

CS 2110 Summer 2017

Homework 11

Rules and Regulations

General Rules

1. Starting with the assembly homeworks: Any code you write (if any) must be clearly commented and the comments must be meaningful. You should comment your code in terms of the algorithm you are implementing; we all know what the line of code does.
2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.
3. Please read the assignment in its entirety before asking questions.
4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere near the top in other files unless otherwise noted.
2. When preparing your submission you may either submit the files individually to T-Square or you may submit an archive (zip or tar.gz only please) of the files (preferred). You can create an archive by right clicking on files and selecting the appropriate compress option on your system.
3. If you choose to submit an archive please don't zip up a folder with the files, only submit an archive of the files we want. (See **Deliverables**).
4. Do not submit compiled files, that is, `.class` files for Java code and `.o` files for C code. Only submit the files we ask for in the assignment.
5. Do not submit links to files. We will not grade assignments submitted this way as it is easy to change the files after the submission period ends.

Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know ***IN ADVANCE*** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.
2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via T-Square. When you submit the assignment, you should get an email from T-Square telling you that you submitted the assignment. If you do not get this email, that means that you did not complete the submission process correctly. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension, you will still turn in the assignment over T-Square.
3. There is a 6-hour grace period added to all assignments. You may submit your assignment without penalty up until 11:55PM, or with 25% (max points) penalty up until 5:55AM. *So what you should take from this is not to start assignments on the last day and plan to submit right at 11:54AM.* You alone are responsible for submitting your homework before the grace period begins or ends; neither T-Square, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until 11:54PM. The penalty for submitting during the grace period (25%) or after (no credit) is non-negotiable. **This 25% is off of the maximum grade you would get**, this means that if you would have gotten a 50% then your late grade will be a 25%.

Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class.

Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative. In addition, many, if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using electronic computer programs to find evidence of unauthorized collaboration.

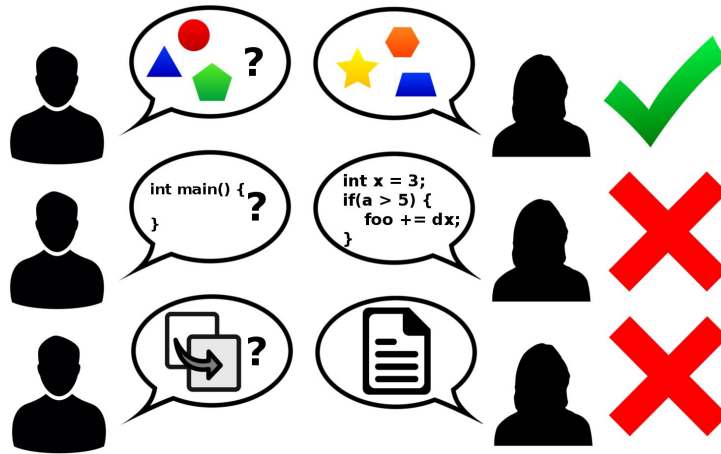
What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you

supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, as well as help each other debug code. What you shouldn't be doing, however, is paired programming where you collaborate with each other on a low level. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, and it is often the case that the recipient will simply modify the code and submit it as their own.



Objectives

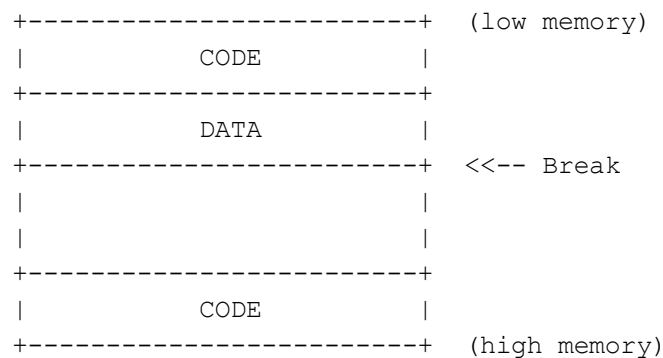
The goal of this assignment is to implement a **memory allocator**. A memory allocator is code that is linked with user programs and provides functions to allocate and deallocate blocks of dynamic memory. In other words, you will be writing the heart and soul of two big dynamic memory allocation functions: `malloc()` and `free()`.

Overview

Warning: Your submission must compile with our flags or we will simply not grade it and give it a zero. After you submit download your submission again and unzip in a clean directory and build it.

The Basics

It is the job of the memory allocator to process and satisfy the memory requests of the user. But where does the allocator get *its* memory? Let us recall the structure of a program's memory footprint:



When a program is loaded into memory there are various segments created for different purposes: code, stack, data, etc. In order to create some dynamic memory space, otherwise known as the **heap**, it is possible to move the **break**, which is the first address after the end of the process's uninitialized data segment. A function called `brk` is provided to set this address to a different value. There is also a function called `sbrk` which moves the break by some amount specified as a parameter.

For simplicity, a wrapper for the system call `sbrk` has been provided for you in the file named `my_sbrk.c`. Make sure to use this call rather than a real call to `sbrk`, as doing this can potentially cause a lot of problems. Note that any problems introduced by calling the real `sbrk` will not be regraded, so make sure that everything is correct before turning in.

If you glance at the code for `my_sbrk`, you will quickly notice that upon the first call it always allocates 8 KB. For the purposes of your program, you should treat the returned amount as whatever you requested. For instance, the first time I call `my_sbrk` it will be done like this:

```
my_sbrk(SBRK_SIZE); /* SBRK_SIZE = 2 KB */

+-----+
|               8 KB               |
+-----+
^
|
`---- The pointer return to me by my_sbrk
```

Even though you have a full 8 KB, you should treat it as if you were only returned `SBRK_SIZE` bytes. Now when you run out of memory and need more heap space you will need to call `my_sbrk` again. Once again, the call is simply:

```
my_sbrk(SBRK_SIZE);

+-----+-----+
| 2 KB  | 6 KB  |
+-----+-----+
^
|
`---- The pointer returned to me by my_sbrk
```

Notice how it returned a pointer to the address after the end of the 2 KB I had requested the first time. `my_sbrk` remembers the end of the data segment you request each time and is able to return that value to you as the beginning of the new data segment on a following call. Keep this in mind as you write the assignment!

Block Allocation

Trying to use `sbrk` (or `brk`) exclusively to provide dynamic memory allocation to your program would be very difficult and inefficient. Calling `sbrk` involves a certain amount of system overhead, and we would prefer not to have to call it every single time a small amount of memory is required. In addition, deallocation would be a problem. Say we allocated several 100 byte chunks of memory and then decided we were done with the first. Where would the break be? There's no handy function to move the break back, so how could we reuse that first 100 byte chunk?

What we need are a set of functions that manage a pool of memory allowing us to allocate and deallocate efficiently. Typically, such schemes start out with no free memory at all. The first time the user requests

memory, the allocator will call `sbrk` as discussed above to obtain a relatively large chunk of memory. The user will be given a block with as much free space as he requested, and if there is any memory left over it will be managed by placing information about it in a data structure where information about all such free blocks is kept. We will call this structure the **freelist**, and revisit it a little later.

In order to keep track of allocated blocks we will create a structure to store the information we need to know about a block. Where should we put this structure? Can we simply call `malloc` to allocate space for the information?

No we can't! We're writing `malloc`: we can't use it or we'd end up with infinite recursion. However, there's an easier way that will keep our bookkeeping structure right with the data we're allocating for easy access.

The trick we will use is storing this information, called **metadata**, about the block inside the block itself! For this assignment, we are also implementing canaries to make sure that accesses do not overrun their space. We will have two canaries, one right before the space the user will use and one right after. Each canary will be an `int` with the value defined in the `#define canary`. We still want the user to have as much space as they requested, though, so when they want a block of size `x` we will actually allocate a block of size: `x + sizeof(the metadata) + 2 * sizeof(int)`.

What this actually looks like:

```
<----- Memory Chunk with metadata ----->

+-----+-----+-----+-----+
| metadata | canary | free space for the user | canary |
+-----+-----+-----+-----+
                        ^
                        |
                        `--- pointer returned to the user
```

The user still only sees a block of the size they requested, but the `metadata` hangs out in the space right before that block, so our allocation and deallocation algorithms can use it. The canaries surround the space the user gets so that, if they overwrite the amount of space they are given, your functions can detect it and print an error message on `free`. Now that you know this, you may see why we're so bothered by writing over the bounds of dynamically allocated blocks: write over the `metadata` and chaos ensues!

Here's a list of the things that your `metadata` will need to contain:

1. the real size of the block (again, this includes the size of the `metadata` structure itself)
2. the size that the user asked for
3. a pointer to the previous block in the `freelist`
4. a pointer to the next block in the `freelist`

Additionally, remember that you will have a canary after the `metadata` and after the end of the free space reserved for the user. Note that the `metadata` does not contain the canary. For ease of reading, the included `struct` definition, found in `my_malloc.h` has been pasted below for a better overview of what we're dealing with:

```
typedef struct metadata {
    short block_size;
    short request_size;
    struct metadata *prev;
    struct metadata *next;
} metadata_t;
```

The Freelist

When we allocate memory or take pieces of blocks we already allocated there may be blocks we don't automatically use. For this reason, we keep a structure called the **freelist** that holds metadata about blocks that aren't currently in use.

The freelist is a linked list of blocks which should be defined as a file `static` variable. To help you out we have already defined a variable `freelist` in the file `my_malloc.c` for you.

```
static metadata_t* freelist;
```

Remember that, since this variable is declared `static`, it will be automatically initialized to 0 by the compiler for us! This means that we do not have to do any special work to initialize the `freelist` (or `head`) pointer to `NULL`.

Since the freelist actually holds pointers to metadata, and metadata structures all have pointers to the `previous` and `next` metadata, the freelist lends itself to a linked list implementation. Keep in mind that `previous` and `next` don't necessarily mean consecutive: these pointers just exist to maintain links between blocks that are free. It is okay if a high address comes before a low address in the freelist for any block size, so long as the freelist holds all the blocks that are free at any time. You may choose to sort the freelist if you think it will make your job easier, but you do not have to.

Simple Linked List - Allocating

When we first allocate space for the heap, it is in our best interest not to just request what we need immediately but rather to get a sizable amount of space, use a piece of it now, and keep the rest around in the freelist until we need it. This reduces the amount of times we need to call `sbrk`, the real version of which, as we discussed earlier, involves significant system overhead. So how do we know how much to allocate, how much to give to the user, and how much to keep?

For this assignment we will request blocks of size 2048 from `my_sbrk`. We don't want to waste space, though, so we want to give to the user the smallest size block in which their request would fit. For example, the user may request 256 bytes of space. It is tempting to give them a block of size 256, but remember we are also storing the metadata inside the block. If our metadata and canaries takes up 12 bytes, we need at least a $256 + 12 = 268$ byte block.

How do we get from one big free block of size 2048 to the block of size 268 we want to give to the user? In this simple implementation, you will traverse the freelist to find the smallest block in the freelist that is larger than what we want, and *split* off however much you need from the front.

So we start with one big block in the freelist:

```

+-----+
| block_size  = 2048;          |
| request_size = 0;          |
| next        = NULL;         |
| prev        = NULL;         |
+-----+
^
+-----+
| FL | -----'
+-----+

```

We want a block of size 268, so we look at the smallest size block that we can split, which is the 2048 block. The freelist now looks like this after splitting the 2048 block:

```

+-----+-----+
|-----| block_size  = 1780;          |
|-----| request_size = 0;          |
|-----| next        = NULL;         |
|-----| prev        = NULL;         |
+-----+-----+
^
+-----+
| FL | -----'
+-----+

```

And you will have a temporary pointer to the block you will return to your user:

```

+-----+ +-----+
| Pointer | --> | block_size  = 268; |
+-----+      | request_size = 256; |
+-----+      +-----+

```


Don't forget to set both canaries and move the pointer to the beginning of the space the user can actually use, after the end of the metadata and canary, before returning the block to the user.

Tips

1. Make helper functions if you want! Consider functions to add and remove nodes from the freelist
2. **Test Test Test!** Test a bunch! Test after having `malloc`-ed multiple things! Test `free`-ing in a different order than `malloc`-ed! Test actually using the data returned!

Simple Linked List - Deallocating

When we deallocate memory, we simply check the block's canaries and return the block to the freelist in the appropriate position.

Note: We don't clear out all the data. That really just takes too long when we're not supposed to care about what's in memory after we free it anyway. For all of you who were wondering why sometimes you can still access data in a dynamically allocated block even after you call `free` on its pointer, this is why!

We like the freelists to contain fairly large blocks so that large requests can be allocated quickly, so if the block on either side of the block we're freeing is also free, we can coalesce them, or join them into the bigger block like they were before we split them. How do we know what blocks we can join with?

- The left one: left block's address + left block's (real) size == your block's address
- The right one: your block's address + your block's (real) size == right block's address

This means you will have to traverse the freelist to find the left and right side blocks to see if they are free and can be joined.

my_malloc

You must write your own version of `malloc` that implements simple linked-list based allocation. As a refresher, this is what entails:

1. Figure out what size block you need to satisfy the user's request. Remember that the `real_size` field in our `metadata` structure includes the size of the metadata structure itself, so take the user size, `sizeof(metadata_t)`, and `sizeof(canary) * 2` into account. (If this size is over request size, set the right `ERRNO` and return `NULL`).
2. Now that we have the size we care about, we need to iterate through our `freelist` to see if there is a free block big enough in that list. If there is one, we need to split off the portion we will use, remove that portion from the freelist, set its canaries, and return the block to the user. Note that the user should not be returned the pointer to our `metadata` structure, so the pointer we actually return to the user is something like: `metadata_pointer + sizeof(metadata_t) + sizeof(int)`.

Note: Remember that you want the address you return to be `sizeof(metadata_t) + sizeof(canary)` bytes away from the `metadata` pointer. Pointer arithmetic can be unforgiving if our pointer was of type `int*` (which it's not, this is just an example), adding 4 to it would result in an address that is 4 `ints` away from the original address. `ints` are not byte sized, so maybe a cast should be involved?

If there is no block large enough, use `my_sbrk` to get more memory.

A couple of other things that are worth noting for the `my_malloc` functions:

1. The first call to `my_malloc` should call `my_sbrk`. Note that `malloc` should call `my_sbrk` when it doesn't have a block to satisfy the user's request anyway, so this isn't a special case.
2. If the user request exceeds 2048 bytes (note that this number is what is calculated *AFTER* `my_malloc` adds on the obligatory `sizeof(metadata_t) + 2 * sizeof(canary)`), then return `NULL` and set the error code.
3. In the event that there are no blocks currently available in the freelist that satisfy the user's request (note that you must attempt the procedure described above before determining this), then you should issue a call to `my_sbrk` to expand the heap size by `SBRK_SIZE` bytes. Failure to use this macro to expand the heap may result in a lower grade than you think you deserve. Please make *SURE* to use this macro when calling `my_sbrk` to avoid any possible problems. Also note that in the event that `my_sbrk` returns failure (by returning `NULL`), you should return `NULL` and set the error code.
4. If there is a block large enough to satisfy the request, but it's not large enough where you could split off the block you need and still have enough left over for another block (that is, having less than `sizeof(metadata) + 2 * sizeof(int) + 1`), then you should return the entire block to the user with the block size set to the full size of the block. The canary should still be at the end of the space the user asked for, so in this case there will actually be some space in the block after the canary. This is why we are having you record both the block size and the request size, so you can find both the end of the block and the ending canary.

my_free

You are also to write your own version of `free` that implements deallocation. This means:

1. Calculate the proper address of the block to be freed, keeping in mind that the pointer passed to any call of `my_free` is the pointer that the user was working with, not the pointer to the block's metadata.
2. Check the canaries of the block, starting with the one by the `metadata` (so that, if it is wrong, you do not try to use corrupted `metadata` to find the second canary) to make sure they are still their original value. If the canary has been corrupted, set the `CANARY_CORRUPTED` error and return.
3. Attempt to merge the block with blocks that are consecutive in address space, if those blocks are free. Don't forget to try to merge with the blocks to both its left and right in memory. Finally, place the resulting block in the `freelist`.

Error Codes

For this assignment, you will also need to handle cases where users of `my_malloc` do improper things with their code. For instance, if a user asks for 12 gigabytes of memory, this will clearly be too much for your 8 KB heap. It is important to let the user know what they are doing wrong. This is where the `enum` in the `my_malloc.h` comes into play. You will see the four types of error codes for this assignment listed inside of it. They are as follows:

<code>NO_ERROR</code>	set whenever <code>my_malloc</code> or <code>my_free</code> complete successfully
<code>OUT_OF_MEMORY</code>	set whenever the user request cannot be met because there's not enough heap space
<code>SINGLE_REQUEST_TOO_LARGE</code>	set whenever the user's request is beyond our biggest block size
<code>CANARY_CORRUPTED</code>	set whenever either canary is corrupted in the freed block

Inside the `.h` file, you will see a variable of type `my_malloc_err` called `ERRNO`. Whenever any of the cases above occur, you are to set this variable to the appropriate type of error. You may be wondering what happens if a single request is too large AND it causes `my_malloc` to run out of memory. In this case, we will let the `SINGLE_REQUEST_TOO_LARGE` take precedence over `OUT_OF_MEMORY`. So in the case of a request of 9 KB, which is clearly beyond our biggest block and total heap size, we set `ERRNO` to `SINGLE_REQUEST_TOO_LARGE`.

Code Documentation

YOU MUST COMMENT YOUR CODE FOR THIS ASSIGNMENT! We don't expect every line of code to have comments, but we expect that you will have some comments so that we can figure out what you are trying to do in your code. Do not ask your TA's exactly how many comments are necessary: use your best judgment.

Example:

```
void* my_malloc(size_t size) {
    /* Declaration of variables*/
    int a,b;
    int *c;

    /* Looks for large-enough block */
    ...your code here...
}
```

Testing

You've been provided with two ways to test:

1. A `test.c` file where you can write some of your own tests that get run in a loop and give you timing information
2. An autograder that will be *very similar* to the one we use to grade your work! (we reserve the right to change the autograder to fix bugs or concerns that come up; we're not trying to trick you)

The `test.c` file has an empty main function. Feel free to modify anything in `test.c` to your heart's content to test your code as strictly as possible.

The autograder has been included in a subfolder of this assignment – just navigate into that folder and run `make` to have the autograder evaluate your homework! Feel free to modify the grader's test files to add your own tests if you wish!

The autograder runs a bunch of tests in different processes to run just one specify it as a parameter (`./grader 4`). Run it in `gdb` with `gdb grader` and then `run 5` (or whatever test number) after removing the alarm function call in `grader.c` (which kills the process after five seconds to deal with infinite loops, but will also kill the process inside of `gdb`).

The Assignment

1. You must use the allocation system exactly as described in this file.
2. The functionality for `my_malloc` and `my_free` should reside in the provided file `my_malloc.c`.
3. We have provided you with a blank tester file (`test.c`). The supplied `Makefile` assumes you will use `test.c` to test your code. If you choose to use a different file you will have to modify your `Makefile` to do so. Note that we are asking you to submit a library, so no file with a `main` function should be submitted (No need to submit `test.c`). But more importantly, make sure there is no `main` file in any of your other code (ex: `my_malloc.c`). If any linking errors are encountered while attempting to grade, including 'multiple definition of main', you WILL receive a ZERO.
4. You may NOT core dump (cause a segmentation fault).
5. You may NOT maintain the freelist/heap in such a way that normal usage of your library would cause a core dump (i.e. returning a block of size 4 when a block of size 32 was requested, returning a block which isn't properly aligned, etc.)
6. You may not call any C library functions which are in any way related to memory management (i.e. `malloc`, `calloc`, etc.).
7. Your code must compile cleanly with the flags provided in the `Makefile`. If your code does not compile with these flags, or if it cannot be compiled using the `Makefile`, you WILL receive a ZERO.
8. You should turn in all of your files. Above we told you there was no need to turn in `test.c`, but if you are having trouble with the assignment it would not hurt for the grader to see your test code (however, this is not a requirement).
9. Test as much as you want: Just make sure no tester output comes out when we run your library files. Bon courage!
10. Be sure to comment your code thoroughly! You will lose points if it is not commented!

Deliverables

Please submit your assignment by the deadline on T-Square. Run the command `make submit` and submit the following file:

`hw11_submission.tar.gz`

Please only submit the above file.

Frequently Asked Questions

1. For code like `void* ptr = my_malloc(18);` does the block associated with `ptr` have a size of `18 + sizeof(metadata_t) + 2 * sizeof(int)`?

Yes, that is correct.

2. Why am I getting a segfault in....

Use GDB to debug! Also see FAQ #7.

3. The homework assignment says to just call `my_sbrk` again.
Won't this mean we then have two heaps?

Not exactly: It will expand the heap by another 2 KB. You don't get two heaps. Once it has been expanded to 8 KB, calls to `my_sbrk` will return `NULL`.

4. When splitting a block into two blocks, do we use `malloc` to create a `metadata` for the second block?

No! The use of the already coded `malloc` in C is expressly forbidden! Instead, you simply use pointer arithmetic to get a pointer into the middle of the block you're splitting and then set the data as necessary so that they function as two blocks.

5. Can we build our freelists with list heads/dummy nodes?

No. No dummy nodes. The autograder checks the state of the freelist and if you have dummy nodes it will throw it off.

6. Should we first initialize the freelist to `NULL`?

No, it is static and is therefore already `NULL`.

7. How do I use `gdb`?

There is a `debugging.txt` text file that comes with the assignment that gives an introduction to using `gdb`. Some useful `gdb` one liners for this assignment:

<code>x/16xw freelist</code>	- examine 16 hexadecimal words by the freelist
<code>print *freelist</code>	- print the freelist (pretty printed as a struct)
<code>display *freelist</code>	- this will print the freelist after every <code>gdb</code> command
<code>watch freelist</code>	- sets a breakpoint anytime freelist is modified

Note that you can call the above commands on any variable you're having issues with, not just the freelist.