Programming Assignment #8

General

For our final project, we will write our own little toy memory allocator. You will be implementing a simple version of `malloc()`, `free()`, `realloc()`, and all of the necessary helper functions to make these work. Details of the implementation are described and shown below.

You will be given a larger OOP codebase than in Project 7. Please take some time to read over all of the code and comments to understand the structure of the code base. Also, you will need to implement functions within multiple classes, so reading over will ensure you don't miss a TODO.
We have written a `Heap.cpp` class that will act as the sandbox in which you will manage memory. Notice the size is configurable and we will test your implementation with various sizes, so we recommend you do the same. This memory is configured with a `base_address` that we have abstracted for you and is accessible for you to intuitively use. For example, if our `base_address` is x2000, requesting a read for the address x2001 will produce the byte located at index "1" in the "Heap". The base address is guaranteed to be greater than 0 and a multiple of 4. This memory is **byte-addressable**, **little-endian**, and uses **32-bit addresses**. Make sure to review what these mean if you are unsure.

Simple Memory Allocation Scheme

In this project, you will use a simple header-and-footer-based scheme for memory allocation. Whenever there is a free block of memory, the first four bytes (one int) of the block will contain a **positive** integer that indicates how many free bytes of memory follow this 'header'. Likewise, the last four bytes of the block act as the 'footer' and contain the same number found at the top of the block. You will find out why this is important later. When a block of memory is allocated, we will use the same scheme except the header and footer will be **negative**. For example, as shown in figure 1, the block on the left contains 4 allocated bytes A, B, C, and D which are between a header and footer with the number '-4' to indicate this. The block on the right shows an example of a block of 32 free bytes. Notice that the meta-data also takes up memory; in this case, an allocated block of four bytes actually takes up 12 bytes in memory.

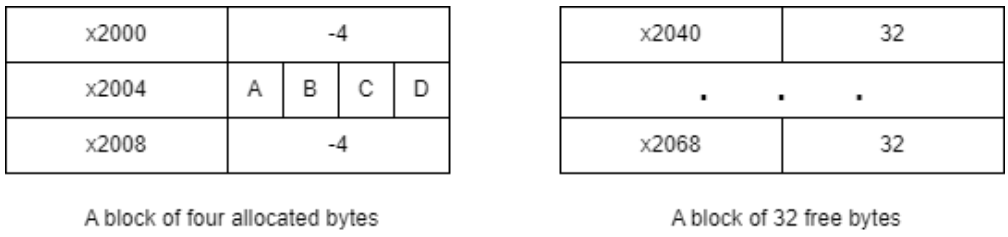

Figure 1: An example of memory layout

When the class constructor is called, in `MyMemoryController.cpp`, we will initialize the memory for you so the first and last block of memory both contain $total\_bytes - 8$. Refer to the diagrams above if you do not understand where the '-8' comes from.

For this lab, we will always guarantee the memory space is a multiple of 4 bytes and can be any size between 8 bytes and 2^{32} bytes since we are using `uint32_t` for addresses.

Your Job

In this section, we will describe the specifications of the project in the order we recommend implementing things. These specifications are also outlined in the code itself.

Helper Functions

There are a few helper functions in the code we have asked you to write. This includes `BaseMemoryController::read`, `BaseMemoryController::write`, `BaseMemoryController::word_to_bytes`, and `BaseMemoryController::read_full_word`. These functions will be necessary to complete the other functions. These should also help you become familiar with the interface. I recommend implementing these and then writing some code in `main.cpp` to observe and test their behavior.

Malloc

When the user calls malloc, you should allocate a block of memory of the given size and return the address to the first address the user can write to. Remember that this is not the same as the start of the allocated block, or else you will overwrite the header. You should allocate the **first** (lowest-address) available memory location that can fit the requested size. If there are no blocks of memory greater than or equal to the requested size, return 0 (even if the sum of the sizes of the free memory blocks is greater). You will also have to word-align the memory allocations, meaning all blocks of memory should start and end on a multiple of four even if the block is not. Fig 2 shows an example of malloc. The left-most diagram shows an empty memory layout starting at x2000. Notice there are 2040 free bytes available (x2800 - x2000 - 8). When we call `malloc(4)`, we allocate address x2000 since it is the first free memory. The right-most block shows two more allocations of which neither are a multiple of four bytes. Because of this, the five-byte request actually contains eight free bytes within the block. You will still need the metadata to hold the true value, but the size itself may be different, similar to the 37-byte block. For malloc(0), simply do nothing and return 0.

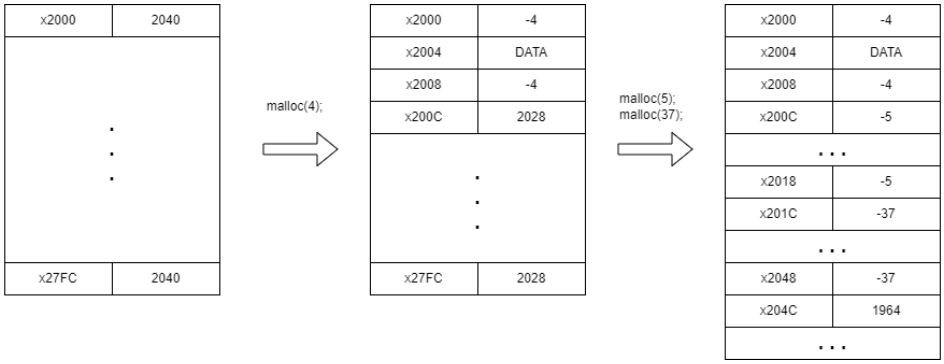

Figure 2: An example of malloc

Free

When the user calls free, you should free the allocated block of memory. We will assume that all frees are valid and any double frees or invalid pointer frees are user-error. However, you should **assert** that the address being free'd is within the memory space. For example, if the heap starts at x2000 and is 48 bytes long, any address less than x2000 or greater than x202F should crash on an assertion. The one exception to this is an address of 0 (nullptr). If a free is called on an address of 0, you should immediately return and do nothing.

Another caveat is you will need to combine adjacent free blocks. For example, if we free a block of size 4 and there is a free block of size 8 right below it, it should be merged into a free block of size 20 (Think about why this is 20 and not 12, where does the extra +8 come from?). This should work for blocks adjacent **above and below**, which is why you will find it useful that each block has a header and a footer. An example is shown below.

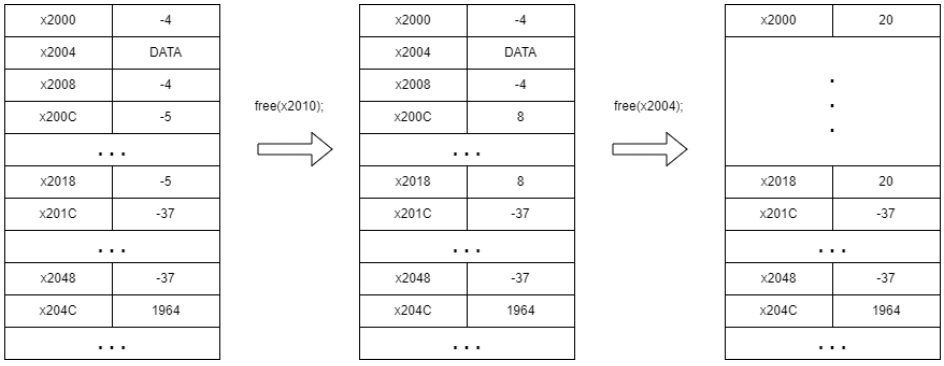

Figure 3: An example of free

In Fig 3, we first call `free(x2010)`, as it is the pointer to the first user-writable block. Looking at the address above, we can see there is a -5 indicating the block was 5 bytes. When we free it, we will overwrite this with +8, as that is how many bytes the block technically had. Then, when we call `free(x2004)`, we free the 4-byte block at the top and since it is adjacent to the free block below, they are merged into a free block of size 20.

Realloc

When the user calls realloc, you should reallocate the given block of memory to the new size. The specifications for realloc are as follows

- If the pointer is 0 (nullptr), realloc should behave like malloc.
- If the new size is 0, realloc should behave like free.
- If the new size is smaller than the original size, you should shrink the current block and keep the same pointer.
- If the new size is larger than the original size and there is enough space **below** the given block, you should expand the block and keep the same pointer.
- If the new size is larger than the original size but there is not enough space to expand the given block, you should malloc a new block, copy the data over, free the old block, and return the new pointer.
- If there is no block in the heap large enough to realloc, free the original block and return 0.
- You do not need to check if the pointer being reallocated is valid, a bad pointer is considered a user error. However, if the pointer is outside of the heap's memory space (less than the base address (but not 0), or greater than the base address plus the total size), **assert** a failure, similar to free.

An example is shown in Fig 4.

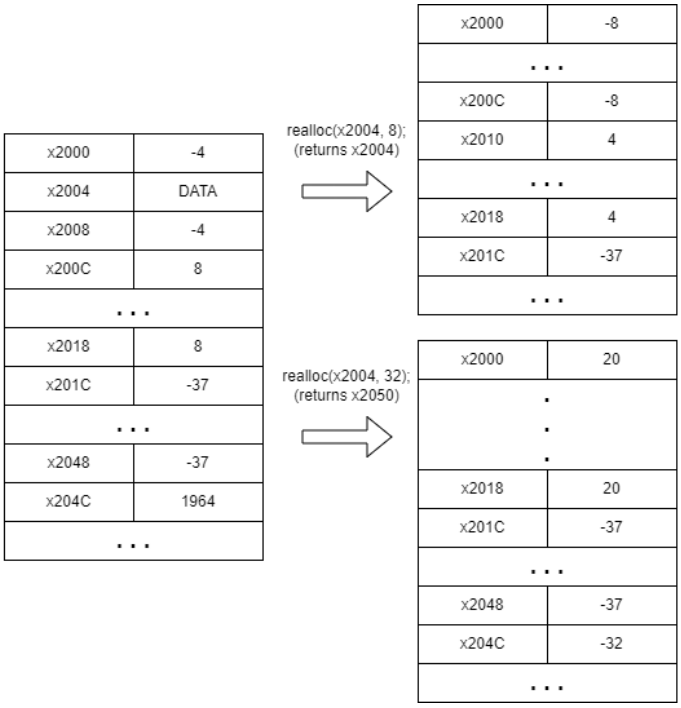

Figure 4: An example of realloc

Testing and Extra Credit

Testing

For this lab, we have given you a **very** basic test case in `main.cpp`. It is up to you to use the skills you have learned throughout the semester to think of ways to test your code.

Extra Credit

As an incentive for this, we will be offering extra credit for test case submission as follows:

1. Create a test case in the main of `test_case.cpp` that instantiates a `MyMemoryController` object.
2. You are allowed to configure to any size and perform and combination of `read`, `write`, `malloc`, `free`, and `realloc` and use `assert` to check pass/fail as long as it abides by the constraints of the assignment.
3. Submit this along with your lab to qualify for extra credit.

Additionally, your code must be able to pass your own test case!

We will award up to 10 points of extra credit (at the TA's discretion) based on how good the test case is. Keep in mind a bad or invalid test case will earn zero points of extra credit. Here is a rough scale on how points will be awarded. Please note we will **not** accept requests to regrade extra credit.

- **0 points** The test case does not work, you do not pass it, it is too simple, or it is too common.
- **3 points** The test case shows an understanding of the lab and its requirements but lacks the thoughtfulness required to break other submissions.

- **7 points** The test case shows a thorough understanding of the lab and its requirements and shows thoughtful stressing of possible implementation errors.
- **10 points** The test case thoughtfully uses the items implemented in this lab to thoroughly stress test submissions in a logical computer-science-oriented approach.

Please note that your test case will **not** affect other student's grades and their test cases will **not** affect your grade.

What To Submit

You are required to submit `BaseMemoryController.cpp`, `MyMemoryController.cpp`, and (optionally) `test_case.cpp` to GradeScope by the deadline. Please note your test case should be submitted at the same time as the code, not in a separate submission.

CHECKLIST – Did you remember to:

☐ Re-read the requirements after you finish your program to ensure that you meet all of them?

☐ Make sure to thoroughly test your program and make your own test cases?

☐ Upload your solution (all files) to Gradescope?