CS 2110 Homework 11

Austin Adams, Kyle Murray, Cem Gökmen, and Maddie Brickell

Fall 2017

Contents

1 Assignment

1.1 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.

```
+-------------------+ (low memory)
|       CODE        |
+-------------------+
|       DATA        |
+-------------------+ <-- Break
|                   |
|                   |
+-------------------+
|       STACK       |
+-------------------+ (high memory)
```

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 KiB. 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 returned to me by my_sbrk
```

Even though you have a full 8 KiB, 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!

1.2 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 they 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'll call this structure the free list, and we'll 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 that we will store 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. Canaries are integers that we will generate using information about the block which will be stored with block and used for verification purposes (`https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries`, though note that the one we implement will be a canary for memory allocated by malloc, not static arrays). We will have two canaries, one right before the body of the block (already included in the metadata, we'll call this the head canary) and one right after, called the tail canary. Each canary will be of type `unsigned int` with the value generated from the block address, the block next pointer, and the block size. As a result, our block will have three parts: the metadata (which also contains the head canary), the block body (the area that the user is given), and the tail canary. Since we still want the user to have as much space as they requested, when they request a block of size `n` we will take that to be the size of just the body of the block, and in total we will actually allocate a block of size `n + sizeof(the metadata) + sizeof(the tail canary)`. What this actually looks like:

```
          <-------------- Single allocated block of memory ------------>

---------+----------+-------------------------------------+-------------+---------
         | metadata | block body (area the user will use) | tail canary |
---------+----------+-------------------------------------+-------------+---------
                     ^
                     |
                     \___ pointer returned to the user
```

The user gets a pointer to the body and thus only gets a memory block of the size they requested, but the metadata hangs out in the space right before the body so our allocation and deallocation algorithms can use it. The canaries surround the space the user gets (the body) 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 size of the *whole* block in number of bytes (not only the body that the user requested, but the metadata and the tail canary too)
2. a pointer to the next block in the freelist
3. the head canary

Additionally, remember that you will have a second canary after the end of the free space reserved for the user. Note that the metadata contains the head canary as the last entry, so it will be placed in memory right before the block. You can think of the user data (the block body) as being sandwiched between the two canaries. If the user tries to go beyond their bounds, they will mess up one of the canaries.

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 {
    struct metadata *next;
    unsigned int size;  // size in number of bytes
    unsigned int canary;
} metadata_t;
```

The canaries (both the head and the tail) are generated as so: the metadata pointer (the pointer to the current block) and `CANARY_MAGIC_NUMBER` (a macro defined as the number `0xE629`) are XOR-ed together and the block size is subtracted. Both canaries will have the same value (note that the tail canary takes some pointer arithmetic to get to). You can grab the integer of the metadata pointer by casting it to a `uintptr_t`:

```
block->canary = ((uintptr_t)block ^ CANARY_MAGIC_NUMBER) - block->size;
```

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

```
metadata_t *freelist;
```

Since the freelist actually holds pointers to metadata, and metadata structures all have pointers to the 'next' metadata, the freelist lends itself to a linked list implementation. **For this assignment, you must keep the freelist sorted in ascending block size order.** In the example below, the 512 byte block is the head and points to the 700 byte block, which points to the 1080 byte block:

```
                                             head of the freelist
                                                      |
                                                      v
 -------------------------------------- ---------- ----------------
|           1080 bytes free            |***used***| 512 bytes free |
 -------------------------------------- ---------- ----------------
  ^                                                   |
  |  /------------------------------------------------/
  |  v
 --------------------
|   700 bytes free   |
 --------------------
```

1.4 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 bytes 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 that is 256 bytes, but remember we are also storing the metadata inside the block. If our metadata and canaries takes up $8 + 4 + 4 + 4 = 20$ bytes, we need at least a

$$256 + 20 = 276$$

byte block.

Note that the size of your metadata will vary based on your computer's architecture. On 64-bit computers pointers will be 8 bytes, hence 20 total bytes for metadata and canaries. 32-bit computers use 4 byte addresses. We use `sizeof()` to avoid depending on the platform, and a macro named `TOTAL_METADATA_SIZE` is given to you so you don't have to worry about it.

How do we get from one big free block of size 2048 bytes to the block of size 276 bytes we want to give to the user? In this simple implementation, you will traverse the freelist to find the best block to satisfy the user's request, which should be equal or greater than the size requested, and "split" off however much you need from the front or the back. For this assignment, you can choose whether you want to split from the front or the back — the tester will work with either implementation. So let's say we start with one big block in the freelist:

```
----      ----------------------------------------
 FL | --> | next=NULL; size=2048 bytes           | --> NULL
----      ----------------------------------------
```

We want a block of size 276 bytes, so we look at the smallest size block that we can split, which is the 2048 byte block. The free list now looks like this after splitting the 2048 byte block:

```
----      ----------------------------------------
 FL | --> | next=NULL; size=1772 bytes           | --> NULL
----      ----------------------------------------
```

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

```
------------------         ----------------------------
 Your Tmp Pointer | -----> | size=276 bytes           |
------------------         ----------------------------
```

Don’t forget to set both canaries and move the pointer to the beginning of the space the user uses after the end of the metadata and canary before returning the block body pointer to the user.

1.5 Simple Linked List: Deallocating

When we deallocate memory, we simply check the block’s canaries and return the block to the free list in the appropriate position. When the user calls the free function with a block body pointer, we do some pointer arithmetic to find the starting point of the entire block (i.e. the metadata). Notice 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 side one will have its address + its size = your block’s address, and the right one will be your block’s size + it’s address.

1.6 my_malloc()

You are to write your own version of malloc that implements simple linked-list based allocation:

1. Figure out what size block you need to satisfy the user's request (should just be the input param). Add `TOTAL_METADATA_SIZE` to the requested block body size to include the size of the metadata and the tail canary, that will be the real block size we need. (Note: if this size in bytes is over `SBRK_SIZE`, set the error `SINGLE_REQUEST_TOO_LARGE` and return `NULL`. If the request size is 0, then mark `NO_ERROR` and return `NULL`).

2. Now that we have the size we care about, we need to iterate through our freelist to find a block that bests fits. Best fit is defined as a block that is exactly the same size, or the smallest block big enough to split and house a new block (`MIN_BLOCK_SIZE` is defined for you), or any block bigger than the size requested.

   (a) If the block is exactly the same size, you can simply remove it from the freelist, set the canaries, and return a pointer to the body of the block.

   (b) If the block is big enough to house a new block, we need to split off the portion we will use. Remember: pointer arithmetic can be tricky, make sure you are casting to a `uint8_t *` before adding the size (in bytes) to find the split pointer!

   (c) If the previous two conditions fail, then the smallest block that's big enough should be selected and removed from the freelist, its canaries set, and a pointer to the body of the block returned. Any extra space not big enough to split into a new block (i.e. one that cannot fit the metadata, a 1-byte body, and a tail canary) should be included as part of the user block before the end canary.

   (d) If no suitable blocks are found at all, then call `my_sbrk()` with `SBRK_SIZE` to get more memory. After setting up its metadata and and merging it with a freelist block directly to the left of it if such a block exists (in this assignment, there must never be two different blocks in the freelist who are directly adjacent in memory), go through steps (a)-(c).

   Remember that you want the address you return to be at the start of the block body and not the metadata, and this is `sizeof (metadata_t)` bytes away from the metadata pointer. Since pointer arithmetic is in multiples of the sizeof the data type, you can just add 1 to a pointer of type `metadata_t*` pointing to the metadata to get a pointer to the body.

3. 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.

4. 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 set the error code `OUT_OF_MEMORY` and return `NULL`.

1.7 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 a pointer to the block body and not to the block's metadata.

2. Check the canaries of the block, starting with the head canary (so that if it is wrong you don't try to use corrupted metadata to find the tail 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 with it if those blocks are free. That is, try to merge with the block to its left and its right in memory if they are in the freelist. Finally, place the resulting block in the freelist.

Just like the `free()` in the C standard library, if the pointer is `NULL`, no operation should be performed.

1.8 my_realloc()

You are to write your own version of realloc that will use your `my_malloc()` and `my_free()` functions. `my_realloc()` should accept two parameters, `void* ptr` and `size_t size`. It will attempt to change the size of the memory block pointed to by `ptr` to `size` bytes, and return a pointer to the beginning of the extended memory block. This might be the same pointer if free memory was available after the block, or a pointer to a different location in memory if the existing block could not be extended. In the latter case, the data in the original block will be copied to the new block, and the original block freed.

Your `my_realloc()` implementation must have the same features as the `realloc()` function in the standard library. For details on what `realloc()` does and edge cases involved in its implementation, read the realloc manual page by opening a terminal in Ubuntu and typing `man realloc`.

If `my_malloc()` returns `NULL`, do not set any error codes (as `my_malloc` will have taken care of that) and just return `NULL` directly.

1.9 my_calloc()

You are to write your own version of calloc that will use your `my_malloc()` function. `my_calloc()` should accept two parameters, `size_t nmemb` and `size_t size`. It will allocate a region of memory for `nmemb` number of elements, each of size `size`, zero out the entire block, and return a pointer to that block.

If `my_malloc()` returns `NULL`, do not set any error codes (as `my_malloc()` will have taken care of that) and just return `NULL` directly.

1.10 Error Codes

For this assignment, you will also need to handle cases where users of your 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 kilobyte 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:

- **NO_ERROR**: set whenever `my_calloc()`, `my_malloc()`, `my_realloc()`, and `my_free()` complete successfully.
- **OUT_OF_MEMORY**: set whenever the user's request cannot be met because there's not enough heap space.
- **SINGLE_REQUEST_TOO_LARGE**: set whenever the user's requested size plus the total metadata size is beyond `SBRK_SIZE`.
- **CANARY_CORRUPTED**: set whenever the sum of the user's requested size and the total metadata size exceeds `SBRK_SIZE`.

Inside the .h file, you will see a variable of type `enum my_malloc_err` called `my_malloc_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 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 9kb, which is clearly beyond our biggest block and total heap size, we set `ERRNO` to `SINGLE_REQUEST_TOO_LARGE`.

1.11 Using the Makefile

Before running the Makefile, you need to install Check, a C unit testing library the provided tests use. The following command should install the packages you need for this homework:

```
sudo apt-get install pkg-config check gdb valgrind
```

You can run the provided tests with `make run-tests`, run gdb with `make run-gdb`, and run valgrind with `make run-valgrind`. I'd run the tests first, then do my debugging with gdb, and *then* use valgrind to check for memory leaks/errors.

1.12 Deliverables

Submit only `my_malloc.c`. Please don't zip it.

2 Frequently Asked Questions

1. **I have a segfault, will you debug it for me?**

   No, debug it yourself with gdb. Here are some gdb tutorials:

   - `https://www.cs.cmu.edu/~gilpin/tutorial/`
   - `http://www.cs.yale.edu/homes/aspnes/pinewiki/C%282f%29Debugging.html`
   - `http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html`
   - `http://heather.cs.ucdavis.edu/~matloff/debug.html`
   - `http://www.delorie.com/gnu/docs/gdb/gdb_toc.html`

2. **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.

3. **Should we first initialize the freelist to NULL?**

   No, it is static and is therefore already NULL

4. **The assignment says to just call `my_sbrk()` again. But won't this mean we then have 2 heaps?**

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

5. **Are the provided tests comprehensive?**

   Not at all, and the percentage passed is not necessarily your grade. When grading, we will test against even more test cases, and they will be weighted differently than in the provided tester.

   We'll pin a Piazza thread where students can share additional test cases.

6. **Can I use the `malloc()` from the C standard library?**

   no

3 Rules and Regulations

3.1 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.

3.2 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 in the file, near the top, for 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.

3.3 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. 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% 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.

3.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

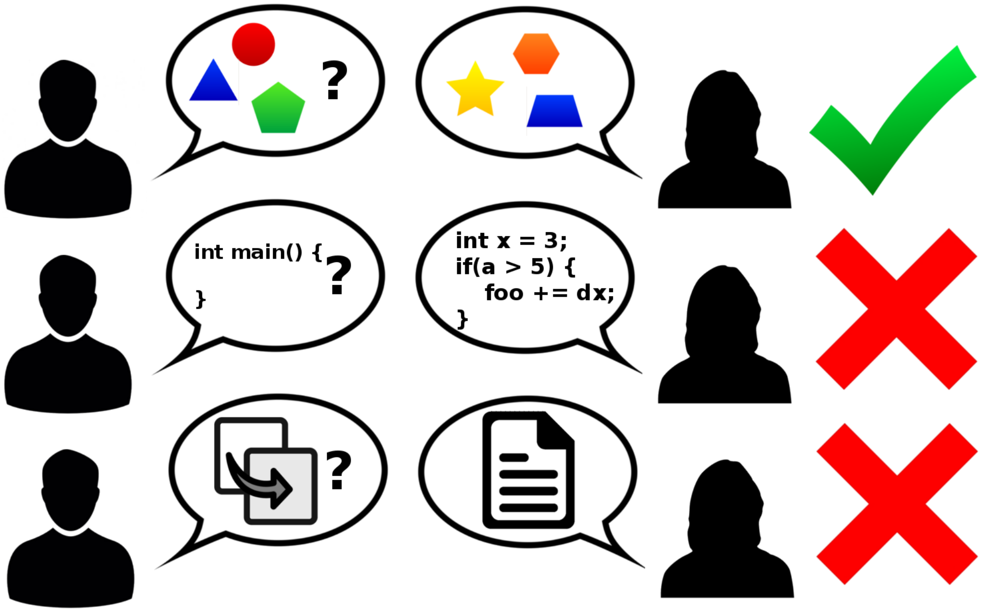
Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to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

3.5 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.



```
int main() {

}
```

```
int x = 3;
if(a > 5) {
    foo += dx;
}
```

Figure 1: Collaboration rules, explained