CS214 Systems Programming Assignment 1: ++Malloc

Dev Patel (**drp145**) and Parth Patel (**ptp26**)

→ What is the purpose of the assignment?

To implement our own version of the system's malloc and free that provides the same basic functionality and also detects common programming and usage errors.

→Personal Algorithm for the Assignment.

The first thing our program does is declare a global static array of size 4096 (size can be changed as per user requirement). After the declaration we check if the array has already been initialized or not. If it is not initialized then we store the magic number of size two bytes at the very beginning and store a meta data next to it with size of two bytes and value (-4092), Here "-4092" means that bytes are available/free. Throughout the program, any positive size at start of block in the metadata column will be treated as if the block of memory is getting used. When the user frees some block, we will convert the block size to negative value, to show that the block is now not being used but has just specified size. Between calls, we will try to maintain adjacent free blocks as a single block by merging them. A void pointer pointing to the starting of free block (right after where the metadata ends) will be returned to the user. **All the files of the assignment have been commented thoroughly to help the reader walk through what's going on.**

→ Working of Mymalloc:

We have declared a static array with the capacity of maximum 4096 bytes to store all the memory needed for the purpose of this assignment. From this 4096 bytes we use 2 bytes for magic number and 2 bytes for metadata (size of block) to keep track of the memory in static array. **Therefore we have 4092 bytes available at first call. The structure of static array is as shown below,**

Magic - Number	Meta-Data	4092 free bytes available at first malloc call

Our malloc function during the first call checks if there is any garbage value present in metadata column of the static array which is used. If that garbage value is not equal to our decided magic number (a random number of 16 bits, we used "57921" in our code), then

our magic number is stored in the 2 bytes allocated for it at the start of array and initialized the remaining array to 0 for further use. After first allocation the metadata column is updated according to the length of free or allocated memory. **The example is shown below when we allocate 400 bytes at first call.**

Magic Number	400	400 bytes are allocated	-3690	3690 bytes are free
Size = 2 bytes	Size = 2 bytes	Size = 400 bytes	Size = 2 bytes	Size = 3960 bytes

In the above case a pointer/address is returned for the function to store values (the pointer/address of the starting of 400 block). Similarly, for each successive call after first call, our "mymalloc" function allocates new blocks in array and returns a pointer/address. In the picture below are all the helping functions that we made to make our mymalloc function perfectly as per the requirement and report errors when needed.

```
static char STORAGE_ARR[MAX_SIZE];

short getBlockSize(void *blockStart) {

void putBlockSize(void *blockStart, short size) {

void *getFirstBlockAddress() {

void checkInitialization() {

int isBlockAvailable(void *blockStart) {

void *nextBlockAddr(void *blockStart) {

void *getFirstFreeBlock(unsigned int minSize) {
```

→ Working of Myfree:

The first thing Myfree does is to check if the array has been initialized or not (compare magic number) before freeing anything. If the array is not initialized it means that this is the first call and return error. Next thing it does check if the address that block begins with is within array or not and return error if it not. If the address is within array then it will compare the address to each block address and see if the address is block address or not to avoid error such as free(*ptr+2). After all these checks are done it will

call another function to check if the block that is to be freed has already been freed or not by checking the block size (if negative then free else if positive then allocated). If the block size is negative (already freed before) then it returns error but if it is positive then it will call another function to free the block (change the value to negative). This function also checks whether two adjacent blocks are free or not. **If two adjacent blocks are free then this function we merge them:**

...	-500	500 free bytes	-650	650 free bytes	....

...	-1150	1150 free bytes	.	.	..

Various functions used in my free:

```
int isPointerInRange(void *ptr) {

int isPointerAllocatedByCode(void *ptr) {

void freeMyBlock(void *blockStart) {
```