Students:
Daniel Fraser 174005456
Peter Laskai 157002815

Our metadata struct:

metaStruct: it holds the size of the memory block (excludes the size of the struct itself also uses only 13 bits since we don't need more than the size of the memory block itself), an int to tell us if the block is free or not (uses only 2 bits), and a pointer to the next metadata (if any). Currently uses 8 bytes of memory.

Our Functions inside myMalloc.c:

myMalloc(): we first determine if the first meta block has been initialized. If the first metadata block is already created then we go through each metadata block and determine if it's big enough to hold the requested bytes. We call **splitBlocks**() to determine if the block is the right size or if we can split the block (returns NULL if it can't). If we couldn't find a spot for a the bytes requested, we return NULL with a message saying we were unable to give the user the requested amount of bytes

splitBlocks(): simply takes in the current metadata (cursor from **myMalloc**()) and number of bytes from **myMalloc**() and determines if its size is bigger than the number of bytes requested. If there is enough memory to fit another metadata between the current metadata and the one it points to(even if the metadata will have size 0, since it can be merged later), it shrinks down the size of the metadata, then creates a new free metadata where the current metadata points to and the new metadata points to what the current metadata previously pointed to.

---

myFree(): it first checks if the pointer taken in is a pointer that we created, (a pointer that lies between the beginning and end of the memory block). If it is a pointer we can free, we first check if the pointer is already free. Otherwise, we simply find its metadata and tell the program this memory is free so it can be used by another pointer later on. We then call **mergeBlocks**() so we can combine it with another free metadata if possible.

mergeBlocks(): starts at the first metadata, then tries to combine as many free consecutive metadata blocks together (takes into account the size of each metadata as well). If the cursor isn't free, it will simply skip until it either hits the end or the next free metadata.

Findings:

- With the size of our metadata and using 1-byte pointers we can only have a maximum of 555 1-byte pointers inside our memory block at any time ((8+1)555 = 4995).
- Easy to segment fault if not careful when adding to pointers (caused the most logic errors for us)
- Not freeing one pointer in a test case can really have a snowball effect on other test cases beside the test case itself

Workload:

- Each case keeps track of its execution time and returns it
- Each case is inside its own function to make it easier to execute
- Each case also has its own variable of total execution time that will be averaged when finished
- Each case is surrounded by dashed lines (ex: test case X starts--------... and test case X ends-------...) so we know where each output is from
-

Overall average execution time: 11 milliseconds
(*times may vary but will be around these numbers)

Test Case A: (average execution time: 7 milliseconds)

- Designed to break the program by overloading are memory with 1000 1-byte pointers
- Then freeing all 1000 after they have all been created

Test Case B (average execution time: .1 milliseconds)

- Designed to create a 1-byte pointer then immediately free it
- Tests to make sure we can reuse memory that has been freed

Test Case C (average execution time: .6 milliseconds)

- Not designed to break the program
- Designed to check if we can keep track of our memory and what we malloc()ed and free()d
- Choose to randomly free or malloc() 1-byte pointers
- Create a maximum of 1000 1-byte pointers
- Free the 1000 1-byte pointers
- if out of memory ignore mallocs, and free instead

Test Case D (average execution time: .5 milliseconds)

- Same as test case C but are pointers are can be between 1 and 64 bytes
- Also checks if there is memory for new pointers

Test Case E & F (our custom test cases)

- See testcases.txt for information
- **Case E average execution time: 2.8 milliseconds**
- **Case F average execution time: .2 milliseconds**