CS 2110 Homework 10
Implementing Malloc

Prabhav Gupta, Richard So, Rohan Bafna, Jordan Miao

Spring 2024

Contents

1 Overview **2**
- 1.1 Purpose . . . 2
- 1.2 Tasks . . . 2
- 1.3 Criteria . . . 2

2 Assignment **2**
- 2.1 The Basics . . . 2
- 2.2 Block Allocation . . . 4
- 2.3 The Freelist . . . 5
- 2.4 Simple Linked List: Allocating . . . 6
- 2.5 Simple Linked List: Deallocating . . . 7
- 2.6 Helper Methods . . . 9
- 2.7 my_malloc() . . . 10
- 2.8 my_free() . . . 11
- 2.9 my_realloc() . . . 11
- 2.10 my_calloc() . . . 12
- 2.11 Error Codes . . . 12
- 2.12 Running the Autograder and Debugging . . . 12
- 2.13 Deliverables . . . 13

3 Frequently Asked Questions **13**

4 Rules and Regulations **14**
- 4.1 Academic Misconduct . . . 14

1 Overview

1.1 Purpose

The purpose of this assignment is to give you a deeper understanding of some of the most important functions in C: malloc, free, calloc, and realloc. These four functions are key to writing programs that can handle dynamic amounts of data, like strings of arbitrary lengths and practically unlimited amounts of user input. Knowing how to use these functions is vital, but it is also important to understand how they work – especially if you are a Devices, Info Internetworks, or Systems/Architecture student who will be taking CS 2200 (and possibly also CS 3210) in a later semester.

1.2 Tasks

You will be implementing the backend of the four core functions for dynamic allocation: **malloc(), free(), calloc(), and realloc()**. While calloc() and realloc() can be implemented in terms of malloc() and free(), malloc() and free() must be implemented from scratch, using a freelist of pointers to keep track of available memory and the sbrk function to obtain more memory when needed.

There are also several helper functions included that you are optional (but highly recommended) to implement. You are encouraged to implement these helper functions first and use them when implementing the fore core functions for dynamic allocation. You are welcome to add any additional helper functions. We provided autograder support for these helper functions but their implementation will not be counted for a grade.

1.3 Criteria

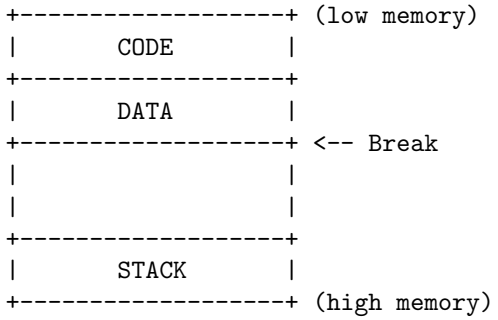
You will be graded using an autograder with several test cases, similar to Homework 9. Each of the four functions will be tested with cases that test different behaviors; for example, one case might expect malloc to find and return a perfectly sized free block, while other cases expect malloc to split a larger block into two.

2 Assignment

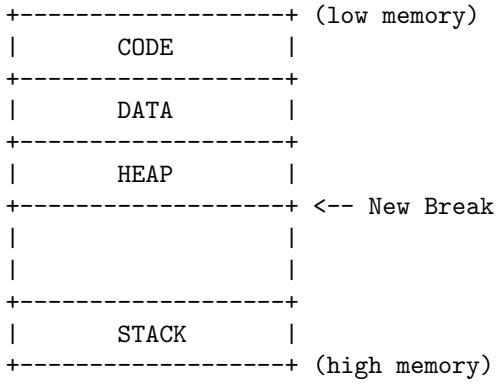
In this assignment, you will be writing the dynamic memory allocation and deallocation functions of malloc, free, realloc, and calloc. These functions are confusing to write, so we have provided an in-depth guide below. **Please read through this entire pdf before beginning.** The specifics for each function are located in `malloc.c` as well as in the subsections below.

2.1 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.



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 as a function called `my_sbrk` located in `suites/malloc_suite.c`. **Make sure to use this call rather than a real call to sbrk, as doing this can potentially cause a lot of problems during program execution.** 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!

We've written my_sbrk to be able to only hand out a certain amount of memory before returning -1 to indicate that its done. This limit gives us the ability to test the behavior of the code when my_sbrk can't get more memory.

2.2 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 decent 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 that left over block of memory in a data structure where information about all such free blocks is kept. This is called the **freelist** and we will return to this 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 store this structure? We can't simply call `malloc()` to allocate space, since we're writing the function, and that'd lead to infinite recursion! However, there's an easier way that will keep our bookkeeping structure right with the data we're allocating for easy access.

In order to keep track of allocated blocks, we will create a structure to store the information we need to know about a block, also known as metadata, inside the block itself. The metadata contains two things: a pointer to the next node in the freelist, and the size of the user data section. Both of these are required in order to accurately keep track of the memory available in the freelist.

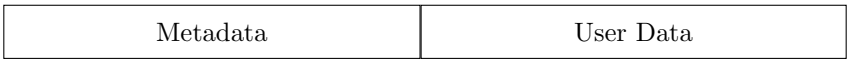


Figure 1. The metadata is placed before the user data. The next byte after the end of the metadata is the first byte that can be given to the user.

We will need to take into consideration the leading metadata whenever we allocate blocks. To let the user have as much space as they requested, when they request a block of size **n** bytes we will allocate a block of size `sizeof(the metadata) + n`. The size requested by the user will be stored in the metadata; additionally, the metadata will contain a pointer to the next metadata struct in the freelist. As depicted in my_malloc.h, this is the struct definition for the metadata:

```
typedef struct metadata {
    struct metadata *next_size;
    unsigned long size;
} metadata_t;
```

The size portion of the metadata struct contains the size that the user requested. In order to get the total size of the block, we add this size with `sizeof(metadata_t)`, which is the size in bytes of the metadata. For ease of reading, this size will be represented as `TMS` (total metadata size, defined as `TOTAL_METADATA_SIZE` in your homework files) in all of our block representation diagrams. The user does not care about the metadata for the block, they just want the size they requested. **Therefore, when you return a block to the user, you will need to use *pointer arithmetic* to 'step over' the metadata and return the address of the data.** What this looks like:

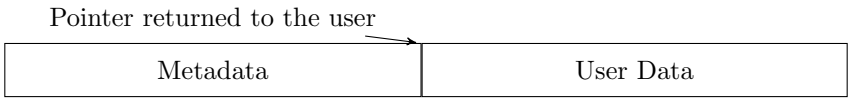


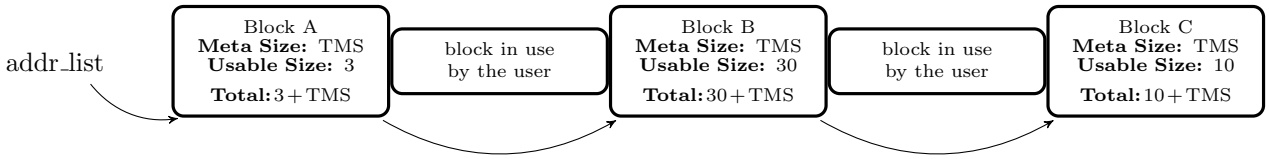
Figure 2. When a block is returned to the user, the pointer returned points to the beginning address of the area used by the user.

2.3 The Freelist

When we split up memory, we give one piece/block to the user. The remaining pieces/blocks are placed in a linked list, called the freelist, to be used at a later time. For this semester, we are representing our freelist as a single singly linked list that is organized by the address in ascending order. This linked list will be defined as a global file variable and to help you out, we have already defined it for you.

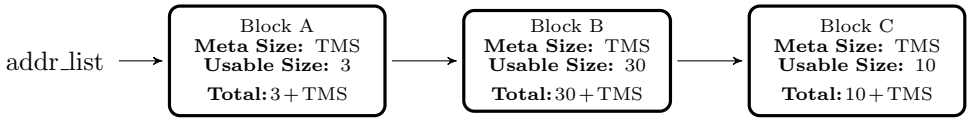
```
metadata_t *addr_list;
```

To help visualize this, below we have an example representation of our freelist.



Note: The name of the blocks refer to their address order. For example, since the letter B comes after the letter A, Block B starts at an address after Block A.

For the remainder of the pdf, we will represent the freelist without spaces for the blocks currently in use by the user like so:



A Quick Note: The node representations in our freelists should be read as the following:

1. First Line: The name of the block ("Block B") - Note that the name refers to the ordering that the blocks should be in.
2. Second Line: **Meta Size** → The size of the metadata for that block
3. Third Line: **Usable Size** → The size of the space available to the user
4. Fourth Line: **Total** → The total size of the memory taken up by this block

Since the addr_list is singly linked, be sure to properly update the next pointer when adding and removing nodes from the list.

2.4 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 takes up `sizeof (metadata_t) = 16` bytes for example, we need at least a

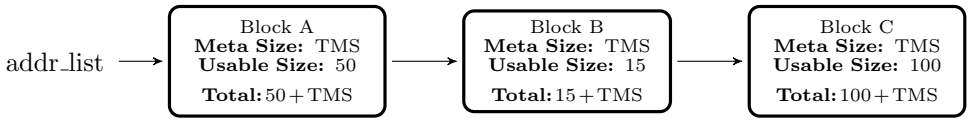
$$256 + 16 = 272$$

byte block.

Note that the size of your metadata will vary based on your computer's architecture and platform. Use `sizeof()` to avoid depending on the platform, specifically `sizeof(metadata_t)` when doing math with the metadata size.

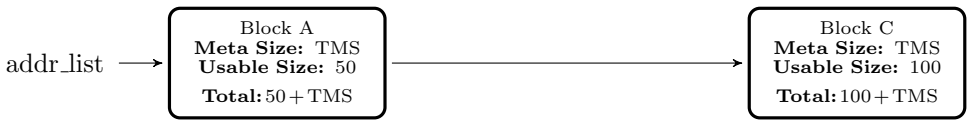
How do we get from one big free block of size 2048 bytes to the block of size 272 bytes we want to give to the user? In this simple implementation, you will traverse the `addr_list` to find the best block to satisfy the user's request, which should be equal or greater than the size requested, and "split" off however much you need from the front or the back. For this assignment, you must split off from the back.

Say we have the following situation:



When we malloc for a certain size, we first want to use a block of that exact or best size, remove it from both the `addr_list` and return it to the user.

Ex: `malloc(15)` would leave the freelist as so:

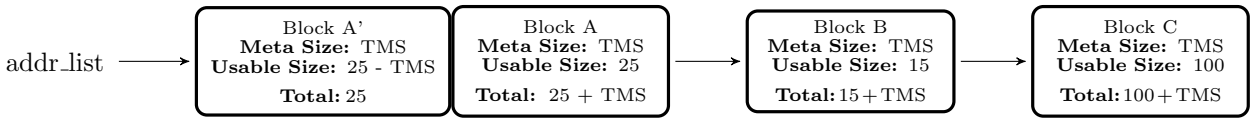


If we do not have a perfectly sized block, then we will return the smallest block in the list that has enough room to hold the user's requested data. There are actually two cases to consider:

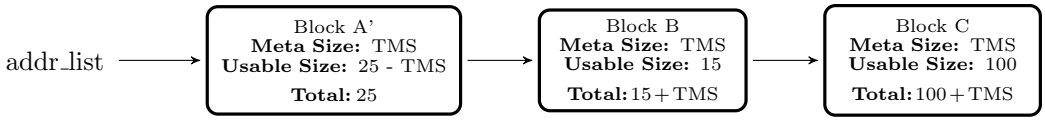
- If a block's user data size is at least `requested_size + MS + 1`, then you should split the block into two halves. The right split should become its own block, with its own metadata and `requested_size` bytes in the user data section. This is the block that will eventually be returned to the user. We need to update the left split to reflect its new user data size of `old_size - MS - requested_size`. Note that the user size is still at least 1 byte, making it a valid block. This block should remain in the freelist.
- If a block's user data size is smaller than `requested_size + MS + 1`, then splitting the block would

not leave enough room for the metadata and at least a byte of user data. So, instead of splitting, **return the whole block**, even though the user gets a bit more space then they need.

In the following diagram, the requested size is 25 bytes, and the first block has enough room:



Once Block A is returned to the user, this call will leave the freelist as such:



Don't forget to move the pointer to the beginning of the space the user uses at the end of the metadata before returning the block to the user.

2.5 Simple Linked List: Deallocating

When we deallocate memory, we simply return the block to the `addr_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 start of the metadata). Notice we don't clear out all the data. That simply 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.

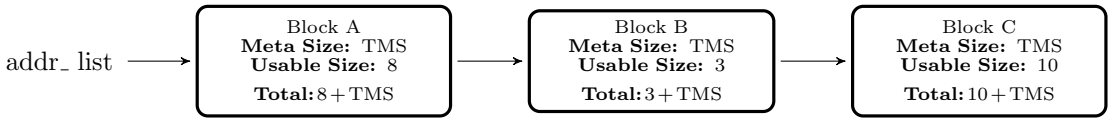
How do we know which blocks we can join together? If adding a free block's address and the block's total size (both TMS and the user data size) gives you the address of another free block's metadata, then you know that those two blocks are next to each other in memory, so they can be merged. If A has a size of M and B has a size of N, then if they are next to each other in memory, you can merge them to create a new block of size M+N+TMS.

Whenever you deallocate a block, you must merge it with nearby free blocks if possible. There are a couple of ways to do this:

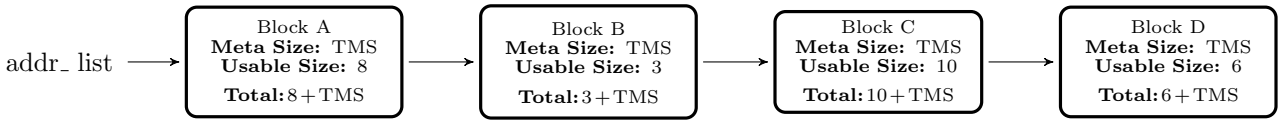
- A simple technique is to search the freelist for blocks that are next to the newly freed block. Conceptually, it looks like this:
  - Given a block B, iterate through the free list and find a block A that is located directly to the left of B in memory. This means that if we take the A's address and add A's total size, we line up perfectly with B's address, meaning there are no gaps between A and B. If this block exists, we can merge A/B, then remove A from the freelist.
  - We can do the same procedure for blocks that are located directly to the right of B in memory. Again, if a block C exists, we merge B/C and remove C from the freelist.
  - Finally, we re-insert B or our newly merged blocks back into the freelist. Don't forget to keep nodes sorted by address!
- We can optimize this process by simply adding the newly freed block to the freelist first, then scanning through the list for blocks to merge. Since the freelist is sorted by address, two blocks MUST be next to each other in the freelist in order to be next to each other in memory. This means you can go through every pair of consecutive elements in the freelist, check if the blocks are touching in memory,

and merge them if necessary. When merging contiguous blocks, it is recommended to leave the left block in the freelist and grow its size, while removing the right block.

Here's an example of how to free and merge blocks:



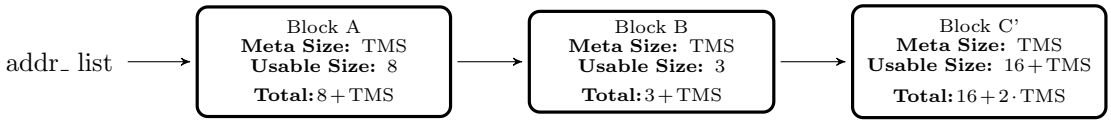
If we deallocated a block of size 6 (Block D), we would first iterate through the `addr_list` for the correct left and right addresses of the block and check to see if the block needs to be merged either to the right of left. In this example, the block to be entered is not directly next to any other blocks in memory, so we would just insert it into the `addr_list`. This would leave the freelist as seen below.



If Block C and D were right next to each other in memory (i.e. the address at end of block C is equal to the address at the beginning of block D), then we would need to perform a left merge. To perform this left merge, pop block C from the `addr_list` , add block D to it, reset the size, and add it to the `addr_list` in the same position that block C was in. Note that this series of steps assumes that you have not yet added D to the freelist. These steps are depicted below.

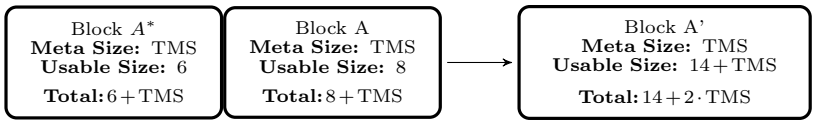
Remove Block C from the free list and merge it with Block D to make Block C'.

Add this new block C' back into the freelist in its proper position.

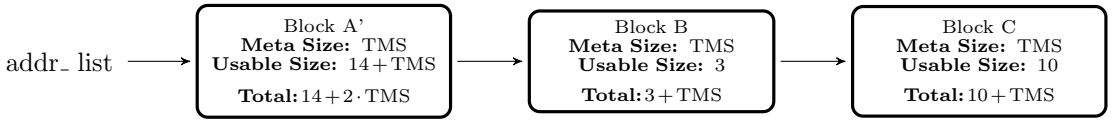


If instead of a Block D we had a Block A^* which was located right before Block A in memory (i.e. the address at the end of block A^* is equal to the address at the beginning of block A), then we would need to perform a right merge. To perform this right merge, pop block A from the `addr_list`, add block A^* to it, move block A's metadata to block A^*, reset the size, and put the block back where block A originally was in the `addr_list`.

Remove Block A from the free list and merge it with Block A^* to form Block A'.



Add this new block A' back into the freelist in its proper position.



Note: To compare pointers (addresses), cast them to uintptr_t first

2.6 Helper Methods

Implementing `malloc()` can seem like quite a daunting challenge, but your debugging process can be helped along tremendously if you do not write all of `malloc()` in one method and instead split it up into helper methods! Helper methods are incredibly useful for understanding what is going on and also results in cleaner code, so it's a win-win strategy. Below are some **required** helper methods to implement that will be tested in the autograder, **we advise that you use them for functions discussed in later sections (e.g. `my_malloc()`)**.

Two helper methods are already created for you:

- `metadata_t* find_left(metadata_t*)`
- `void remove_from_addr_list(metadata_t* remove_block)`

Function comments in the homework files will tell you what these methods do (along with the following methods you will be implementing).

The helper methods that you have to implement:

- `metadata_t* find_right(metadata_t*)`
  - Given a pointer to a free block, find a block in the free list that is exactly besides that block in memory, to its right. Note that we are NOT trying to find the next block in the linked list relative to the free block pointer, as that might not be directly next to this free block in memory. Additionally, note that the block directly right of the free block (in memory) may not be in the free list.
- `void merge(metadata_t* left, metadata_t* right)`
  - Given two free blocks that are next to each other in memory, merge them such that the left block now takes up the space of the two existing blocks, and that the right block is essentially removed from the free list.
- `metadata_t* split_block(metadata_t* block, size_t size)`
  - Given a pointer to a free block and a `size` argument, split the block into two such that the right portion has a usable size of `size` bytes (and thus a total block size of `TMS + size` bytes). The left portion of the split should be shrunk accordingly, while the pointer to the right portion is returned. Do not alter the free list.

- `void add_to_addr_list(metadata_t* add_block)`
  - Given a free block pointer, add it to the free list (which is essentially a linked list). Realize that the free list is sorted by the address of the block, so insert the given block accordingly.
- `metadata_t* find_best_fit(size_t size)`
  - Traverses through the linked list to return a free block pointer that best satisfies the best-fit conditions for allocation specified in section **2.4**. Note that you are only to return the pointer to the best-fit free block, DO NOT split this block.

Remember, this is not an exhaustive list of operations that can performed with helper methods. Feel free to implement additional helper methods for any aspect of malloc that works for you.

Note: Typically in production code, we would have these functions to be static because we want them to be private to my_malloc.c. However, in order for the autograder to run your helper functions outside of my_malloc.c, these functions will be declared normally in my_malloc.h and my_malloc.c.

2.7 my_malloc()

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

1. The size of the block we are looking for is the size that the user is requesting. (Note: if this size in bytes is over `SBRK_SIZE - TOTAL_METADATA_SIZE`, set `my_malloc_errno` to the error `SINGLE_REQUEST_TOO_LARGE` and return `NULL`
2. If the request size is 0, we do not have to allocate anything; mark `NO_ERROR` and return `NULL`).
3. Now that we have the size we care about, we need to iterate through our freelist to find a block that best fits. Best fit is defined as the first block that has the same user data size as the requested amount of bytes, or the smallest block that has more than enough space for the requested data.
   (a) If the best fit block is exactly the same size, you can simply remove it from the `addr_list` and return a pointer to the body of the block to the user.
   (b) If the best fit block is larger than the requested size, but is not big enough to split, then simply remove and return the block, as if it were a perfect fit. To tell if a block is too small to split, take the user data size of the original block and subtract the number of requested bytes. If the remaining number of bytes is not enough to contain a valid block of metadata size + 1 user byte (defined as `MIN_BLOCK_SIZE`), then it is too small to split.
   (c) If the block is big enough to house a new block, we need to split off the portion we will use from the right side of the block, keeping the remaining left side in the freelist. Remember: pointer arithmetic can be tricky, so **make sure you are casting to a `uint8_t *`** before adding the total size (measured in bytes) to find the split pointer! Otherwise, your calculations might get implicitly multiplied by `sizeof(metadata_t)`.

   If no suitable blocks are found at all (i.e., all blocks have a user data size smaller than the requested size), then call `my_sbrk()` with the size `SBRK_SIZE` to get more memory. Our autograder expects exactly this amount of size to be requested, so please make sure to use the macro.

   The chunk of memory returned by `my_sbrk()` should be treated as a new block, so you will need to write a metadata_t to the start of the block. This means that the block's available user size is actually `SBRK_SIZE - TMS`.

   Additionally, **you must merge this new block with any free blocks that are directly next to the new block in the freelist**. Since the freelist is sorted by address, there is only one block you

need to check: the last block in the freelist. However, you may find it easier to repurpose some of your code from `my_free()`; you can even call `my_free()` on the newly sbrk'd block in order to put it back, but you will need a working implementation of `my_free()` first.

After setting up the block's metadata and merging it if possible, restart the search for a best-fit block. In the event that `my_sbrk()` returns failure (by returning `-1`), you should set the error code `OUT_OF_MEMORY` and return `NULL`.

Remember that you want the address you return to be at the start of the block body, not the metadata. This is `sizeof (metadata_t)` bytes away from the very front of the block. 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 front of the metadata to get a pointer to the body. If you have not specifically set the error code during this operation, set the error code to `NO_ERROR` before returning.

4. 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.

2.8 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's user data section and not to the block's metadata.
2. Attempt to merge the block with blocks that are next to it if those blocks are free. See the previous subsection on deallocating for more information on how to do this.
3. Finally, place the resulting block in the `addr_list` by setting the respective next pointer in each node for the `addr_list`. Do not forget to keep the list sorted by address.

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

2.9 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 from the user, `void *ptr` and `size_t size`. It will attempt to effectively change the size of the memory block pointed to by `ptr` to `size` bytes, and return a pointer to the beginning of the new memory block.

Do **not** directly change the freelist or blocks in `my_realloc()` — leave that to `my_malloc()` and `my_free()`. This means you don't need to worry about shrinking or extending blocks in place[1]; if `size` is nonzero, just call `my_malloc()` to attempt to allocate a new block of the new size. Make sure to copy as much data as will fit in the new block from the old block to the new block (don't forget to eventually free the old pointer). The rest of the data in the new block (if any) should be uninitialized.

Your `my_realloc()` implementation must have the same features as the `realloc()` function in the standard library. Specifically:

1. If the pointer is null, call `my_malloc` using the `size` argument (i.e. `my_malloc(size)`)
2. If the size is equal to zero, and pointer is non-null, call `my_freefree` using the ptr argument and return null (i.e. `my_free(ptr)`)
3. Else, create a new block via `my_malloc` and and copy the old block's data to the new block up to min(new block data size, old block data size)

[1]Even though we don't extend or shrink blocks in place in this homework, keep in mind that real-world implementations (which are not written in a panic right before finals) very well could.

4. If the requested size is nonzero and `my_malloc` fails, then do not free the old pointer.

Hint: Look at the man page for the C function memcpy

2.10 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.

Hint: Look at the man page for the C function memset

2.11 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`.

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`.

2.12 Running the Autograder and Debugging

First, spin up the docker container environment using the `cs2110docker.sh` or `cs2110docker.bat` scripts and ensure you are in this homework directory.

To run the autograder's test suite, run:

```
# Run all test cases
$ make run-tests

# Run a specific test case
$ make run-tests TEST=Malloc_Perf_Block1
```

When you run the tests, you will see a pretty hefty output in your terminal. Each line of the output provides critical information depicting which tests you are failing/passing. The general format of:

```
suite_filename.c:420:Fun_Test_Case:test_description
```

states a test named `test_description` is failing/passing in an individual test case named `Fun_Test_Case`, located in that specific test suite `suite_filename.c` at line 420. That is, test suites contain test cases which contain tests. For example,

```
malloc_suite.c:37:Malloc_Perf_Block1:test_malloc_perf_block1_lists
```

tells us whether the address_list and size_list is correct when we malloc for a perfectly sized block. More information about the test is written in malloc_suite.c, and the assertion that failed is on line 37.

To debug an individual test case with gdb, run

```
$ make run-gdb TEST=Malloc_Split_Block_SBRKmerge
```

There will be no checks with valgrind for this homework since you are implementing malloc and friends yourself, not using them!

2.13 Deliverables

Submit the following files to the "Homework 10: Malloc Implementation" assignment on Gradescope:

- `my_malloc.c`

Do **NOT** modify or submit the header file, `my_malloc.h`. We will grade with the original copy. Any functions or variables you add should be marked `static` so they do not conflict with the grader.

Note that we reserve the right to change test case weighting or add additional test cases to the autograder after the assignment is due.

3 Frequently Asked Questions

1. **I have a segfault, what do I do?**

   The quickest way is to debug it yourself with gdb. Here is the link to our supplemental gdb video:

   - `https://www.youtube.com/watch?v=GMF2tpXVKqQ`

   Here are some other gdb tutorials:

   - `https://www.youtube.com/playlist?list=PLsK1fComPkFiYc4oX8Ef9QUyiWVM5BaKe` (playlist created by a previous TA)
   - `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; the autograder checks the state of the freelist and will fail if you have dummy nodes.

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

   No, it is static and is therefore already initialized to NULL by the compiler.

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

   Yes. We reserve the right to change our mind on this, but if you get a 100 on Gradescope, you should expect 100 on the homework. Just keep in mind that the tests may be weighted differently when grading than in the provided student autograder.

4 Rules and Regulations

1. Please read the assignment in its entirety before asking questions.
2. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
3. If you find any problems with the assignment, please report them to the TA team. Announcements will be posted if the assignment changes.
4. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency please reach out to your instructor and the head TAs **IN ADVANCE** of the due date with documentation (i.e. note from the dean, doctor's note, etc).
5. You are responsible for ensuring that what you turned in is what you meant to turn in. 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 Gradescope. Email submissions will not be accepted.
6. See the syllabus for information regarding late submissions; any penalties associated with unexcused late submissions are non-negotiable.

4.1 Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work. Homework assignments will be examined using cheat detection programs to find evidence of unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student. If you supply a 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 platform which would allow other parties to it (public repositories, pastebin, etc). If you would like to use version control, use a private repository on github.gatech.edu

Homework collaboration is limited to high-level 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.

High-level collaboration means that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code. What you shouldn't be doing, however, is pair programming where you collaborate with each other on a single instance of the code, or providing other students any part of your code.

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.

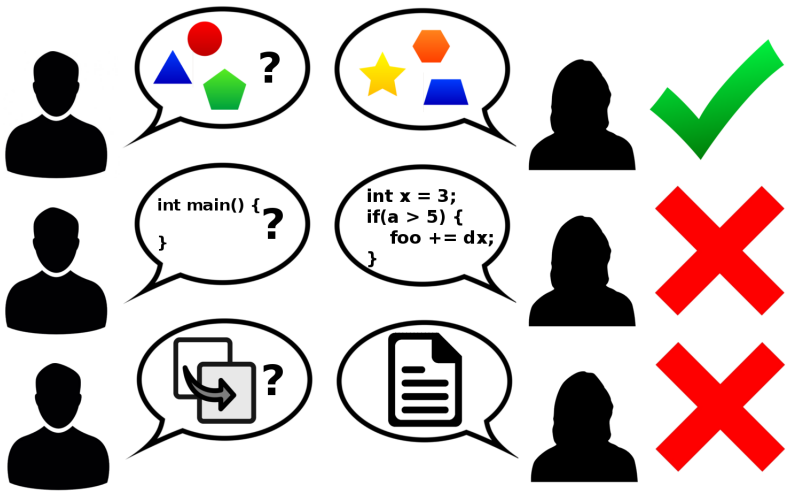


Figure 1: Collaboration rules, explained colorfully