CSE 303, Spring 2009
Homework 6: Mallocater, 60 points

Milestone 1 (SVN repository) due Monday, May 25, 2009, 9:00 AM

Milestone 2 (Complete Code) due Monday, June 1, 2009, 9:00 AM

This group assignment focuses on manual memory management, bit-packing, and version control. It is inspired by a similar assignment given by lecturer Hal Perkins of UW CSE. You will write the following files:

- `memalloc.c`, `memalloc.h` - code/header for core memory allocation functions
- `memops.c`, `memops.h` - code/header for bulk memory operations
- `memdebug.c`, `memdebug.h` - code/header for debugging utility functions
- `memimpl.h` - header for implementation details/structures not intended to be used publicly by clients
- `test.c` - testing and benchmarking program for your memory allocater
- `Makefile` - basic compilation manager script to build your program
- `README` - brief text description of important details about your project
- `extracredit.c`, `extracredit.h` - code and header for extra credit features, if you choose to implement any *(optional)*

Since there are so many files, you will submit all files from this assignment as an archive named `cse303hw6.tar.gz` from the Homework section of the course web site.

Assignment Description:

The purpose of this assignment is to implement your own library for dynamic memory allocation, similar to what is provided by the standard library functions `malloc` and `free`. You will begin with a large block of memory from which you will allocate smaller chunks to clients as they request it. You will maintain a list of allocated and free blocks that is manipulated as each client asks to allocate or free a block. You will also write a testing program that verifies that your code properly allocates and deallocates memory.

Since you are implementing a memory allocater, **you are not allowed to call the built-in standard library functions for heap memory allocation, such as `malloc`, `calloc`, `realloc`, or `free`, at any place in your code for any reason.**

Your code will simulate a heap of memory to be dynamically allocated to client programs upon request. An important concept to understand about this assignment is that most of your code is not in itself a complete C program. Instead, it is a library of functions that others could build upon to write their own C programs. Your library has functions for allocating memory. Basically, you are re-implementing crappier versions of `malloc` and `free`. Various client programs could include your library in with their code and could call your functions to perform dynamic memory allocation. Therefore when working on this assignment, you must think about the client's view versus your own internal view of your code.

The C standard library functions `malloc` and `free` allocate memory dynamically from the system heap. In our allocation library, you will allocate memory dynamically from a large single block of bytes. The client program will begin interacting with your library by calling an initialization function `init303` and passing you a pointer to the start of this large block of bytes, along with its size. It's your job to store that information for later, so that when the client later asks to allocate memory, you will pull chunks from this large overall block and give them back to the client.

Memory Block List Implementation:

You will implement four core functions for allocating and deallocating memory, named `malloc303`, `calloc303`, `realloc303`, and `free303`. Internally these functions share a single data structure called the *block list*, which is a linked-list of memory blocks that are available to satisfy memory allocation requests, or that have been allocated to satisfy prior memory allocation requests. Think of it as a linked list where each node stores a block of memory, knows the size of that block of memory and whether it is free or in use, and has a pointer to the next node in the block list.

The client initializes your system by calling a function `init303` and passing you a pointer to the start of a large block of memory, along with an integer representing the size of that block. This block is the overall "heap" of memory that you will be managing when the client later calls your malloc/free functions. At first, all of the memory is stored in a single large free block. It may seem odd that the client is passing you the memory and then expecting your code to manage that memory; why doesn't the client just manage the memory itself? This is more of a practical issue, so that you can write client programs that initialize your memory manager with different amounts of bytes to work with. In a more real implementation your library would probably construct its overall heap block of memory itself.

Your block list must be implemented in a very particular way to get full credit on this assignment. Your block list must be implemented *in place*, inside the overall pool of heap memory that you're managing. Each "linked list node" in the system must be a single 4-byte structure that is packed into part of the overall heap block. So, in other words, if the client calls `init303` and gives you a 2048-byte block to play with, your initial memory layout will be the following. Notice that the 4-byte list node headers slightly take away from the actual usable amount of memory that you can allocate.

bytes	**0-3**	**4-2047**
contents	list node *[size=2044, free]*	2044-byte block of free memory

Figure 1 Initial heap memory

The exact contents of each list node are up to you, but generally you need to keep track of things like the size of the upcoming block of memory, whether it is free or in use, possibly where to find the next node in the block list, etc. You may wonder how all of this information can be packed into just 4 bytes (32 bits). Keep in mind that boolean flags can be stored using a single bit, and that you probably don't need all 32 bits to store a pointer or offset if you store it compactly. You may not need to store the actual address of the next free list node, since you can figure this out using the size. You may also want to have a flag to tell which node is the last node in the block list.

When a client program requests to **allocate** a block of memory by calling one of your memory allocation functions, you should scan the block list looking for the first free block of storage that is at least as large as the amount requested, break off a chunk from that block of appropriate size, and return a pointer to that chunk. This is called a **"first-fit"** memory allocation strategy, because we use the first available chunk of free memory that we find. There are other memory allocation strategies such as "best-fit", "worst-fit", and so on, each having its own merits and drawbacks.

For example, suppose we start with 2048 bytes and the client requests to allocate 100 bytes by calling `malloc303(100)`. These 100 should be broken off from the overall master heap chunk of 2048, yielding the following block list state:

bytes	**0-3**	**4-103**	**104-107**	**108-2047**
contents	list node *[size=100, in use]*	100-byte block of **used** memory	list node *[size=1940, free]*	**1940**-byte block of free memory

Figure 2 Heap memory after `malloc303(100)` call

`malloc303(100)` returns this address

The pointer that should be returned by the `malloc303` call should point to byte #4 from the start of the overall heap, as indicated by the arrow above. Technically just before this pointer (and just after the end of the memory returned) would be the contents of the list node structures, but the client doesn't know that.

Suppose another call is made of `malloc(1024)`. The result would be the following:

bytes	**0-3**	**4-103**	**104-107**	**108-1131**	**1132-1135**	**1136-2047**
contents	list node *[size=100, in use]*	100-byte block of used memory	list node *[size=1024, in use]*	1024-byte block of **used** memory	list node *[size=912, free]*	**912**-byte block of free memory

Figure 3 Heap memory after second `malloc303(1024)` call

`malloc303(1024)` returns this address

The main idea is that whenever your library is asked to allocate memory, you must loop through the block list looking for a sufficiently large free block and allocate from that free block to the caller. Most often the free block is larger than the amount of memory requested, so you'll have to break it into two pieces as in our diagram.

If the free block of memory is only a little bit larger than the size requested by the client, then it may not make sense to split it and leave a tiny chunk on the free list that is unlikely to be useful in satisfying future requests. You can experiment with this threshold and decide what number should be the smallest allowed free chunk in the block list. The actual number should be declared as a constant in your code. **You should at least allocate memory in chunks that are multiples of 4 bytes**, so that blocks line up evenly in the heap.

If many blocks of memory have been allocated, your code will have to potentially loop through many of them to find a suitable free block to allocate from. Your code runs in O(N) time where N is the number of allocated blocks. This is not ideal if many small blocks are allocated by the client program, but we will ignore the issue.

When a client requests to **free** memory by calling your `free303` function, you should set the corresponding block of memory to be free in your block list. For example, suppose we have the block list from the preceding diagram and the client calls `free303` and passes the memory address of the start of the 100-byte block (byte 4 of the overall heap). Your code can shift this pointer backward by 4 bytes and examine the list node stored there to manipulate your linked list. The block list contents would be the following after the call.

bytes	**0-3**	**4-103**	**104-107**	**108-1131**	**1132-1135**	**1136-2047**
contents	list node *[size=100, **free**]*	100-byte block of **free** memory	list node *[size=1024, in use]*	1024-byte block of used memory	list node *[size=912, free]*	912-byte block of free memory

Figure 4 Heap memory after `free303` call

`free303` call passes this address

The idea here is that the client is passing you a pointer to the start of the block of memory you previously gave them. In the 4 bytes immediately before this address should be your linked list node structure that stores information about the block of memory. So in your `free303` function, you can subtract 4 bytes from the pointer the client gives you to examine the block's list node.

Sometimes the client frees a block of memory that neighbors other free blocks of memory. In such cases, you don't want to have lots of small neighboring free blocks, because this would lead to a problem called *memory fragmentation* where we have lots of free memory but can't give it out in large chunks at a time. To prevent this problem, **your library must merge neighboring free blocks into a single larger free block**. Notice that you also reclaim the 4 bytes used by the former list node header. For example, if the client had called `free303` with the 1024-byte block's address instead of the 100-byte block's, the block list would look like the following:

bytes	**0-3**	**4-103**	**104-107**	**108-2047**
contents	list node *[size=100, in use]*	100-byte block of **used** memory	list node *[size=1940, free]*	**1940**-byte block of free memory

Figure 5 Heap memory after alternate `free303` call

`free303` call passes this address

Note that a freed block might have a free neighbor either before it, after it, or both. So it is possible that up to 3 blocks of memory must be merged into a single free block. For example, if the client freed first the 100-byte block as in Figure 4, then freed the 1024-byte block, at that point the 1024-byte block would have both a 100-byte free block before it and a 912-byte free block after it. All three of those blocks would need to be merged into a single 2044-byte free block.

The act of combining adjacent free blocks is not so difficult if you have pointers to both list nodes. Just increase the size field of the earlier block by the size of the second block plus the extra 4 bytes for the list node being removed.

If the client calls one of your allocation functions but there is not any single block of free memory large enough to satisfy the request, you will not allocate any memory and will instead return `NULL`. Notice that it is possible for an allocation request to fail even if there is enough memory available, if that memory is not contiguous. For example, if the block list has the state from Figure 4, with two non-neighboring free blocks of size 100 and 912, and the client requests to allocate 1000 bytes, the request will fail, even though more than 1000 bytes of free memory are available in the system.

`memalloc` Functions:

Your `memalloc` code should implement the following functions:

`void init303(void* block, int size)`

This function must be called by the client program to initialize your memory allocation system. The client will pass you a pointer to the start of a large block of memory, as well as an integer representing the size of that block of memory. You must store this information so that chunks of this large block can be allocated later to client code. Subsequent calls to `malloc303`, `calloc303`, and `realloc303` will return blocks of memory from within this large overall block.

You may assume that `block` points to a valid location in memory and that `size` is a fairly large positive integer. All subsequent `memalloc` methods assume as a precondition that `init303` has been called prior to calling them, and their behavior is unspecified if `init303` has not been previously called.

You may assume that this function will be called only once for a given run of a program. You may also assume that the block is a region of memory of no more than 268,435,456 total bytes (2^{28} bytes, or 256 megabytes).

`void* malloc303(int size)`

In this function you should allocate a block of memory of at least the given size and returns a pointer to the start of the block. Blocks should be allocated in multiples of 4 bytes, so for example `malloc303(13)` would actually return a block of at least 16 bytes of memory. The memory's contents are not initialized. If no sufficiently large block of free memory exists, or if *size* is ≤ 0, you should return `NULL`. (Note that you are supposed to return a pointer to the start of the usable memory that the client is supposed to use, not a pointer to your linked list node structure in front of that usable memory.)

`void* calloc303(int num, int size)`

In this function you should allocate a block of memory large enough to hold at least *num* values, where each value occupies *size* bytes, and returns a pointer to the start of the block. (In other words, allocate at least `num * size` bytes.) The bytes of the block should be initialized so that they contain all 0s. Blocks should be allocated in multiples of 4 bytes, so for example the call of `calloc303(3, 3)` should actually return a block of 12 bytes of memory. If there is not enough free memory to allocate a block of the given size, or if *size* is less than or equal to 0, you should return `NULL`.

`void free303(void* p)`

In this function you should reclaim the bytes of memory referred to by the given block `p`, so that they can be potentially allocated again later You may assume as a precondition that `p` points to a valid block of memory that was previously allocated by a call to `malloc303`, `calloc303`, or `realloc303`, unless it is `NULL`, in which case you should do nothing. Presumably the 4 bytes just before `p` store the block list node structure of information about `p`'s memory, so you should examine this node and update its state to indicate that the block is free. If the value passed for `p` represents an invalid pointer (one that was not returned by your allocation functions), or if `p` refers to a block of memory that was already previously freed, the behavior of this function is undefined.

Your function should not only reclaim `p`'s memory for future allocations, but you should also combine any neighboring free blocks of memory as described previously to prevent memory fragmentation. That is, if you are freeing a block that is immediately preceded or followed by another free block, they should be merged into a single larger block.

`int sizeof303(void* p)`

In this function you should return the size of the memory block referred to by `p`; that is, the number of bytes that were allocated for this block when it was originally allocated. You may assume as a precondition that `p` points to a valid block of memory that was previously allocated by a call to `malloc303`, `calloc303`, or `realloc303`, unless it is `NULL`, in

which case this function should return 0. If the value passed for `p` represents an invalid pointer (one that was not returned by your functions), or if `p` refers to a block of memory that was already freed, the behavior of this function is undefined.

`memops` Functions:

Your `memops` code should implement the following functions. They are related to memory but do not explicitly rely on the `memalloc` functions you'll write; they could be passed any pointer, regardless of where it came from.

`void memcpy303(void* to, void* from, int size)`

In this function you should copy *size* bytes from `from` to `to`. You may assume that the regions of memory do not overlap.

If *size* is less than or equal to 0, or if `to` or `from` is `NULL`, nothing is copied. So long as they are not `NULL`, you should assume that they are valid memory locations that are read/writeable as needed.

Note that if you perform pointer arithmetic on a `void*`, it will shift by multiples of 4 bytes, not by single bytes. You may need to cast pointers to `char*` to be able to shift them by individual byte increments.

`void* memset303(void* p, int ch, int size)`

In this function you should set the next *size* bytes starting from `p` to store the integer value `ch`. `ch` is assumed to be a single-byte character value between 0 and 255 inclusive. Returns `p`.

If *size* is less than or equal to 0, or if `p` is `NULL`, nothing is written. So long as `p` is not `NULL`, you should assume that it is a valid memory location that is writeable.

`void* memzero303(void* p, int size)`

In this function you should set the next *size* bytes starting from `p` to store zeros. Returns `p`.

If *size* is less than or equal to 0, or if `p` is `NULL`, nothing is written. Note that `p` does not necessarily need to be blocks of memory that were returned by functions from your `memalloc` library; you should not check for this. So long as it is not `NULL`, you should assume that it is a valid memory location that is writeable.

`memdebug` Functions:

Your `memdebug` code should implement the following functions:

`void print_block(void* block)`

In this function you should print debugging information about the given block of memory, including the address of the block in hexadecimal, whether the block is in use or free, and its size in bytes. For example, a 512-byte used block at address 0x22cd80 would be printed as:

```
[0x22c8d0, used, 512B]
```

`void print_blocks(void)`

In this function you should print debugging information about all of the used and free blocks being managed by your `memalloc` system. The blocks should be numbered starting at 1, with 2-digit right-aligned number fields. For example:

```
 1: [0x22c8c4, used, 8B]
 2: [0x22c8d0, used, 512B]
 3: [0x22cad4, free, 148B]
 4: [0x22cb6c, used, 64B]
 5: [0x22cbb0, used, 16B]
 6: [0x22cbc4, free, 252B]
```

`memimpl.h`:

The exact contents of your `memimpl.h` file are not specified, but it is intended to be a place for you to declare any other globally important types, macros, and values that will be used in several of your `.c` files, but which are not generally

intended to be accessed by client programs. Specifically, any data structures and types related to your memory/free list should be declared here, so that this header can be included by your other files.

`README`:

You should submit a text file named `README` that briefly contains the following information. The document is not intended to be long; describe the information below as briefly as possible so long as it contains the information.

- Who was responsible for which part(s) of the work for your project.
- A brief description of how your allocation system works internally and any relevant algorithms not in this spec.
- Any additional or extra features you may have chosen to implement, or details you want the grader to notice.
- A brief description of the testing you have performed on your code, the behavior of your `test` program, etc.
- A brief comment about what resources you used when working on the project.
- A very brief reflection on your project experience: What parts were hardest? What parts were fun or interesting?

Testing Program:

Unlike in past assignments, we will not be providing you with any test cases or testing code to help you verify that your program works. So as part of this assignment you will also write and turn in a testing program that uses all of your various functions to verify that they work properly. You do not need to exhaustively test every function or every usage, but your testing program should be non-trivial and should test some function(s) in some complex way. In particular you should allocate and free several blocks of memory in different orders to make sure that your allocation algorithms work.

Though in general your group should not share code with other groups, you *will* be allowed to share your testing program with other groups. The goal is to help each other to ensure that everyone produces better and more robust code. We will post information on the course web site about how to find and post your testing program for others to use. You may not copy code directly from another group's testing program; you must still submit your own test that is your own work.

It can also be fun to convert a past C program (such as one of your previous CSE 303 assignments) to use your allocater library, to see if it will still work. Feel free to do this on your own if you like. But please do not submit such a program as your test program nor share it with others, since we don't want solutions to past assignments to be distributed publicly.

Some optional things you could do in your test program include: accepting command-line arguments, using randomness (the `rand` function), measuring time elapsed (the `clock` function), and displaying total memory or blocks allocated.

Makefile and Compilation:

You should submit a basic make file named `Makefile` to build your application. Your `Makefile` must provide the following functionality (it can optionally have more if you like):

- The command `make` or `make all` must compile all `.c` files into `.o` and also build your `test` executable.
- The command `make test` must build your `test` executable.
- The command `make dist` must create a file `cse303hw6.tar.gz` archive of your project files for turnin.
- The command `make clean` must clean up your directory by removing any `.o` files and built executables.
- There should be targets to make the `.o` object for each corresponding `.c` file in the system.

Unlike in the last assignment, this time we will grade some aspects of the internal correctness and style of your makefile. It should have a comment header, should appropriately variables to avoid redundantly repeating names of common files, and should have rule(s) to individually build each source file into its corresponding `.o` file. Your makefile should properly express dependencies between files, and it should only rebuild files as needed when a file is modified. In other words, if the developer modifies `memdebug.c`, the makefile should not recompile `memops.c` and so on.

All files in your program should produce no errors or warnings from `gcc -Wall` .

SVN Repository:

As part of your grade, your group must create a Subversion (svn) repository on `attu` and use it to store your code. Each team has been assigned to a Unix group named `cse303`***x***, where ***x*** is a single letter such as a, b, c, ... Your group should

have received this group name information by email. Each group has a corresponding project directory located at `/projects/instr/09sp/cse303/x` on `attu`. Everyone in your group already has group read/write permission for this directory. You should create a `svn` repository in this directory and store your code there. In other words, it should be possible for someone in your group (or the TAs / instructor) to issue the following command to check out your code:

```
$ svn co svn+ssh://attu.cs.washington.edu/projects/instr/09sp/cse303/x
```

Each member of the group will, of course, have separate working copies of the repository, and these can either be in individual directories on `attu`, or on any other machine(s) you use. Further information on the project area setup and remote access is located at the following URL:

- http://www.cs.washington.edu/lab/support/docs/uwcseinstr.html#SEC16

To receive full credit for the `svn` aspect of this assignment, you must demonstrate non-trivial usage of the repository in your development. In other words, you should regularly check in and update your files in the repository, not simply check them in once at the end when you are done with them. Checkins should be accompanied by comment messages describing the changes that are being submitted.

You should properly set the file permissions on your svn repository so that the files can be read by your group but not read or modified by any other students.

By Monday, May 25 at 9am, you must have set up your svn repository and checked in a version of all of the files listed at the start of this document. The files do not need to be completed; they can be short or empty stubs, but they should be there in the repository. We will check your repository's contents at this milestone as (a small) part of your grade.

Extra Credit:

For optional extra credit, you can choose to implement any of the functions listed below. Each is worth +2 extra points if it works fully, and you may be eligible for +1 extra point if it mostly/partly works. Put any such functions in a file named `extracredit.c` and their corresponding headers in a file named `extracredit.h`.

You may do any number of these extra credit features: some, all, or none. It is possible to earn a score over the maximum for the assignment; if so, you will be allowed to keep the extra points above 100% credit. Your score will not be capped.

`void* realloc303(void* p, int new_size)`

In this function you should move the contents stored at memory block `p` into a new block of memory that can hold at least *new_size* bytes of data, freeing the old block in the process, and return a pointer to the start of the new block. You may assume as a precondition that `p` points to a valid block of memory that was previously allocated by a call to `malloc303`, `calloc303`, or `realloc303`, unless it is `NULL`, described later. Notice that the client is not passing you the current size of memory occupied by `p`; you need to be able to figure that out yourself by examining your block list structures.

If the new size is larger than the block `p`'s old size, the contents of the new memory bytes after `p`'s old length are not initialized. As an optimization, you should elongate the existing allocated block of memory at `p` if possible, avoiding the need to move the contents. If this is not possible (for example, if there are not enough free bytes directly after `p`), then the contents stored at `p` should be moved into a new memory block of sufficient size.

If the new size is smaller than `p`'s old size, the remaining bytes are reclaimed so that they can be allocated again later. If the new size is less than or equal to 0, you should free and reclaim the memory stored at `p` and return `NULL`. Note that since blocks are allocated in multiples of 4 bytes, if the new size is smaller and is within 4 bytes of the original size of the block allocated to `p`, you won't have to reallocate anything and should just return the original pointer `p`.

If `realloc303` is unable to allocate a new block of the given size (if no free block exists that is large enough), your function should return NULL and should not move or free the existing contents of the block.

If `NULL` is passed for `p`, this function should allocate a new block of the given size and return a pointer to the start of it. Blocks should be allocated in multiples of 4 bytes, so for example the call of `realloc303(p, 13)` should return a block of 16 bytes of memory. If there is not enough free memory to allocate a block of the given size, you should return `NULL`.

`void valgrind303(void)`

In this function you should print debugging information about how many blocks have been allocated and freed, somewhat like a simpler version of the `valgrind` utility. This function would help a client discover statistics about memory they have allocated and about any memory leaks in their program. You should output the total number of blocks that have been allocated (not just currently allocated), the total number of blocks that have been freed, the total number of bytes that have been allocated/freed (usable bytes by the client, excluding your own list node headers), the total amount of free bytes of memory available, and the size of the largest individual free block in bytes.

Every time a block is allocated using your library, that counts 1 toward the total allocations performed, and the number of usable bytes returned to the client is added to the total bytes allocated. Every time a block is freed (even if this is done internally by your own code), that counts 1 toward the total frees performed, and the total number of usable bytes freed is added to the total bytes freed. You do not need to check for client errors such as freeing the same block twice.

Suppose that the current state of memory is the following, as would be reported by `print_blocks`:

```
 1: [0x22c8c4, used, 32B]
 2: [0x22c8e8, free, 616B]
 3: [0x22cb54, used, 40B]
 4: [0x22cb80, free, 320B]
```

The following might be the expected output of the `valgrind303` function when in such a state:

```
Total allocations performed = 10
Total frees performed       = 8
Total bytes allocated       = 1304B
Total bytes freed           = 1244B
Total free memory available = 936B
Largest free memory block   = 616B
Leaked blocks:
 1: [0x22c8c4, used, 32B]
 2: [0x22cb54, used, 40B]
```

Groups:

You must work with one partner on this assignment. You may work with anyone you like. If you do not notify us otherwise, we will assume that you are working with the same partner as on the last project, `hw5`. If this is not true, contact us as soon as possible. Your group should be registered using the Grouper tool found on the course web site.

Each group should submit only one set of files for the assignment. In most cases, both group members will be given the same grade for the program. If a group member did not do his/her share of the work, the other group member should contact the instructor immediately to discuss the situation. If there is a significant group problem, the instructor may ask to meet with one or both members to get more information and/or to possibly assess different grades to the members.

Please **place a comment at the top of each file** indicating which group member(s) worked on which parts of that file.

Lateness penalties will be applied individually to each student in the group based on how many late days that student has remaining. For example, if a group submits the project a day late and one student has a late day and the other does not, then the student with the late day will consume that late day, while the other student will receive a lateness deduction.

Development Strategy:

We suggest attacking this challenging program in roughly the following way:

- Set up all of your files, your svn repository, and your Makefile. Check in your stubs.
- If you like, you can implement an initial version of your memory allocation functions that just call the standard library functions like `malloc` and `free`. Of course you won't leave it this way for long, but it would allow you to write a client testing program and verify that it works, so that when you switch to your actual implementation, you'll know you have a working test case. Get your system to compile using your Makefile.
- Decide on the structure of your block list nodes and how you'll lay out the memory.
- Write an initial version of `malloc303`, ignoring the others such as `free303` or `calloc303`. Try running some very small test cases on this code. (Segfault time!)

- Write some of the `memdebug` functions so that you can print the state of the memory. This will be very helpful to you for debugging as you work on the rest of your system.
- Once you're able to allocate memory successfully, add other similar methods one at a time, such as `calloc303`. Test each one immediately after you write it, rather than moving on to the next one.
- Write an initial version of `free303` that frees the block but does not combine neighboring free blocks. Make sure that the block goes back onto the free list and can be used for allocations again later.
- Modify your `free303` to combine adjacent blocks. (It is easier to combine with an adjacent next block than an adjacent previous block, so do the next case first.) Test a variety of situations.

You will probably need a debugger such as `gdb` to figure out your trickier bugs, along with plenty of print statements.

Grading:

Over half of the points will come from the behavioral ("external") correctness and output of your program when run with various test cases. Getting correct output for any particular test case does not necessarily guarantee full credit. You are expected to thoroughly test your own program. You may not use C++ on this project.

A significant amount of points will also come from the style, design, and "internal correctness" of your code. Avoid redundancy and overly complex logic as much as possible. Minimize the use of global variables, and if you do have some globals, make them `static` as much as possible. Replace global variables with `#define` macros or with variables declared with the `const` keyword when possible. Use helper functions (`static` when possible) appropriately to split up your program to indicate its structure, to make it more readable, and to capture repeated code that would be redundant. Please also note that several of the functions you'll be implementing are very similar to other required functions. As appropriate, your functions should call each other to avoid redundancy.

Part of your grade comes from showing a good separation of functionality into the different files for the assignment.

Part of your grade comes from properly using header files in your multi-file program. The `.h` files should contain declarations for all functions and relevant data types related to the corresponding `.c` file. No file should include a `.c` file. Each `.c` file should be able to be successfully compiled into a `.o` file without warnings from `gcc -Wall -c`. Your `.c` files should not expose any global variables or functions that are not declared in the corresponding `.h` file; any other variables/functions not in the `.h` file should be declared `static` to protect them from outside modification.

You may use libraries `stdio`, `stdlib`, `stdbool`, `string`, `ctype`, `time`, and `unistd` on this assignment. You may not use any other pre-written data structures or other libraries without instructor permission.

Part of your grade will be based on appropriate usage and allocation of memory. You should show an understanding of when to allocate data on the stack vs. on the heap, and proper use of pointers. You may not use the built-in `malloc` or `calloc` anywhere in this assignment. In general you may assume that the computer has enough available memory to accommodate all data structures you will need to create.

Format your code nicely with proper indentation, spacing, and clear variable names. Place a descriptive comment heading at the top of each file describing the assignment as well as describing the purpose of that file. Place a descriptive comment header atop each function, type declaration, global variable declaration, and brief comments throughout your code inside methods explaining what each major section of code is doing. In particular, thoroughly document the data format and structures used to manage your memory block list, such that a grader would not need to trace through your code to understand it. The `.c` files should generally have more detailed commenting than the `.h` files. Your comments should be written in your own words and should not be copied directly from this document.

```
$ wc -l *.c *.h Makefile
  110 extracredit.c
  148 memalloc.c
   56 memdebug.c
   44 memops.c
  182 test.c
   11 extracredit.h
   16 memalloc.h
   17 memdebug.h
   25 memimpl.h
   12 memops.h
   34 Makefile
  655 total
```

For reference, the box at right shows the number of lines in our solution files, including comments and blank lines. You do not need to match these numbers; they are just a rough guideline.

Different computers can behave differently with the same C program; for full credit, your program must compile and run successfully on `attu` or on the basement lab computers.