Here is our own implementation of the malloc() and free() library calls. In our project we write a program to dynamically allocate up to 5000 bytes of memory for the user.

In our mymalloc(), we must first make sure the user is not trying to allocate 0 bytes, and also make sure that there is enough space remaining. We then move on to our two main cases: an empty block, or a block with some memory already allocated. In the first case, we simply allocate the proper amount of memory at the beginning of the block, and return the block's data. In the second case, when we have gotten to a gap in allocated memory, we must split what is there if there is enough space. If there is not enough space, NULL is returned. Otherwise, we run splitfrom(), which returns 1 if successful, and then return the block's data. We also use a meta struct to store metadata for each piece of allocated memory. This metadata is 4 bytes large and must be accounted for each time we malloc.

In our free(), we must first make sure the user is not trying to free a pointer that wasn't allocated, and that they're not trying to free a pointer that is already freed. The next step is to check the given pointer's adjacent blocks because if they are empty then we can consolidate them. We achieve this using the flags from our meta struct. Once we have the necessary information, we can move on with the actual deallocation. Here we have four major cases: if the pointer to be freed has an empty block in front, behind, both, or neither. In the first three cases, we consolidate the memory in order to accurately represent the new status of the block, and assure our pointers are in the right spots. In the final case, we simply set the flag to unused, and the memory becomes available to the user again.

EXTRAS:
We wrote a method printmem() to display the status, metadata, and size of all pointers in the block of memory. This proved to be extremely useful for debugging our code, as it allowed us to easily understand the current state of our memory block.

Our splitfrom() method splits an empty block of memory into the memory that is allocated, and leaves the rest as empty. It first checks if there is enough space for the memory, and if not, returns 0. If there is enough space, the space needed is updated appropriately, and a used flag is left at 0 at the end of the newly allocated block. The memory is then marked as used and 1 is returned.

WHAT WE LEARNED:
In this project, we learned that memory management is a very difficult, precise task with a lot to consider and keep track of. It is also extremely important to be efficient, both with space and time, such as creating a meta struct of no more than four bytes. Each of our test cases operate in a matter of microseconds, which we found to be plenty efficient.

Here are the average times (other than outliers) we received for our test cases (ms = microseconds) when run on Matthew's computer:
Test Case A: 4300ms - 4400ms
Test Case B: 31ms - 33ms
Test Case C: 95ms - 115ms
Test Case D: 100ms - 130ms
Test Case E: 12ms - 14ms
Test Case F: 205ms - 220ms