4.4 [Debug Tokens], page 41. This line shows that a call to free was made from an unknown location. It is unknown because the file in question did not include `dmalloc.h` to get file/line-number information. The call to free was freeing the

pointer address `0xed310080` which we have “seen” 2 times (s2). We saw the pointer when it was allocated and then we are seeing it again when it was freed. Because the library is reusing pointers (reclaiming freed memory) the seen count helps to track how many times a pointer was used. The last part of the line shows that the pointer to be freed was allocated by `bar.c` line 102.

Lines 8-10 is the next problem that the library caught and this one is an error. It happened 5 seconds from the start of the log (1098918230) and at the 50th call into the library. It shows that an allocation that had been freed then was overwritten. This may imply that someone tried to use memory after it was freed or that there was a loose pointer reference. The last two lines give more details about when the error was discovered, the address of the offending pointer, and when the pointer was previous accessed, in this case freed. To discover where this problem is happening, you can use a debugger. See Section 3.9 [Using With a Debugger], page 25.

3.14 Various Other Hints That May Help

One of the problems that is often seen is that a program crashes in the `libc` memory code and you suspect a heap memory problem but both `dmalloc` and maybe `valgrind` don’t show any problems. One of the big problems with debugging is that it is very difficult to do it without effecting how the program is run. Sometimes errors are due to subtle race conditions that are only seen when the program is running at full speed – not slowed down by debugging code.

This is especially true with threaded code which is often heavily affected when used with `dmalloc` and `valgrind`. Older versions of `valgrind` (maybe current) forced all threads into a single virtual system by design, which often masks reentrance bugs.

One way to work through these issues is to run with the library with very few debugging flags enabled. Many memory problems are fence-post areas so start with `dmalloc` checking just the fence post and error logging enabled:

```
dmalloc -d 0 -l dmalloc.log -p log-stats -p log-non-free -p check-fence -p check-funcs
```

This enabled a small number of checks and should cause your program to run at close to full speed. The library has never been optimized for speed so some performance penalties will be felt.

4 Dmalloc Utility Program

The `dmalloc` program is designed to assist in the setting of the environment variable `'DMALLOC_OPTIONS'`. See Section 4.3 [Environment Variable], page 40. It is designed to print the shell commands necessary to make the appropriate changes to the environment. Unfortunately, it cannot make the changes on its own so the output from `dmalloc` should be sent through the `eval` shell command which will do the commands.

4.1 Using a Shell Alias with the Utility

The `dmalloc` program is designed to assist in the setting of the environment variable `'DMALLOC_OPTIONS'`. See Section 4.3 [Environment Variable], page 40. It is designed to print the shell commands necessary to make the appropriate changes to the environment. Unfortunately, it cannot make the changes on its own so the output from `dmalloc` should be sent through the `eval` shell command which will do the commands.

With shells that have aliasing or macro capabilities: `csh`, `bash`, `ksh`, `tcsh`, `zsh`, etc., setting up an alias to `dmalloc` to do the `eval` call is recommended. `Bash`, `ksh`, and `zsh` users should add the following to their `.bashrc`, `.profile`, or `.zshrc` file respectively (notice the `-b` option for bourne shell output):

```
function dmalloc { eval 'command dmalloc -b $*'; }
```

If your shell does not support the `command` function then try:

```
function dmalloc { eval '\dmalloc -b $*'; }
```

or

```
function dmalloc { eval '/usr/local/bin/dmalloc -b $*'; }
```

If you are *still* using `csh` or `tcsh`, you should add the following to your `.cshrc` file (notice the `-C` option for c-shell output):

```
alias dmalloc 'eval '\dmalloc -C \!*''
```

This allows the user to execute the `dmalloc` command as `'dmalloc arguments'`.

Users of versions of the Bourne shell (usually known as `/bin/sh`) that don't have command functions will need to send the output to a temporary file and then read it back in with the `."` command:

```
$ dmalloc -b arguments ... > /tmp/out
$ . /tmp/out
```

By the way, if you are looking for a shell, I heartily recommend trying out `zsh`. It is a bourne shell written from scratch with much the same features as `tcsh` without the `csh` crap.

NOTE: After you add the alias to the file you need to log out and log back in to have it take effect, or you can execute the above appropriate command on the command line. If you enter *dmalloc runtime* and see any output with `DMALLOC_OPTIONS` in it then the alias did not work.

4.2 How to Use the Dmalloc Program

The most basic usage for the program is `dmalloc [-bC] tag`. The `-b` or `-C` (either but not both flags used at a time) are for generating Bourne or C shell type commands respectively. `dmalloc` will try and use the `SHELL` environment variable to determine whether bourne or C shell commands should be generated but you may want to explicitly specify the correct flag.

The `tag` argument to `dmalloc` should match a line from the user's runtime configuration file or should be one of the built-in tags. See Section 4.5 [RC File], page 44. If no tag is specified and no other option-commands used, `dmalloc` will display the current settings of the environment variable. It is useful to specify one of the verbose options when doing this.

To find out the usage for the debug malloc program try `dmalloc --usage-long`. The standardized usage message that will be displayed is one of the many features of the `argv` library included with this package.

It is available on the web at URL <http://256.com/sources/argv/>. See the documentation there for more information.

Here is a detailed list of the flags that can be passed to `dmalloc`:

- `-a address` Set the `addr` part of the `DMALLOC_OPTIONS` variable to address (or alternatively address:number).
- `-b` Output Bourne shell type commands. Usually handled automatically.
- `-C` Output C shell type commands. Usually handled automatically.
- `-c` Clear/unset all of the settings not specified with other arguments. You can do this automatically when you set to a new tag with the `-r` option.
NOTE: clear will never unset the `debug` setting. Use `-d 0` or a tag to `none` to achieve this.
- `-d bitmask` Set the `debug` part of the `DMALLOC_OPTIONS` env variable to the bitmask value which should be in hex. This is overridden (and unnecessary) if a tag is specified.
- `-D` List all of the debug-tokens. Useful for finding a token to be used with the `-p` or `-m` options. Use with `-v` or `-V` verbose options.
- `-e errno` Print the `dmalloc` error string that corresponds to the error number `errno`.
- `-f filename` Use this configuration file instead of the RC file `$HOME/.dmalloccrc`.
- `-g` Output gdb type commands for using inside of the gdb debugger.
- `-h` (or `--help`) Output a help message for the utility.
- `-i number` Set the checking interval to number. If the `check-heap` token is enabled, this causes the library to only check the heap every Nth time which can *significantly* increase the running speed of your program. If a problem is found, however, this limits your ability to determine when the problem occurred. Try values of 50 or 100 initially.

-k Do not reset all of the settings when a tag is specified. This specifically overrides the **-r** option and is provided here to override **-r** if it has been added to the `dmalloc` alias.

-l filename

Write the debugging output and other log-file information to the filename. Filename can include some of the following patterns which get expanded into strings:

%h	Gets expanded into the hostname if the <code>gethostname()</code> function is available.
%i	Gets expanded into the thread-id if the library has been configured to be used with threads. See Section 3.10 [Using With Threads], page 28. See the end of the <code>settings.dist</code> file for settings which return the thread-id and convert it into a string.
%p	Gets expanded into the process-id if the <code>getpid()</code> function is available.
%t	Gets expanded into the time value in seconds if the <code>time()</code> function is available.
%u	Gets expanded into the user-id number if the <code>getuid()</code> function is available.

Some examples:

```
# logfile produced with pid extension:
# logfile.8412 or logfile.31451
dmalloc -l logfile.%p

# hostname and time extensions:
# dmalloc-box1.foo.com-1055213240
dmalloc -l dmalloc-%h-%t

# if threads enabled, have thread-id extension: log.thread32
dmalloc -l log.thread%i
```

-L Write the debug-value into the environment not in hex but by individual debug-tokens in long form.

-m token(s)

Remove (minus) the debug capabilities of token(s) from the current debug setting or from the selected tag (or **-d** value). Multiple **-m** options can be specified.

-M limit Set the memory allocation limit which will abort the program if the total memory allocations exceed this number of bytes. The limit can be a number with a k, m, or g at the end to indicate kilobyte, megabyte, and gigabyte respectively. Ex: 100k, 200m, 1g. If the limit is exceeded, this will generate an `ERROR_OVER_LIMIT` error. See Section 3.6 [Error Codes], page 21.

-n Without changing the environment, output the commands resulting from the supplied options.

- o times** Set the “lock-on” period which dictates to the threaded version of the library to not initialize or lock the mutex lock around the library until after a certain number of allocation calls have been made. Some number between 2 and 30 is probably good. See the “Using With Threads” section for more information about the operation of the library with threads. See Section 3.10 [Using With Threads], page 28.
- p token(s)** Add (plus) the debug capabilities of token(s) to the current debug setting or to the selected tag (or **-d** value). Multiple **-p** options can be specified.
- r** Remove (unset) all settings when using a tag. This is useful when you are returning to a standard development tag and want the logfile, address, and interval settings to be cleared automatically. If you want this behavior by default, this can be put into the dmalloc alias.
- R** Output rc shell type commands. This is not for the runtime configuration file but for the rc shell program.
- s file:line** Set the ‘start’ part of the ‘DMALLOC_OPTIONS’ env variable to a file-name and line-number location in the source where the library should begin more extensive heap checking. The file and line numbers for heap transactions must be working for this option to be obeyed. This is used if you are trying to locate a problem and you want the extensive checking to not happen initially because it’s too slow.
- S number** Set the ‘start’ part of the ‘DMALLOC_OPTIONS’ env variable to an dmalloc mark number. The library will begin more extensive heap checking after this number of memory transactions. If you LOG_ITERATION enabled in your settings.h file then the entries in the log file will be prepended with the number of memory transactions that the library has handled so far. This number can be used to delay the start of the fine grained heap checking which can be very slow.
- start-size size** Set the ‘start’ part of the ‘DMALLOC_OPTIONS’ env variable to a number of bytes. The library will begin more extensive heap checking after this amount of memory has been allocated by the library. This allows you to start the slow and detailed checking of the library later in the program execution. You can use patterns like 250m, 1g, or 102k to mean 250 megabytes, 1 gigabyte, and 102 kilobytes respectively.
- t** List all of the tags in the rc-file. Use with **-v** or **-V** verbose options.
- u (or --usage)** Output the usage information for the utility.
- v** Give verbose output. Especially useful when dumping current settings or listing all of the tags.
- V** Give very verbose output for outputting even more details about settings.

--version

Output the version string for the utility. *Please note* that the version of the library that is installed or has been linked into your application may be different from the utility version.

If no arguments are specified, dmalloc dumps out the current settings that you have for the environment variable. For example:

```
Debug-Flags  '0x40005c7' (runtime)
Address      0x1f008, count = 3
Interval     100
Logpath      'malloc'
Start-File   not-set
```

With a -v option and no arguments, dmalloc dumps out the current settings in a verbose manner. For example:

```
Debug-Flags  '0x40005c7' (runtime)
  log-stats, log-non-free, log-bad-space, check-fence, catch-null
Address      0x1f008, count = 10
Interval     100
Logpath      'malloc'
Start-File   not-set
```

Here are some examples of dmalloc usage:

```
# start tough debugging, check the heap every 100 times,
# send the log information to file 'logfile'
dmalloc high -i 100 -l logfile

# find out what error code 20 is (from the logfile)
dmalloc -e 20

# cause the library to halt itself when it sees the address 0x34238
# for the 6th time.
dmalloc -a 0x34238:6

# send the log information to file 'logfile' with the time in seconds
# as an extension.
dmalloc -l logfile.%t

# return to the normal 'runtime' settings and clear out all
# other settings
dmalloc -c runtime

# enable basic 'low' settings plus (-p) the logging of
# transactions (log-trans) to file 'logfile'
dmalloc low -p log-trans -l logfile

# print out the current settings with Very-verbose output
dmalloc -V
```

```
# list the available debug malloc tokens with Very-verbose output
dmalloc -DV
```

```
# list the available tags from the rc file with verbose output
dmalloc -tv
```

4.3 Environment Variable Name and Features

An *environment variable* is a variable that is part of the user's working environment and is shared by all the programs. The 'DMALLOC_OPTIONS' variable is used by the dmalloc library to enable or disable the memory debugging features, at runtime. *NOTE:* you can also use the `dmalloc_debug_setup` function to set the option string. It can be set either by hand or with the help of the dmalloc program. See Chapter 4 [Dmalloc Program], page 35.

To set it by hand, Bourne shell (sh, bash, ksh, or zsh) users should use:

```
DMALLOC_OPTIONS=value
export DMALLOC_OPTIONS
```

C shell (csh or tcsh) users need to invoke:

```
setenv DMALLOC_OPTIONS value
```

The value in the above examples is a comma separated list of tokens each having a corresponding value. The tokens are described below:

debug This should be set to a value in hexadecimal which corresponds to the functionality token values added together. See Section 4.4 [Debug Tokens], page 41. For instance, if the user wanted to enable the logging of memory transactions (value '0x008') and wanted to check fence-post memory (value '0x400') then 'debug' should be set to '0x408' ('0x008' + '0x400').

NOTE: You don't have to worry about remembering all the hex values of the tokens because the dmalloc program automates the setting of this variable especially.

NOTE: You can also specify the debug tokens directly, separated by commas. See Section 4.4 [Debug Tokens], page 41. If 'debug' and the tokens are both used, the token values will be added to the debug value.

lockon Set this to a number which is the "lock-on" period. This dictates to the threaded version of the library to not initialize or lock the mutex lock around the library until after a certain number of allocation calls have been made. See the "Using With Threads" section for more information about the operation of the library with threads. See Section 3.10 [Using With Threads], page 28.

log Set this to a filename so that if 'debug' has logging enabled, the library can log transactions, administration information, and/or errors to the file so memory problems and usage can be tracked.

To get different logfiles for different processes, you can assign 'log' to a string with %d in it (for instance 'logfile.%d'). This will be replaced with the pid of the running process (for instance 'logfile.2451').

WARNING: it is easy to core dump any program with dmalloc, if you send in a format with arguments other than the one %d.

- addr** When this is set to a hex address (taken from the dmalloc log-file for instance) dmalloc will abort when it finds itself either allocating or freeing that address. The address can also have an `:number` argument. For instance, if it was set it to `'0x3e45:10'`, the library will kill itself the 10th time it sees address `'0x3e45'`. By setting the number argument to 0, the program will never stop when it sees the address. This is useful for logging all activity on the address and makes it easier to track down specific addresses not being freed.
- This works well in conjunction with the `STORE_SEEN_COUNT` option. See Section 3.9.2 [Memory Leaks], page 26.
- NOTE:* dmalloc will also log all activity on this address along with a count.
- inter** By setting this to a number X, dmalloc will only check the heap every X times. This means a number of debugging features can be enabled while still running the program within a finite amount of time.
- A setting of `'100'` works well with reasonably memory intensive programs. This of course means that the library will not catch errors exactly when they happen but possibly 100 library calls later.
- start** Set this to a number X and dmalloc will begin checking the heap after X times. This means the intensive debugging can be started after a certain point in a program.
- `'start'` also has the format `'file:line'`. For instance, if it is set to `'dmalloc_t.c:126'` dmalloc will start checking the heap after it sees a dmalloc call from the `dmalloc_t.c` file, line number 126. If you use `'dmalloc_t.c:0'`, with a 0 line number, then dmalloc will start checking the heap after it sees a call from anywhere in the `dmalloc_t.c` file.
- This allows the intensive debugging to be started after a certain routine or file has been reached in the program.

Some examples are:

```
# turn on transaction and stats logging and set
# 'logfile' as the log-file
setenv DMALLOC_OPTIONS log-trans,log-stats,log=logfile

# enable debug flags 0x1f as well as heap-checking and
# set the interval to be 100
setenv DMALLOC_OPTIONS debug=0x1f,check-heap,inter=100

# enable 'logfile' as the log-file, watch for
# address '0x1234', and start checking when we see
# file.c line 123
setenv DMALLOC_OPTIONS log=logfile,addr=0x1234,start=file.c:123
```

4.4 Description of the Debugging Tokens

The below tokens and their corresponding descriptions are for the setting of the debug library setting in the environment variable. See Section 4.3 [Environment Variable], page 40. They should be specified in the user's `.dmallocrc` file. See Section 4.5 [RC File], page 44.

Each token, when specified, enables a specific debugging feature. For instance, if you have the `log-stats` token enabled, the library will log general statistics to the logfile.

To get this information on the fly, use `dmalloc -DV`. This will print out the Debug tokens in Very-verbose mode. See Chapter 4 [Dmalloc Program], page 35.

- `none` No debugging functionality
- `log-stats`
 Log general statistics when `dmalloc_shutdown` or `dmalloc_log_stats` is called.
- `log-non-free`
 Log non-freed memory pointers when `dmalloc_shutdown` or `dmalloc_log_unfreed` is called.
- `log-known`
 Log only known memory pointers that have not been freed. Pointers which do not have file/line or return-address information will not be logged.
- `log-trans`
 Log general memory transactions (quite verbose).
- `log-admin`
 Log administrative information (quite verbose).
- `log-bad-space`
 Log actual bytes in and around bad pointers.
- `log-nonfree-space`
 Log actual bytes in non-freed pointers.
- `log-elapsed-time`
 Log elapsed-time for allocated pointers (see `conf.h`).
- `log-current-time`
 Log current-time for allocated pointers (see `conf.h`).
- `check-fence`
 Check fence-post memory areas.
- `check-heap`
 Verify heap administrative structure.
- `check-blank`
 Check to see if space that was blanked when a pointer was allocated or when it was freed has been overwritten. If this is enabled then it will enable `free-blank` and `alloc-blank` automatically.
- `check-funcs`
 Check the arguments of some functions (mostly string operations) looking for bad pointers.
- `check-shutdown`
 Check all of the pointers in the heap when the program exits.

catch-signals

Shutdown the library automatically on SIGHUP, SIGINT, or SIGTERM. This will cause the library to dump its statistics (if requested) when you press control-c on the program (for example).

realloc-copy

Always copy data to a new pointer when realloc.

free-blank

Write special “dmalloc-free” byte (hexadecimal 0xdf, octal 0337, decimal 223) into space when it is freed. You can set this to be something else in the `settings.dist` file. This ensures that your program is not using memory after it has been freed. You can check to see if areas have been improperly overwritten with the `check-blank` token. If the free space has been overwritten, then `ERROR_FREE_OVERWRITTEN` is triggered. See Section 3.6 [Error Codes], page 21.

error-abort

Abort the program (and dump core) on errors. See `error-dump` below. See Section 3.4 [Dumping Core], page 16.

alloc-blank

Write special “dmalloc-alloc” byte (hexadecimal 0xda, octal 0332, decimal 218) into space when it is allocated. You can set this to be something else in the `settings.dist` file. If you are not using `calloc` this will overwrite the user space with the special bytes ensuring that your program is initializing its dynamic memory appropriately. Also, if you ask for 35 bytes and the library has to give you a block of 64 because of rounding issues, it will overwrite the extra memory with the special byte. You can then check to see if the extra areas have been improperly overwritten by enabling the `check-blank` token.

print-messages

Log any errors and messages to the screen via standard-error.

catch-null

Abort the program immediately if the library fails to get more heap space from the heap allocation routine `mmap` or `sbrk`.

never-reuse

Have the heap never use space that has been used before and freed. See Section 3.9.2 [Memory Leaks], page 26. *WARNING*: This should be used with caution since you may run out of heap space.

error-dump

Dump core on error and then continue. Later core dumps overwrite earlier ones if the program encounters more than one error. See `error-abort` above. See Section 3.4 [Dumping Core], page 16.

NOTE: This will only work if your system supports the `fork` system call and the configuration utility was able to fork without going recursive.

error-free-null

By default the library will not generate an error when a program tries to free a NULL pointer. By enabling this token, you can change this behav-

ior so an error is reported. See also the `ALLOW_FREE_NULL` and `ALLOW_FREE_NULL_MESSAGE` settings in the `settings.h` file to change the default behavior.

4.5 Format of the Runtime Configuration File

By using a *RC File* (or runtime configuration file) you can alias tags to combinations of debug tokens. See Section 4.4 [Debug Tokens], page 41.

NOTE: For beginning users, the `dmalloc` program has a couple of tags built into it so it is not necessary for you to setup a RC file:

<code>runtime</code>	Enables basic runtime tests including fence-post checking, null handling, and logging of any errors.
<code>low</code>	Runtime settings plus minimal checking of heap structures and overwriting of allocated and freed space.
<code>medium</code>	Low settings plus checking of all heap structures on each memory call, always relocates block on realloc, and aborts on errors. You may want to use <code>-i</code> option to the <code>dmalloc</code> utility. See Chapter 4 [Dmalloc Program], page 35.
<code>high</code>	Medium settings plus checking of overwritten freed and allocated memory and checking of arguments to a number of common functions. You may want to use <code>-i</code> option to the <code>dmalloc</code> utility. See Chapter 4 [Dmalloc Program], page 35.

For expert users, a sample `dmallocrc` file has been provided but you are encouraged to roll your own combinations. The name of default rc-file is `$HOME/.dmallocrc`. The `'$HOME'` environment variable should be set by the system to point to your home-directory.

The file should contain lines in the general form of:

```
tag      token1, token2, ...
```

`'tag'` is to be matched with the tag argument passed to the `dmalloc` program, while `'token1, token2, ...'` are debug capability tokens. See Chapter 4 [Dmalloc Program], page 35, Section 4.4 [Debug Tokens], page 41.

A line can be finished with a `'\'` meaning it continues onto the next line. Lines beginning with `'#'` are treated as comments and are ignored along with empty lines.

Here is an example of a `.dmallocrc` file:

```
#
# Dmalloc runtime configuration file for the debug malloc library
#

# no debugging
none      none

# basic debugging
debug1    log-stats, log-non-free, check-fence

# more logging and some heap checking
debug2    log-stats, log-non-free, log-trans, \
```

```
        check-fence, check-heap, error-abort

# good utilities
debug3  log-stats, log-non-free, log-trans, \
        log-admin, check-fence, check-heap, realloc-copy, \
        free-blank, error-abort

...
```

For example, with the above file installed, you can type `dmalloc debug1` after setting up your shell alias. See Chapter 4 [Dmalloc Program], page 35. This enables the logging of statistics, the logging of non-freed memory, and the checking of fence-post memory areas.

Enter `dmalloc none` to disable all memory debugging features.

5 Information on the Source Code

5.1 Definition of Terms and other Information

Here are a couple definitions and other information for those interested in “picking the brain” of the library. The code is a little ugly here and there and it conforms to the Gray-Watson handbook of coding standards only.

bblock basic block containing 2^{\wedge} BASIC_BLOCK bytes of info

bblock_adm
 administration for a set of basic blocks

dblock divided block containing some base 2 number of blocks smaller than a basic block.

dblock_adm
 administration for a set of divided blocks

chunk some anonymous amount of memory

For more information about administration structures, see the code and comments from `chunk_loc.h`.

5.2 General Compatibility Concerns

- `Realloc()` backwards compatibility with being able to realloc from the last freed block is *not* supported. The author is interested to know who is using this (cough, cough) feature and for what reason.
- `Realloc()` of a NULL pointer is supported in which case the library will just make a call to `malloc()`. This can be disabled with the help of the `ALLOW_REALLOC_NULL` manual compilation option in the `settings.h` file to adjust the library’s default behavior.
- Some systems allow `free(0)` to not be an error for some reason. Since 0 is not a valid address returned by the `malloc` call, it is debatable that this should be allowed. See `settings.h` for the `ALLOW_FREE_NULL` manual compilation option to adjust the library’s default behavior.
- Aside from possibly being slower than the system’s memory allocation functions, the library should be fully compatible with the standard memory routines. If this is *not* the case, please bring this to my attention.

5.3 Issues Important for Porting the Library

General portability issues center around:

- `mmap`, `sbrk`, or compatible function usages. The library does support a preallocated memory chunk heap. See the `INTERNAL_MEMORY_SPACE` define in the `settings.dist` file.
- The locating of the caller’s address from the `dmalloc` functions. This is useful in locating problems from `dmalloc` functions called from C files which did not include `dmalloc.h`: C library calls for instance.

See `return.h` for the available architecture/compiler combinations. You may want to examine the assembly code from `gcc` (GNU's superior c-compiler) version 2+ being run on the following code. It should give you a good start on building a hack for your box.

```
static char * x;

a()
{
    x = __builtin_return_address(0);
}

main()
{
    a();
}
```

6 Some Solutions to Common Problems

This section provides some answers to some common problems and questions. Please send me mail with any additions to this list – either problems you are still having or tips that you would like to pass on.

When diagnosing a problem, if possible, always make sure you are running the most up to date version of Dmalloc available from the home page at URL <http://dmalloc.com/>. Problems are often fixed and a new release can be published before people encounter them.

‘Why does my program run so slow?’

This library has never been (and maybe never will be) optimized for space nor speed. Some of its features make it unable to use some of the organizational methods of other more efficient heap libraries.

If you have the `check-heap` token enabled, your program might run slow or seem to hang. This is because by default, the library will run a full check of the heap with every memory allocation or free. You can have the library check itself less frequently by using the `-i` option to the `dmalloc` utility. See Chapter 4 [Dmalloc Program], page 35. If you are using the `high` token and you need your program to run faster, try the `medium` or `low` tokens which don’t check as many heap features and so run faster although they will not catch as many problems. See Section 4.5 [RC File], page 44.

‘Why was a log-file not produced after I ran my program?’

This could be caused by a number of different problems.

1. Are you sure you followed all of the items in the “Getting Started” section? Please review them if there is any doubt. See Section 2.2 [Getting Started], page 6.
2. Use the `env` or `printenv` commands to make sure that the ‘DMALLOC_OPTIONS’ variable is set in your exported environment. See Section 4.3 [Environment Variable], page 40.
3. Make sure that your program has been compiled correctly with the `dmalloc` library. The `ident` program should show `chunk.c` and other `dmalloc` files compiled into your program. You can also do `strings -a your-program | grep chunk.c` and look for something like ‘\$Id: chunk.c,v 1.152 1999/08/25 12:37:01 gray Exp \$’ with different versions and date information. If this doesn’t show up then chances are `dmalloc` was not linked into your program.
4. If your program changes its working directory, it may write the `dmalloc` log-file somewhere else in the filesystem. You will need to check both where the program was started and to where it might change directory.
5. The logfile is only produced when `dmalloc_shutdown()` is called. By default it will be called when `exit()` gets called. If you are running your program and press `Control-C` under Unix the program will stop immediately and `dmalloc_shutdown()` will not get called. You can either setup a signal handler for `SIGINTR` and call `exit` yourself, or you can enable the `catch-signals` token. See Section 4.4 [Debug Tokens], page 41.

6. If your program is segfaulting or otherwise crashing when it exits, the `exit()` routine may not be called. You will have to resolve these issues so the dmalloc library can gracefully exit and write its log file.
7. You may want to call `dmalloc_log_stats()` and `dmalloc_log_unfreed()` (or `dmalloc_log_changed()`) directly to have the library write its log file. Some system modules may not have shutdown if you call this before `exit()` so extra unfreed memory may be reported.

‘I don’t see any information about my non-freed (leaked) memory?’

The library will not (by default) report on “unknown” non-freed memory. Unknown means memory that does not have associated file and line information.

This will be necessary if you are *not* including `dmalloc.h` in all of your C files or if you are interested in tracking leaks in system functions.

‘Dmalloc is returning the error "malloc library has gone recursive"’

This most likely indicates that you are using the Dmalloc library within a threaded application and two threads are trying to use the dmalloc library at once. Please see the section of the manual about threads for more information about properly configuring the library. See Section 3.10 [Using With Threads], page 28.

If you are not using threads, then your program could have caught a signal while within Dmalloc, which then in turn called a memory allocation routine. It is unwise to allocate memory on the heap in most signal handlers. Lastly, some functions called by the library may call memory routines that it does not anticipate. If you think this the case, please report the problem and include a stack trace, operating system version/type, and the version of Dmalloc you are using.

Index of Concepts

%

%h	37
%i	37
%p	37
%t	37
%u	37

—

—disable-cxx	5
—enable-shlib	5
—enable-threads	5

.

.dmallocrc file	44
-----------------------	----

0

0332 character	43
0337 character	43
0xda character	43
0xdf character	43

1

1, error code	21
10, error code	21
11, error code	21
13, error code	21

2

2, error code	21
20, error code	21
21, error code	21
218 character	43
22, error code	21
223 character	43
23, error code	22
24, error code	22
25, error code	22
26, error code	22
27, error code	22
28, error code	22

3

30, error code	22
332 character	43
337 character	43

4

40, error code	22
41, error code	23
43, error code	23
45, error code	23

6

60, error code	23
61, error code	23
67, error code	23

7

70, error code	23
72, error code	23
73, error code	24

A

abort	17
address list error	23
address locating	41
address setting	41
admin list error	23
alias, shell	6, 35
alloc failed error	23
alloc-blank	43
allocation basics	8
allocation macros	15
Allocation of zeros	10
ALLOW_FREE_NULL	43
ALLOW_FREE_NULL settings.h option	47
ALLOW_FREE_NULL_MESSAGE	33, 43
ALLOW_REALLOC_NULL settings.h option	47
already free error	23
ANSI-C compiler	5
argument checking	16
assembly hacks	15
atexit	6
atexit function	17
author	1
automatic shutdown	6

B

bad admin structure list	23
bad file error	22
bad line error	22
bad setup error	21
bad size error	22
bash shell	6, 35
bash usage	40
basic allocation information	8
basic definitions	8
beginning	6
blank space	43
blanking memory	43
bounds checking	11
Bourne shell usage	40
building the library	5

C

c++ usage	25
C shell usage	40
caller address translation	28
caller's address	15
calloc	10
cannot locate pointer in heap	21
catch-null	43
catch-signals	42
cgi-bin process debugging	30
cgi-bin usage of dmalloc	30
changed memory log	20
check-blank	23, 42
check-fence	42
check-funcs	42
check-heap	12, 42, 49
check-shutdown	42
checking arguments	16
checking bounds	11
checkpoint memory usage	20
child process debugging	30
clearing memory	43
comma separated tokens in env variable	40
common problems	49
compatibility	47
compiling the library	5
conf.h file	5
configuration file	44
configure script	5
configuring the library	5
constancy verification	11
copying	3
core dump	12, 16, 43
core dump, none	16
core file	16
could not grow heap by allocating memory	23
count changed	19
count number of bytes changed since mark	19
cpp	10
csh shell	7, 35

csh usage	40
current debug value	18
cygwin	30

D

da character	43
daemon process debugging	30
debug setting	40
debug tokens	41
debugger	16
debugger usage with dmalloc	25
debugging cgi-bin processes	30
debugging child processes	30
debugging daemon processes	30
debugging server processes	30
decimal 218 character	43
decimal 223 character	43
delay heap checking	38
destructor	6
df character	43
diagnosing errors	26
disabling library checking	24
disabling the library	24
dmalloc program	35
dmalloc utility	35
dmalloc.h file	15
dmalloc_debug	24
dmalloc_debug function	18
dmalloc_debug_current function	18
dmalloc_debug_setup	18
dmalloc_debug_setup function	30
dmalloc_debug_setup usage	30
dmalloc_errno number	17
dmalloc_error() routine	25
dmalloc_examine function	18
dmalloc_get_stats function	20
dmalloc_log_changed function	20
dmalloc_log_stats function	20
dmalloc_log_unfreed function	20
dmalloc_logpath variable	17
dmalloc_mark function	19
dmalloc_memory_allocated function	19
dmalloc_message function	20
dmalloc_shutdown function	17
dmalloc_strerror function	20
dmalloc_t test program	6
dmalloc_tag function	19
dmalloc_track function	19
dmalloc_verify function	18
dmalloc_verify() routine	26
dmalloc_vmessage function	20
DMALLOC_FUNC_CHECK	7
DMALLOC_FUNC_CHECK flag	16
DMALLOC_OPTIONS	40, 49
DMALLOC_SIZE option	5
dmalocc.cc	25
dmalocc.cc file	25

dmallocrc file 44
 dump core 12, 16, 43

E

env 49
 environment variable 40
 errno value is not valid 21
 error code 10 21
 error code 11 21
 error code 13 21
 error code 20 21
 error code 21 21
 error code 22 21
 error code 23 22
 error code 24 22
 error code 25 22
 error code 26 22
 error code 27 22
 error code 28 22
 error code 30 22
 error code 40 22
 error code 41 23
 error code 43 23
 error code 45 23
 error code 60 23
 error code 61 23
 error code 67 23
 error code 70 23
 error code 72 23
 error code 73 24
 error code, 1 21
 error code, 2 21
 error codes 21
 error message 20
 error number 17
 error-abort 16, 43
 error-dump 16, 43
 error-free-null 43
 error-free-null token 21
 ERROR_ADDRESS_LIST 23
 ERROR_ADMIN_LIST 23
 ERROR_ALLOC_FAILED 23
 ERROR_ALREADY_FREE 23
 ERROR_BAD_FILE 22
 ERROR_BAD_LINE 22
 ERROR_BAD_SETUP 21
 ERROR_BAD_SIZE 22
 ERROR_FREE_OVERWRITTEN 23, 43
 ERROR_IN_TWICE 21
 ERROR_IS_FOUND 22
 ERROR_IS_NULL 21
 ERROR_LOCK_NOT_CONFIG 21
 ERROR_NONE 21
 ERROR_NOT_FOUND 21
 ERROR_NOT_IN_HEAP 21
 ERROR_NOT_ON_BLOCK 23
 ERROR_NOT_START_BLOCK 22

ERROR_OVER_FENCE 22
 ERROR_OVER_LIMIT 23, 37
 ERROR_SLOT_CORRUPT 24
 ERROR_TOO_BIG 23
 ERROR_UNDER_FENCE 22
 ERROR_WOULD_OVERWRITE 22
 examine a pointer 18
 extensions 17

F

failed over picket-fence magic-number check 22
 failed under picket-fence magic-number check ... 22
 faq 49
 features 10
 fence-post checking 11
 fence-post errors 27
 file/line numbers 10
 found pointer the user was looking for 22
 free 10
 free null token 21
 free overwritten error 23
 free space has been overwritten 23
 free-blank 23, 43
 freed memory 11

G

gcc 47
 gdb 25
 gdb with shared libraries 26
 general errors 26
 generating core dump 16
 get heap statistics 20
 getenv, problems with 30
 GETENV_SAVE conf.h option 30
 GetEnvironmentVariableA, using with 30
 gethostname function usage 37
 getpid function usage 37
 getting started 6
 getuid function usage 37

H

hanging program 49
 HAVE_GETENVIRONMENTVARIABLEA 30
 heap memory 9
 heap statistics function 20
 help 49
 hexadecimal 0xda character 43
 hexadecimal 0xdf character 43
 high token 44
 hints that may help 34
 hostname in logfile path 37
 how do i 49
 how to begin 6
 HUP signal 42

I

ident	49
in twice error	21
initialization and setup failed	21
installing the library	5
INT signal	42
interaction count	19, 38
internal address list corruption	23
internal error number	17
internal memory slot corruption	24
INTERNAL_MEMORY_SPACE	47
interval setting	36, 41
introduction	1
invalid allocation size	22
invalid errno value	21
INVALID_ERROR	21
is found error	22
is null error	21
iteration count	12

J

jump start	6
------------------	---

K

kill process	17
KILL_PROCESS	17
ksh shell	6, 35
ksh usage	40

L

largest maximum allocation size exceeded	23
leaking memory	11, 26
libdmallocxx.a	25
library permissions	3
library utility	35
library version	38
license	3
lock on	29, 37
lock-on	21
lock-on not configured	21
lockon setting	40
log memory changes	20
log statistics	20
log unfreed memory	20
log-admin	42
log-bad-space	42
log-current-time	42
log-elapsed-time	42
log-known	42
log-non-free	42
log-nonfree-space	42
log-stats	42
log-trans	42
LOG_ITERATION	19, 38
LOG_ITERATION_COUNT	12

logfile format	33
logfile message writer	20
logfile name	17
logfile not produced	49
logfile setting	40
logging information to disk	40
logging statistics	11
low token	44

M

macros, allocation	15
making the library	5
malloc	9
malloc functions	9
malloc library has gone recursive	21
mark count	38
mark memory position	19
medium token	44
memalign	15
memory allocated function	19
memory definitions	8
memory leaks	11, 26
memory limit	23, 37
memory position marker	19
memory problems in system functions	12
memory transaction count	12, 38
missing core dump	16
mmap	21, 23, 26, 43
mmap, usage without	47

N

name of host in logfile path	37
never-reuse	43
no core dump	16
no error	21
no logfile produced	49
none token	42
not found error	21
not in heap error	21
not on block boundary error	23
not start block error	22
number bytes changed since mark	19
number of bytes allocated	19

O

octal 332 character	43
octal 337 character	43
on_exit	6
on_exit function	17
other hints	34
over fence error	22
over limit error	23
over user specified allocation limit error	23
override debug settings	18

overview 5
 overwriting memory 43

P

page size 19
 permissions of the library 3
 pid in logfile path 37
 pointer information 18
 pointer is not on block boundary 23
 pointer is not pointing to heap data space 21
 pointer is not to start of memory block 22
 pointer is null error 21
 pointer not found error 21
 pointer not in heap error 21
 pointer seen count 27
 portability 47
 posix_memalign 15
 possibly bad .c file line-number 22
 possibly bad .c filename pointer 22
 preallocated memory heap 47
 print-messages 43
 printenv 49
 problems 49
 process-id in logfile path 37
 program core 16
 programming 15
 pthreads 28

Q

questions 49
 quick start 6

R

ra 15
 ra_info.pl 16, 28
 rc file 44
 rc shell 7
 reading environment, problems with 30
 realloc 10
 realloc-copy 43
 realloc 15
 recursion 21
 report heap statistics 20
 restoring library flags 24
 return-address 10, 15, 47
 return-address translation 28
 return.h file 15
 runtime token 44
 runtime-config file 44

S

sbrk 21, 26, 43
 sbrk, usage without 47
 seen count 18, 27, 33
 server process debugging 30
 set debug functionality flags 18
 settings.dist file 5
 settings.h file 5
 setup debug flags 18
 sh usage 40
 shared libraries with gdb 26
 shell usage 40
 shutdown on signal 42
 shutdown the library 17
 shutdown, automatic 6
 SIGHUP 42
 SIGINT 42
 signal shutdown 42
 SIGTERM 42
 size of memory pages 19
 slot corrupt error 24
 slow running 49
 source code 47
 source definitions 47
 stack memory 8
 start heap check later 38
 start setting 41
 static memory 8
 statistics 11
 statistics logging 20
 STORE_SEEN_COUNT conf.h option 27
 strdup 15
 string error message 20
 string of debug tokens 18
 strings 49
 system memory problems 12

T

tag a pointer 19
 tcsh shell 7, 35
 tcsh usage 40
 TERM signal 42
 testing the library 6
 thread locking has not been configured 21
 thread-id in logfile path 37
 threads 28
 time function usage 37
 time in logfile path 37
 token high 44
 token low 44
 token medium 44
 token runtime 44
 tokens, debug 41
 too big error 23
 too slow 49
 track memory calls 19
 tracking addresses 41

transaction count	19, 38
tried to free previously freed pointer	23
troubleshooting	49
turning off library flags	24

U

uid in logfile path	37
under fence error	22
unfreed memory log	20
usage of dmalloc in a daemon	30
usage of dmalloc in a server	30
usage of dmalloc with cgi-bin	30
usage of the utility	36
use of pointer would exceed allocation	22
USE_MMAP conf.h option	23
USE_RETURN_MACROS conf.h option	5
user-id in logfile path	37
using a debugger with dmalloc	25
utility program	35
utility usage	36
utility version	38

V

valloc	15
various hints	34
verify pointers	18
verify the heap	18
version of library	33
version of utility	38

W

where to begin	6
why hanging	49
why running slow	49
would overwrite error	22
write message to logfile	20

Z

zeros, allocation of	10
zsh shell	6, 35
zsh usage	40

Table of Contents

Debug Malloc Library	1
1 Library Copying and Licensing Conditions.....	3
2 Description of Features and How to Get Started ..	5
2.1 How to Install the Library	5
2.2 Getting Started with the Library	6
2.3 Basic Description of Terms and Functions.....	8
2.3.1 General Memory Terms and Concepts.....	8
2.3.2 Functionality Supported by All Malloc Libraries.....	9
2.4 General Features of the Library.....	10
2.5 How the Library Checks Your Program	12
3 How to Program with the Library.....	15
3.1 Macros Providing File and Line Information	15
3.2 Getting Caller Address Information.....	15
3.3 Checking of Function Arguments	16
3.4 Generating a Core File on Errors	16
3.5 Additional Non-standard Routines.....	17
3.6 Description of the Internal Error Codes	21
3.7 How to Disable the library.....	24
3.8 Using the Library with C++	25
3.9 Using Dmalloc With a Debugger.....	25
3.9.1 Diagnosing General Problems with a Debugger.....	26
3.9.2 Tracking Down Non-Freed Memory	26
3.9.3 Diagnosing Fence-Post Overwritten Memory	27
3.9.4 Translating Return Addresses into Code Locations	28
3.10 Using the Library with a Thread Package.....	28
3.11 Using the library with Cygwin environment.....	30
3.12 Debugging Memory in a Server or Cgi-Bin Process.....	30
3.13 Explanation of the Logfile Output	33
3.14 Various Other Hints That May Help	34
4 Dmalloc Utility Program.....	35
4.1 Using a Shell Alias with the Utility	35
4.2 How to Use the Dmalloc Program	36
4.3 Environment Variable Name and Features	40
4.4 Description of the Debugging Tokens	41
4.5 Format of the Runtime Configuration File	44

5	Information on the Source Code	47
5.1	Definition of Terms and other Information	47
5.2	General Compatibility Concerns	47
5.3	Issues Important for Porting the Library	47
6	Some Solutions to Common Problems	49
	Index of Concepts	51