Our program contains our implementation of malloc() and free(), as well as error checks for those functions.

In mymalloc.c, we have a static char array, smallBlock, of size 1000, and a static char array, bigBlock, of size 4000. smallBlock is used when malloc() requests of size 100 or lower are made. bigBlock is used when malloc() requests of size 101 or greater are made. This is to prevent fragmentation, where many small blocks are allocated and freed in a way that leaves only small blocks available for allocation. We also have a static void* myMemList, of size 5000. This keeps track of malloc() address locations.

initializeList() goes through the entire myMemList array, setting all of the values in each index to NULL.

assignPtrSpace() takes a void* ptr as an argument, and goes through all of myMemList array, checking if the value at each index is NULL. If it is, then the value at that index is set to ptr.

isAddrValid() takes a void* ptr as an argument, and goes through myMemList array, checking if the value at each index is ptr. If it is, then then the value at that index is set to NULL, and 1 is returned.

bigmalloc() allocates space for something in the bigBlock array, if there is space available. If there is not, the program prints to stderr, giving the line that caused it and the file that it occurred in.

smallmalloc() allocates space for something in the smallBlock array, if there is space available. If there is not, the program prints to stderr, giving the line that caused it and the file that it occurred in.

mymalloc() takes an unsigned int size, a char * file, and an int line as arguments. If the list has not been initialized, initializeList() is called, and initList is set equal to 1. If size > 100, bigmalloc() is returned. Else, smallmalloc() is returned.
myfree() takes a void *p, a char * file, and an int line as arguments. If p is NULL, the program prints to stderr; a NULL pointer cannot be freed. The line that caused it and the file that it occurred in are also printed. If the address of p is not valid, the program prints to stderr; the address passed was not returned from malloc() or has already been freed. The line that caused it and the file that it occurred in are also printed. Our program deals with invalid addresses by checking the address given to our free function and comparing it against the myMemList array which holds all addresses that malloc has returned. Otherwise, the chunk of memory is freed.

There is one case where a solution could not be thought of. When a user calls free(p+10), if the address that results from this addition happens to be equal to the address of a pointer that was malloced later on in code, free will run. The problem is that our free cannot see the context in which we received this address, if we are able to see that the user performed addition on a pointer, we may be able to mitgate this problem, but as of right now, we would not be able to catch this
problem.

myfree() takes a void *p, a char * file, and an int line as arguments. If p is NULL, the program prints to stderr; a NULL pointer cannot be freed. The line that caused it and the file that it occurred in are also printed. If the address of p is not valid, the program prints to stderr; the address passed was not returned from malloc() or has already been

freed. The line that caused it and the file that it occurred in are also printed. Otherwise, the chunk of memory is freed.

chkMallocSpace() goes through each index of myMemList, checking if the values at each index are not NULL. If they are not, the program prints to stderr; not all malloced space was freed. This function is called by atexit() in main.c. This is so that myMemList is checked after the program exits.