# Debugging Malloc Lab

## Introduction

The usual implementation of malloc and free are unforgiving to errors in their callers' code, including cases where the programmer overflows an array, forgets to free memory, or frees a memory block twice. This often does not affect the program immediately, waiting until the corrupted memory is used later (in the case of overwrites) or gradually accumulating allocated but unused blocks. Thus, debugging can be extremely difficult.

In this assignment, you will write a wrapper for the malloc package that will catch errors in the code that calls malloc and free. The skills you will have learned upon the completion of this exercise are pointer arithmetic and a greater understanding of the consequences of subtle memory mistakes.

## Logistics

Unzip **debugging_malloc.zip** into an empty directory. The files contained are as follows:

| **File(s):** | **Function:** |
|---|---|
| debugmalloc.c | Contains the implementation of the three functions you will be writing. **This is the one file you will be editing and handing in.** |
| debugmalloc.h | Contains the declaration of the functions, as well as the macros that will call them. |
| driver.c | Contains main procedure and the code that will be calling the functions in the malloc package |
| dmhelper.c, dmhelper.h | Contain the helper functions and macros that you will be calling from your code |
| grader.pl | Perl script that runs your code for the various test cases and gives you feedback based on your current code |
| debugmalloc.dsp | Exercise 3 project file |
| debugmalloc.dsw | Exercise 3 workspace file |
| tailor.h, getopt.c,getopt.h | Tools that are used only by the driver program for I/O purposes. You will not need to know what the code in these files do. |
| Others | Required by Visual C++. You do not need to understand their purpose |

## Specification

Programs that use this package will call the macro MALLOC and FREE. MALLOC and FREE are used exactly the same way as the malloc() and free() functions in the standard C malloc package. That is, the line

```
void *ptr = MALLOC ( n ) ;
```

will allocate a ***payload*** of at least n bytes, and ptr will point to the front of this block. The line

```
FREE(ptr);
```

will cause the payload pointed to by ptr to be deallocated and become available for later use. The macros are defined as follows:

```
#define MALLOC(s) MyMalloc(s, __FILE__, __LINE__)
#define FREE(p) MyFree(p, __FILE__, __LINE__)
```

The __FILE__ macro resolves to the filename and __LINE__ resolves to the current line number.

The debugmalloc.c file contains three functions that you are required to implement, as shown:

```
void *MyMalloc(size_t size, char *filename, int linenumber);
void MyFree(void *ptr, char *filename, int linenumber);
  int AllocatedSize();
```

Using the macros above allow MyMalloc and MyFree to be called with the filename and line number of the actual MALLOC and FREE calls, while retaining the same form as the usual malloc package. By default, MyMalloc() and MyFree() simply call malloc() and free(), respectively, and return immediately.  AllocatedSize() should return the number of bytes currently allocated by the user: the sum of the requested bytes through MALLOC minus the bytes freed using FREE. By default, it simply returns 0 and thus is unimplemented. The definitions are shown below:

```
void *MyMalloc(size_t size, char *filename, int linenumber) {


     return (malloc(size));
}
```

```
     void MyFree(void *ptr, char *filename, int linenumber) {


          free(ptr);
     }

     int AllocatedSize() {


          return 0;
     }
```

Your job is to modify these functions so that they will catch a number of errors that will be described in the next section.

There are also two optional functions in the debugmalloc.c file that you can implement:

```
     void PrintAllocatedBlocks(); int HeapCheck();
```

PrintAllocatedBlocks should print out information about all currently allocated blocks.
HeapCheck should check all the blocks for possible memory overwrites.

## Implementation Details

To catch the errors, you will allocate a slightly larger amount of space and insert a header and a footer around the "requested payload". MyMalloc() will insert information into this area, and MyFree() will check to see if the information has not changed. The organization of the complete memory block is as shown below:

| Header | | | ***Payload*** | Footer |
|---|---|---|---|---|
| Checksum | ... | Fence | | Fence |

**Note:** MyMalloc() returns a pointer to the payload, *not* the beginning of the whole block. Also, the ptr parameter passed into MyFree(void *ptr) will point to the payload, *not* the beginning of the block.

Information that you might want to store in this extra (header, footer) area include:

- a "fence" immediately around the requested payload with a known value like 0xCCDEADCC, so that you can check if it has been changed when the block is freed.
- the size of the block
- a checksum for the header to ensure that it has not been corrupted (A checksum of a sequence of bits is calculated by counting the number of "1" bits in the stream. For example, the checksum for "1000100010001000" is 4. It is a simple error detection mechanism.)
- the filename and line number of the MALLOC() call

The errors that can occur are:

- Error #1: Writing past the beginning of the user's block (through the fence)
- Error #2: Writing past the end of the user's block (through the fence)
- Error #3: Corrupting the header information
- Error #4: Attempting to free an unallocated or already-freed block
- Error #5: Memory leak detection (user can use ALLOCATEDSIZE to check for leaks at the end of the program)

To report the first four errors, call one of these two functions:

```
void error(int errorcode, char *filename, int linenumber);
        errorcode is the number assigned to the error as
        stated above. filename and linenumber contain the
        filename and line number of the line (thefree call) in
        which the error is invoked. For example, call error(2,
        filename, linenumber) if you come across a
        situation where the footer fence has been changed.
void errorfl(int errorcode, char *filename_malloc, int
linenumber_malloc, char *filename_free, int linenumber_free);
        This is the same as the error(), except there are
        two sets of filenames and line numbers, one for the
        statement in which the block was malloc'd, and the
        other for the statement in which the block was
        free'd (and the error was invoked).
```

The fact that MyMalloc() and MyFree() are given the filename and line number of the MALLOC() and FREE() call can prove to be very useful when you are reporting errors. The more information you print out, the easier it will be for the programmer to locate the error. Use errorfl() instead of error() whenever possible.errorfl() obviously cannot be used on situations where FREE() is called on an unallocated block, since it was not ever MALLOC'd.

**Note:** You will only be reporting errors from MyFree(). None of the errors can be caught in MyMalloc()

In the case of memory leaks, the driver program will call AllocatedSize(), and the grader will look at its return value and possible output. AllocatedSize()should return the number of bytes currently allocated from MALLOC and FREE calls. For example, the code segment:

```
void *ptr1 = MALLOC(10), *ptr2 = MALLOC(8); FREE(ptr2);
printf("%d\n", AllocatedSize());
```

should print out "10".

Once you have gotten to the point where you can catch all of the errors, you can go an optional step further and create a global list of allocated blocks. This will allow you to perform analysis of memory leaks and currently allocated memory. You can implement the void PrintAllocatedBlocks() function, which prints out the filename and line number where all currently allocated blocks were MALLOC()'d. A macro is provided for you to use to print out information about a single block in a readable and gradeable format:

```
PRINTBLOCK(int size, char *filename, int linenumber)
```

Also, you can implement the int HeapCheck() function. This should check all of the currently allocated blocks and return -1 if there is an error and 0 if all blocks are valid. In addition, it should print out the information about all of the corrupted blocks, using the macro #define PRINTERROR(int errorcode, char *filename, int linenumber), with errorcode equal to the error number (according to the list described earlier) the block has gone through.

You may find that this global list can also allow you to be more specific in your error messages, as it is otherwise difficult to determine the difference between an overwrite of a non-payload area and an attempted FREE() of an unallocated block.

## Evaluation

You are given 7 test cases to work with, plus 1 extra for testing a global list. You can type "debugmalloc -t n" to run the n-th test. You can see the code that is being run in driver.c. There are a total of 100 possible points.

Here is a rundown of the test cases and desired output (do not worry about the path of the filename):

| **Test case #1** | |
|---|---|
| **Code** | char *str = (char *) MALLOC(12); strcpy(str, "123456789"); |

| | |
|---|---|
| | FREE(str); printf("Size: %d\n", AllocatedSize()); PrintAllocatedBlocks(); |
| **Error #** | None |
| **Correct Output** | Size: 0 |
| **Points worth** | 10 |
| **Details** | 10 points for not reporting an error and returning 0 in AllocatedSize() |

| **Test case #2** | |
|---|---|
| **Code** | char *str = (char *) MALLOC(8); strcpy(str, "12345678"); FREE(str); |
| **Error #** | 2 |
| **Correct Output** | Error: Ending edge of the payload has been overwritten. in block allocated at driver.c, line 21 and freed at driver.c, line 23 |
| **Points worth** | 15 |
| **Details** | 6 pts for catching error<br>3 pts for printing the filename/line numbers<br>6 pts for correct error message |

| **Test case #3** | |
|---|---|
| **Code** | char *str = (char *) MALLOC(2); strcpy(str, "12"); FREE(str); |
| **Error #** | 2 |
| **Correct Output** | Error: Ending edge of the payload has been overwritten. in block allocated at driver.c, line 28 and freed at driver.c, line 30 |
| **Points worth** | 15 |
| **Details** | 6 pts for catching error<br>3 pts for printing the filename/line numbers<br>6 pts for correct error message |

| **Test case #4** |
|---|

| | |
|---|---|
| **Code** | void *ptr = MALLOC(4); *ptr2 = MALLOC(6); FREE(ptr); printf("Size: %d\n", AllocatedSize()); PrintAllocatedBlocks(); |
| **Error #** | None |
| **Correct Output** | Size: 6 Currently allocated blocks: 6 bytes, created at driver.c, line 34 |
| **Points worth** | 15 |
| **Details** | 15 pts for not reporting an error and returning 6 from AllocatedSize<br>Extra for printing out the extra block |

| **Test case #5** | |
|---|---|
| **Code** | void *ptr = MALLOC(4); FREE(ptr); FREE(ptr); |
| **Error #** | 4 |
| **Correct Output** | Error: Attempting to free an unallocated block. in block freed at driver.c, line 43 |
| **Points worth** | 15 |
| **Details** | 15 pts for catching error<br>Extra for correct error message |

| **Test case #6** | |
|---|---|
| **Code** | char *ptr = (char *) MALLOC(4); *((int *) (ptr - 8)) = 8 + (1 << 31); FREE(ptr);（见代码，修改了header中的size） |
| **Error #** | 1 or 3 |
| **Correct Output** | Error: Header has been corrupted.<br>or<br>Error: Starting edge of the payload has been overwritten. in block allocated at driver.c, line 47 and freed at driver.c, line 49 |
| **Points worth** | 15 |
| **Details** | 9 pts for catching error<br>6 pts for a correct error message |

| **Test case #7** | |
|---|---|
| **Code** | char ptr[5]; FREE(ptr); |

| | |
|---|---|
| **Error #** | 4 |
| **Correct Output** | Error: Attempting to free an unallocated block. in block freed at driver.c, line 54 |
| **Points worth** | 15 |
| **Details** | 15 pts for recognizing error<br>Extra for printing correct error message |

| **Test case #8 (Optional)** | |
|---|---|
| **Code** | int i; int *intptr = (int *) MALLOC(6); char *str = (char *) MALLOC(12); for(i = 0; i < 6; i++) { intptr[i] = i; } if (HeapCheck() == -1) { printf("\nCaught Errors\n"); } |
| **Error #** | None |
| **Correct Output** | Error: Ending edge of the payload has been overwritten. Invalid block created at driver.c, line 59 Caught Errors |
| **Points worth** | Extra |
| **Details** | "Caught Errors" indicates that the HeapCheck() function worked correctly. Extra points possible. |

Your instructor may give you extra credit for implementing a global list and the PrintAllocatedBlocks() and HeapCheck() functions.