Assignment One: ++Malloc

0. This assignment is an implementation of pseudo-malloc and free functions to simulate how the two functions operate in the C language. There are 3 files: mymalloc.c, mymalloc.h, and memgrind.c. The mymalloc.c is the code for the actual malloc and free functions, the mymalloc.h file is a header containing functions, includes, and global variables that were used in the c files, and the memgrind.c file contains test cases that use the pseudo-malloc and free.

1. mymalloc.c: This file contains four functions: mymalloc(), myfree(), allocatenode(), and cleanHeap().
    a. mymalloc(size_t size, const char *file, int line): the inputs are the given size a user wants to allocate from the heap, the name of the file, and the line number of the program. First, the function checks to see if the user is trying to input 0 or negative space from the heap and we return NULL if that's the case. Next, the function checks if the space is too large for the heap and returns NULL if that's true. Then, it checks to see if the user is calling malloc for the first time and inputs a struct into the heap with size BLOCKSIZE. Finally, the function iterates through the available blocks in memory and allocates enough space for the pointer if there is enough available.

    b. allocateNode(): This function takes a given node and splits it in two, one node to store memory for the users pointer and the other for the remainder of memory. First, we create an offset for node2. Next, we initialize the size of node2 and the new node and we make the new node in use and node2 becomes available. Last, we set the pointers of the two nodes to the correct location.

    c. myfree():This function finds the given pointer and sets the node it's stored in to 0 so we know it's not in use or it gives an error because the pointer has already been freed. First, we check to see if the heap has been malloc previously and we check to see if the given pointer is NULL, in both cases we return and send an error. Next, we run the cleanheap() function. Finally, we iterate through the array and find the given pointer, if its found we check if its been freed or not, if it hasn't we set the inUse variable to 0.

D. cleanHeap(): This function finds two nodes and merges them together to make more space for the next pointers. First, we set the variables for the head, current, and next nodes. Then, we iterate through the array and if we find 2 consecutive nodes that are not in use, we merge them then seat the current to the head and search again until we have no more consecutive unused nodes.

2. mymalloc.h: This file is the header file, it contains functions we use throughout the program, defines mymalloc and myfree, and creates the struct for the nodes we store in the heap.

3. memgrind.c: This file is used to test our mymalloc with six different cases and see if it correctly runs. Outputs the average time for each case.
    a. functionA(): We do a basic test to see if we can allocate a single block of memory and free it 150 times. Our mymalloc correctly functions in this case.
    b. functionB(): We malloc 150 bytes in an array and free 50 every time we malloc 50 bytes. Our malloc correctly functions in this case.
    c. functionC(): We malloc one byte 50 times and free it until we reach 50 mallocs, then we free the rest in another while loop.
    d. functionD(): We malloc a random amount of memory 50 times and free it until we reach 50 mallocs, then we free the rest in another while loop.
    e. The other two functions E and F are described in detail in the test plan.

4. Metadata Implementation: For our metadata we included three variables: A char to determine if the node is in use or not, an unsigned short int to determine the size of the data a user wants to malloc, and a pointer for the next node. The “#pragma pack(1)” that is above the struct makes it so there is no extra padding in the struct so the total amount if space is 1 + 2 + 8 = 11 bytes.