Function Explanations:
For this assignment, the functionality of the malloc() and free() library calls for dynamic memory allocation were examined, and with this knowledge of how bethod functions worked, we created our own versions. Through the use of a header file (mymalloc.h) and these redesigned mymalloc() and myfree() functions (mymalloc.c), the average running time for different levels of workloads were calculated in order to establish and review their efficiency. These specific details are explained below.

mymalloc.c
The purpose of this file was to recreate our own malloc and free functions, mymalloc and myfree respectively, in hopes to optimize the previous existing library functions, as well as preventing users from creating illegal arguments. This was done in the following way:

mymalloc - First, the function checks if the memory function is empty, and if this is the case, it is initialized by being passed to the initialize function. Then, the size of the memory requested is checked to make sure it is acceptable through the validateinput function, if the size is not between 0 and 4998, it will tell users "Invalid Request, Cannot Allocate". Next, the checkSpace function is called which searches the array through a loop and returns TRUE if a chunk of free memory large enough to accomodate the request is found. After that, the freeSpace function is called which is essentially the same as checkSpace except it returns a pointer to the start of the acquired chunk of memory. If findSpace returns FALSE, the defrag function is called and findSpace is attempted again. The defrag function loops through the memory array and searches for chunks of free memory, combining them until a chunk of adequate size is found.

myfree - In the myfree function, the memory array is looped through from the start of the chunk of memory being freed, to either the end of the chunk or the end of the memory array itself. Once one of these conditions is met, each byte of used memory is freed and the function returns TRUE. Otherwise, the users will be told "Error: invalid address, cannot be freed", and the function will return FALSE.

memgrind.c
In this file, as previously described, six different functions, grindA, grindB, grindC, grindD, grindE, and grindF were created for the purpose of examining the average efficiency of our own malloc and free functions. These worked in the following ways:

grindA - Declaring a variable char* ptr, a for loop was used to allocate 1 byte of memory to ptr, and then immediately free it 150 times.

grindB - Using the character array char* myblock[5000], a for loop was used to allocate a total of 150 bytes of memory and store the pointer in the array with myblock[j] one at a time, and then a second loop was created to free the 150 pointers in the array also one at a time.

grindC - For this function, using the character array char* myblock[5000] again, the rand() function was implemented to randomly choose between allocating or freeing one byte of memory. If rand()%2 returned a 1, memory would be allocated to the array, and the variable index would increase, along with the variable timesMalloced to keep track of position and amount of times malloced was called. However, if rand()%2 returned a 0, if there is at least one space of memory allocated it would be freed, then increase the freeindex variable, and decrease the number variable to keep track of how much space is possible to be freed. After mallocing 150 times, if there were more spaces to be freed, a while loop was used to free every space one at a time until number equalled 0.

grindD - Following the previous function, this one worked in the same manner with the only difference being instead of allocating 1 byte of memory at a time, an integer variable stored a random variable between 1 and 64 which was used to determine the amount of allocated bytes.
grindE - Making this workload up on our own, this function mimicked the functionality of grindA, but instead of allocating 1 byte of memory and freeing it 150 times, we decided to push it to exactly the limit by allocating a single byte of memory and immediately freeing it 4998 times.

grindF - Similarly, the concept for this function came from grindD, but this time, instead of allocating between 1 and 64 bytes, it was decided to choose and smaller range being 1 to 5 bytes, randomly mallocing and freeing the allocated memory.

Overarchingly, the commonality that these functions shared were the array float time[100], struct timeval start, end, gettimeofday(&start, NULL), gettimeofday(&end, NULL), and the float variable avg. Running through each grind function 100 times, each time they restarted, the start time was saved before running through the workload, then the end time was saved, and the difference between the two times was stored in the position time[i]. This array of 100 individual times was then passed to the function average which of course calculated the average running time, and then each function returned the efficiency.

Results:
grindA(): ~2 microseconds
grindB(): ~201 microseconds
grindC(): ~21 microseconds
grindD(): ~21 microseconds
grindE(): ~82 microseconds
grindF(): ~20 microseconds

In order to prevent runtime errors and users from inputting improper commands, the following messages are printed to define the problem:
Freeing pointers that were not allocated by malloc: “Error: invalid address, cannot be freed”
Redundant freeing of the same pointer: “Error: invalid address, cannot be freed”
Saturation of Dynamic Memory: “Invalid request, cannot allocate”