Debug Malloc Library

Version 3.1.3
December 1995

Gray Watson <gray@letters.com>

Published by Gray Watson '<gray@letters.com>'

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, BSDI, DG/UX, HPUX, Irix, Linux, MS-DOG, NeXT, OSF, Solaris, SunOS, Ultrix, Unixware, and even Unicos on a Cray Y-MP.

The package includes the library, configuration scripts, debug utility application, test program, and extensive documentation (formats: text, texi, info, and ps). Documentation is available online at URL `http://www.letters.com/dmalloc/`.

The library is available via ftp from '`ftp.letters.com`' in the '`/src/dmalloc`' directory. See Section 2.3 [How To Get], page 9. I can be reached at '`gray@letters.com`' with any questions, feedback, or general comments. Please include the version number of the library that you are using as well as your machine and operating system types.

Gray Watson.

1 Library Copying and Licensing Conditions

Please see the following sections for usage permissions.

1.1 Non-Commercial License

Permission to use, copy, and modify this software for any *academic* or *non-commercial* purpose without fee is hereby granted.

Although only commercial users are required to license the software, *all* registrations are greatly appreciated. Licensees get a registration file for the software (which cannot be distributed), a letter documenting their registered status, as well as product support and notification of free-upgrades. For more information about registration see the following section. See Section 1.2 [Commercial License], page 2.

1.2 Commercial License

In order to fund maintenance and continued development of this product, the requirement has been added that commercial entities license the software for a nominal fee. There is a thirty (30) day free evaluation period but if your organization uses the software after this time period expires, please consider purchasing a registered copy. This allows me to provide better support as well as bug fixes and feature updates on a more frequent basis. Although not required, academic or non-commercial user registration is greatly appreciated.

A licensed copy of the software is available for US$35. Commercial entities should purchase one license per workstation, or one per dmalloc user, whichever is the smaller number. Licensees get a registration file for the software (which cannot be distributed), a letter documenting their registered status, as well as product support and notification of free-upgrades.

For "site" licenses (at significant discount) or any other reasonable agreements, please contact me directly. It should be noted that a "site" can be defined as anything you'd like. It can be a physical location (a room, building, etc.), an organizational grouping (a workgroup, department, etc.) or any other logical grouping ("the folks working on the widget project", etc.).

1.3 Printed Manuals

A printed and bound copy of the manual is available for US$15. A copy of the software on a Unix tar floppy is available for US$15. International shipping costs outside of the US or Canada, please add US$5 for the extra postal charges.

1.4 Registration Information

Please include the following registration information with your check made out to "Gray Watson". Money orders are also accepted. Purchase orders are as well, although amounts less than US$50 are discouraged.

- Name
- Position
- Company/Organization
- Street Address
- City
- State/Prov/Pref
- Zip/Postal Code
- Country
- Email-Address
- Telephone and Fax Number

Send it along with any other correspondence that cannot be sent via email to:

```
Gray Watson
330 Harmar St.
Pittsburgh, PA  15219-3748
USA
```

2 How to Debug with the Library

2.1 Allocation Basics: Terms and Functions

2.1.1 Basic Concept Definitions

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 run-time 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;
        ...
}
```

Heap data is also allocated at run-time 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 I'm 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 run-time 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 checkout 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.

2.1.2 Malloc Library Functions

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 in old_pnt as it can into the new_pnt space up to new_size bytes.

`void` **free** `( void *` *pnt* `)` Function

Usage: `free(pnt)`

The free routine releases an allocation 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.

NOTE: the returned address from the memory allocation/reallocation functions should *always* be cast to the appropriate pointer type for the variable being assigned.

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.

2.2 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 3.1.1 [Allocation Macros], page 13.

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 3.1.2 [Return Address], page 14.

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 3.2.3 [Fence-Post Overruns], page 22.

  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 3.2.2 [Memory Leaks], page 20.

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

examining unfreed 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.

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.

2.3 How to get the library.

Standard Repository:

The newest versions of the dmalloc library are available via anonymous ftp from '`ftp.letters.com`' in the '`/src/dmalloc`' directory.

To use anonymous ftp, you ftp to the site and when the system prompts you for a login-id or username you enter `anonymous`. When it prompts you for a password you enter your email address. I, for example, enter `gray@letters.com`. You then can change-directory (cd) into '`/src/dmalloc`' and get the '`README`' and '`dmalloc.tar.gz`' files.

The versions in this repository include such stuff as a postscript version of the manual and other large files which may not have been included in the distribution you received.

Other Repositories:

You can also get a recent version from anonymous ftp via '`gatekeeper.dec.com`' in the '`/pub/misc/dmalloc`' directory. This repository has been made available through the generosity of the Digital Equipment Corporation with special help from Dave Hill and the gatekeepers.

Thanks much to them all.

2.4 Installation of the Library

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

1. First, please examine the '`PERMISSIONS`' file. If you are using the library in a commercial setting, please consider paying for a licensed copy of the software.
2. You probably will want to edit the settings in '`settings.dist`' to tune specific features of the library. The below '`configure`' script will copy this file to '`settings.h`' which is where you should be adding per-architecture settings.
3. Type `sh ./configure` to configure the library. You may want to first examine the '`config.help`' file for some information about configure. Configure should generate the '`Makefile`' and some configuration files automatically.

   NOTE: It seems that some versions of tr (especially from HP-UX) don't understand `tr '[a-z]' '[A-Z]'`. Since configure uses tr often, you may need to either get GNU's tr (in their textutils package) or generate the '`Makefile`' and '`conf.h`' files by hand.
4. You may want to examine the '`Makefile`' and '`conf.h`' files created by configure to make sure it did its job correctly.
5. You might want to tune the settings in '`settings.h`' file to tune the library to the local architecture – for instance handling local thread library features. The '`configure`' script created this file from the '`settings.dist`' file.
6. 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.

7. Typing `make` should be enough to build '`libdmalloc.a`', '`libdmalloclp.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. In general 64 bit systems usually have lots of warnings about int to pointer promotion.

   NOTE: The code is pretty dependent on a good ANSI-C compiler like gcc. 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. Otherwise, you will have to try `make noansi`. This will run the '`Deansify.pl`' perl script on the code which:

   - *WARNING*: modifies the source code in place
   - changes '`stdarg.h`' and `...` to '`varargs.h`' and `va_alist`
   - changes all `void *` references to `char *`.
   - fixes all functions to remove `foo(char * var)` declarations.

   If it doesn't work you may have to do Deansify.pl's job by hand.

8. Typing `make tests` should build the '`dmalloc_t`' test program.
9. Typing `make light` should 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.
10. Typing `make install` should install the '`libdmalloc.a`' and '`libdmalloc_lp.a`' library files in '`/usr/local/lib`', the '`dmalloc`' utility in '`/usr/local/bin`', and the '`dmalloc.info`' documentation file in '`/usr/local/info`'.

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

See the Getting Started section to get up and running with the library. See Section 2.5 [Getting Started], page 12.

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

2.5 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. First, please examine the 'PERMISSIONS' file. If you are using the library in a commercial setting, please consider paying for a licensed copy of the software.
2. Follow the installation instructions on how to configure and make and install the library (i.e. type: `make install`). See Section 2.4 [Installation], page 10.
3. Add an alias for dmalloc to your shell's rc file. csh/tcsh users should add the following to their '.cshrc' file (notice the -C option for c-shell output):
   ```
   alias dmalloc 'eval '\dmalloc -C \!*''
   ```
   zsh users should add the following to their '.zshrc' file (notice the -b option for bourne shell output):
   ```
   dmalloc() { eval 'command dmalloc -b $*' }
   ```
4. 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 3.1.1 [Allocation Macros], page 13.
5. Link the dmalloc library into your program.
6. Enable the debugging features by typing `dmalloc -l logfile -i 100 low` (for example). This will:
   - set the malloc log path to 'logfile' (`-l logfile`)
   - have the library check itself every 100 iterations (`-i 100`)
   - enable a number of debug features (`low`). You can also try `runtime` for minimal checking or `medium` or `high` for more extensive heap verification. `all` is also provided but generates a multitude of log messages without many more tests.

   `dmalloc --usage` will provide verbose usage info for the dmalloc program. See Section 3.6 [Dmalloc Program], page 28.

   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 3.5 [RC File], page 27.
7. Run your program, examine the logfile, and voila!

3 Details about the Library's Operations

3.1 Programming with the Library

3.1.1 Allocation Macros

By including '`dmalloc.h`' in your C files, your calls to calloc, free, malloc, or realloc are replaced with calls to _calloc_leap, _free_leap, _malloc_leap, and _realloc_leap. Additionally the library replaces calls to xcalloc, xfree, xmalloc, xrealloc, and xstrdup with _leap calls.

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 malloc functions if they come after it.

These *leap 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 leap routines take this information and pass it on to the library making it able 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 3.2.2 [Memory Leaks], page 20.

Along with the above leap functionality, '`dmalloc.h`' also adds the following macros which take care of all the type-casting and make the code look much cleaner (IMHO).

`void * ALLOC(`*type*`, int `*count*`)` Function

Usage: `long_pnt = ALLOC(long, 30)`. This means allocate space for 30 longs.

`void * MALLOC ( int `*size*` )` Function

Usage: `char_pnt = MALLOC(1000)`. This is like ALLOC but for characters only. It means allocate space for 1000 characters.

`void * CALLOC ( type, int count )` Function

Usage: `info_pnt = CALLOC(struct info_st, 100)`. This means allocate space for 100 info_st structures and zero them all.

NOTE: the arguments for the CALLOC macro are sort of reversed from calloc(unsigned int count, unsigned int size).

`void * REALLOC ( void * pnt, type, int count )` Function

Usage: `long_pnt = REALLOC(old_pnt, long, 10)`. This takes old_pnt and and changes its size to accommodate 10 longs.

NOTE: the arguments for the REALLOC macro are different from the realloc function.

`void * REMALLOC ( void * pnt, int size )` Function

Usage: `char_pnt = REMALLOC(char_pnt, 100)`. This is like REALLOC but for characters only. It takes char_pnt and changes its size to 100 characters.

`void` **FREE** `( void * pnt )` Function

Usage: `FREE(pnt)`. This frees memory pointers.

3.1.2 Return 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/compilers. See Section 4.3 [Portability], page 34.

The library attempts to use some assembly hacks to get the the return-address or the address of the line that called the dmalloc function. If you have the '`log-unknown`' token enabled and you run your program, 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 the manual methods below for gdb.

```
(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.

3.1.3 Argument 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 3.4 [Debug Tokens], page 25.

```
cc -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 3.1.6 [Disabling the Library], page 19.

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 `strcasecmp`. 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.

3.1.4 Extension Routines

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

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

`int dmalloc_errno`
: 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.

`int dmalloc_address`
: This variable holds the address to be specifically looked for when allocating or freeing by the library.

`int dmalloc_address_count`
: This variable stores the argument to the address library setting. If it is set to a greater than 0 value then after the library has seen the '`addr`' address this many times, it will call `dmalloc_error()`.

: This works well in conjunction with the `STORE_SEEN_COUNT` option. See Section 3.2.2 [Memory Leaks], page 20.

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 `on_exit()` or `atexit()` calls (see '`conf.h`'). If you do not have these routines, then `dmalloc_shutdown` should be called right before `exit()` or as the last function in `main()`.

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

`void` **dmalloc_log_heap_map** (`void`) — Function

This routine logs to the logfile (if it is enabled) a graphical representation of the current heap space.

`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 dumps the unfreed-memory information to the log file. This is also useful to dump the currently allocated points to the log file to be diff'ed against another dump later on.

`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.

`void` **dmalloc_debug** (`long` *debug*) — Function

This routine overrides the debug setting from the environment variable and sets the library debugging features explicitly. 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.

One problem however is that some systems make calls to memory allocation functions *before* `main()` is reached therefore before `dmalloc_debug()` can be called meaning some debugging information may be generated regardless.

`long` **dmalloc_debug_current** (`void`) Function

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

`int` **dmalloc_examine** (`char *` *pnt*, `int *` *size*, char ** *file*, int * *line*, void ** *ret_address*) Function

This function returns the size of a pnt's allocation as well as the file and line or the return-address from where it was allocated. It will return NOERROR or ERROR depending on whether pnt is good or not.

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

`char * dmalloc_strerror` (`int` *errnum*) Function

This function returns the string representation of the error value in errnum (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 Section 3.6 [Dmalloc Program], page 28.

3.1.5 C++ and the Library

For those people using the C++ language, some special things need to be done to get the library to work. The problem exists with the fact that the dynamic memory routines in C++ are `new()` and `delete()` as opposed to `malloc()` and `free()`.

The file '`dmalloc.cc`' is provided in the distribution which effectively redirects `new` to the more familiar `malloc` and `delete` to the more familiar `free`. Compile and link this file in with the C++ program you want to debug.

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.

3.1.6 Disabling the Library

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 your 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 leap macros to not be applied. See Section 3.1.1 [Allocation Macros], page 13.

  ```
  cc -g -DMALLOC_DISABLE 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 not compiled all your source with `DMALLOC_DISABLED` defined then you need to link your program with the '`libdmalloclp.a`' library.

  ```
  cc main.o -L/usr/local/lib -ldmalloclp -lgmalloc
  ```

  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 the '`libdmalloclp.a`' library.

  ```
  cc -g main.o -L/usr/local/lib -lgmalloc
  ```

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

3.2 Using a Debugger with the Library

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 (I use '`dmalloc`') and turn on a set of debug features. You can use `dmalloc -l dmalloc low` to accomplish this. If you are interested in having the error messages printed to your terminal as well, enable the '`print-error`' token by typing `dmalloc -p print-error` afterwards. See Section 3.6 [Dmalloc Program], page 28.

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.

3.2.1 Diagnosing General Errors 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 `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 3.4 [Debug Tokens], page 25 and See Section 3.6 [Dmalloc Program], page 28.

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 3.1.4 [Extensions], page 16.

3.2.2 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 3.1.2 [Return Address], page 14.

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 3.1.1 [Allocation Macros], page 13.

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 break point 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.

3.2.3 Diagnosing Fence-Post Overwritten Memory

For a definition of fence-posts please see the Features section. See Section 2.2 [Features], page 8.

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 3.4 [Debug Tokens], page 25 and See Section 3.6 [Dmalloc Program], page 28.

3.3 Environment Variable 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. It can be set either by hand or with the help of the dmalloc program. See Section 3.6 [Dmalloc Program], page 28.

To set it by hand, C shell (csh or tcsh) users need to invoke:

```
setenv DMALLOC_OPTIONS value
```

Bourne shell (sh, bash, ksh, or zsh) users should use:

```
DMALLOC_OPTIONS=value
export DMALLOC_OPTIONS
```

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 3.4 [Debug Tokens], page 25. 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 3.4 [Debug Tokens], page 25. If 'debug' and the tokens are both used, the token values will be added to the debug value.

`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 3.2.2 [Memory Leaks], page 20.

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 line number is 0 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 'malloc' as the log-file
setenv DMALLOC_OPTIONS log-trans,log-stats,log=malloc

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

3.4 Debugging Tokens

The below tokens and their corresponding descriptions are for the setting of the debug library setting in the environment variable. See Section 3.3 [Environment Variable], page 23. They should be specified in the user's '`.dmallocrc`' file. See Section 3.5 [RC File], page 27.

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 Section 3.6 [Dmalloc Program], page 28.

`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-thread-id`
: for systems that have multi-threaded programs (don't worry if this does not make sense to you), log thread-id for allocated pointer (see '`conf.h`')

`log-trans`
: log general memory transactions (quite verbose)

`log-stamp`
: log a time stamp for all messages

`log-admin`
: log administrative information (quite verbose)

`log-blocks`
: log detailed block information when dmalloc_log_heap_map is called

`log-unknown`
: like log-non-free but logs non-freed memory pointers that did not have file/line information associated with them

`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-lists`
: examine internal heap linked-lists

`check-blank`
: check to see if space that was blanked by free-blank or alloc-blank has been overwritten

`check-funcs`
: check the arguments of some functions (mostly string operations) looking for bad pointers

`realloc-copy`
: always copy data to a new pointer when realloc

`free-blank`
: write special bytes (decimal 197, octal 0305, hex 0xc5) into space when it is freed

`error-abort`
: abort the program (and dump core) on errors. See `error-dump` below.

`alloc-blank`
: write special bytes (decimal 197, octal 0305, hex 0xc5) into space when it is alloced

`heap-check-map`
: log a heap-map to the logfile every time the heap is checked

`print-error`
: 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 sbrk

`never-reuse`
: have the heap never use space that has been used before and freed. See Section 3.2.2 [Memory Leaks], page 20. *WARNING*: This should be used with caution since you may run out of heap space.

`allow-nonlinear`
: have the heap not complain when additional program functionality seems to have made use of the system's heap-allocation routine `sbrk` directly. This is now enabled by default

since an increasing number of operating system functions seem to be doing this – one example is the pthreads package.

WARNING: This should be used with caution since it may hide certain heap problems.

`allow-zero`
: the library will not generate errors when a program asks for a 0 byte allocation or when someone tries to free a NULL pointer.

  NOTE: This does not impact the ALLOW_REALLOC_NULL compilation options which can be adjusted in '`conf.h`'.

`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.

  NOTE: This will only work if your system supports the `fork` system call.

3.5 Run-Time Configuration File

By using a *RC File* (or run-time configuration file) you can alias tags to combinations of debug tokens. See Section 3.4 [Debug Tokens], page 25.

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:

`runtime` enables basic run-time tests

`low` turns on minimal checking of heap structures

`medium` significant checking of heap areas

`high` extensive checking of heap areas

`all` turns on all the checking possible. This generates a multitude of log messages without many more tests than high.

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 Section 3.6 [Dmalloc Program], page 28 and Section 3.4 [Debug Tokens], page 25.

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 run-time 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, check-lists, error-abort

# good utilities
debug3 log-stats, log-non-free, log-trans, \
        log-admin, check-fence, check-heap, check-lists, 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 Section 3.6 [Dmalloc Program], page 28. 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.

3.6 Dmalloc Utility Program

The dmalloc program is designed to assist in the setting of the environment variable 'DMALLOC_OPTIONS'. See Section 3.3 [Environment Variable], page 23. 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. Csh/tcsh users (for example) should put the following in their '`.cshrc`' file:

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

Zsh users on the other hand should put the following in their '`.zshrc`' file:

```
dmalloc() { eval 'command dmalloc -b $*' }
```

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

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 run-time configuration file or should be one of the built-in tags. See Section 3.5 [RC File], page 27. 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 via ftp from '`ftp.letters.com`' in the '`/src/argv`' directory. See '`argv.info`' there for more information.

Here is a detailed list of the flags that can 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.

`-C`
: Output C shell type commands.

`-c`
: Clear/unset all of the settings not specified with other arguments.

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/.dmallocrc`'.

`-i number`
: Set the checking interval to number.

`-k`
: Keep the settings when using a tag. This overrides -r.

`-l filename`
: Set the log-file to filename.

`-L`
: Output the debug-value 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's can be specified.

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

`-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's 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.

`-s number`
: Set the '`start`' part of the '`DMALLOC_OPTIONS`' env variable to number (alternatively '`file:line`').

`-S`
: Output the debug-value not in hex but by individual debug-tokens in short form.

`-t`
: List all of the tags in the rc-file. Use with -v or -V verbose options.

`-v`
: Give verbose output. Especially useful when dumping current settings or listing all of the tags.

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-blocks, log-unknown,
   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 'dmalloc'
dmalloc high -i 100 -l dmalloc

# 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

# 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 'dmalloc'
dmalloc low -p log-trans -l dmalloc

# 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 Source Code Information

4.1 Definitions of Terms

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 ^ 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`'.

4.2 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 a manual compilation option in the '`conf.h`' file.
- The library does *not* provide memalign() nor valloc() support as of yet, but may in future releases. The author is interested to know who is using these functions and for what reason.
- 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 '`conf.h`' for some manual compilation options to handle this.

- 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.

4.3 Portability Issues

General portability issues center around:

- sbrk or compatible function usages
- Whether the systems' heap grows towards high or low memory. The chunk.c code is designed (loosely) around the fact that consecutive calls to sbrk should give higher memory addresses.

  The library has not been tested on a system whose heap grows towards low memory. If you are trying to run the library on such a system please contact the author.
- 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 (GNUs 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();
  }
  ```

5 Plugs and Soapbox Comments

The author would like to bring the following organizations to your attention. If you would like any more information about the below, please mail to the supplied addresses or drop the author a line with any questions.

The Electronic Frontier Foundation (EFF)
: The EFF is a organization committed to ensuring that the rules, regulations, and laws being applied to emerging communications technologies are in keeping with our society's highest traditions of the free and open flow of ideas and information while protecting personal privacy. <eff@eff.org>

Computer Professionals for Social Responsibility (CPSR)
: CPSR is a public-interest alliance of computer scientists and others interested in the impact of computer technology on society. We work to influence decisions regarding the development and use of computers because those decisions have far-reaching consequences and reflect basic values and priorities. <cpsr@csli.stanford.edu>

Berkeley Software Design, Inc. (BSDI)
: The author has been a proud and enthusiastic owner and user of the BSD/386 operating system for some time now. For around $1k you get a *complete* BSD-flavor operating system with *full source* for 386 and 486 systems (binary licenses are available). Along with the obvious benefits of full source code come excellent customer support/service and system features such as a MS-DOG runtime environment, complete tcp/ip networking facilities including nfs, full software development utilities, X, etc. <bsdi-info@bsdi.com>

Index of Concepts

Table of Contents