Kunal Pednekar, Sanjana Dodley
Asst2: Malloc

CODE IMPLEMENTATION:
This code implements malloc using a nonlinear approach that allocates memory from the left if user requested space is less than or equal to 30 bytes, and from the right otherwise. This is done to maximize space. More information on this is shown under "Algorithm/Logic Overview".

ENUM, STRUCTS and GLOBAL VARIABLES:

- **Enum boolean -** true or false are given values zero and one.
- **struct memEntry* head [global variable]-** global head pointer for the Linked List of memEntry structs
- **struct char memblock [global variable]** - memory array of characters.
- **struct memEntry -** a metadata struct that holds information about the data that it points to (right after it in the memory array) and serves as a linked list node with next and previous pointers as well as an isFree boolean variable that indicates whether the data it points to is free to be allocated.

FUNCTIONS:

- **void * mymalloc (unsigned int size, char * file, unsigned int line)** - Is called when user wants to allocates space. Depending on the user requested size, parameters will either be passed to *addFromLeft* (if size < 30 bytes) or *addFromRight* (is size >= 30 bytes). This is the function that is called in order to allocate memory in test case files. Returns a void * on success.

  Can return null if:
  - User requested for 0 bytes
  - User requested for data such that sizeof(<user requested space>)+sizeof(struct memEntry) > available size (or max capacity).

- **void myfree(void * data, char * file, unsigned int line)** - Free does a bunch of things, as mentioned below
  - Check to see if this was allocated in the range of our static char array
  - Check to see if it data points to beginning of allocated space (data - sizeof(struct memEntry)) should not give errors
  - Check is already freed in the past (prevent double freeing)
  - Check to see if data points to NULL

  Assuming that all of the void pointer passes all these checks, we traverse the linked list of memEntry structs to find a match. ***Overall takes O(n) time***

- **void tooMuchSpace(struct memEntry * ptr, unsigned int size, struct memEntry * head)** - This function checks to see if there is enough space after a block of memory to add another memEntry such that a user can put some data (at least 1 byte of data) there too, thereby trying to address the issue of fragmentation of memory/trying to use up as much memory as we can.

- **struct memEntry * findLargest(unsigned int size, struct memEntry * head)** - traverses the memory Array from right to left to find the leftmost chunk that has size greater than 30 bytes. Used in *addFromRight*.
    - The result of is NULL if there are no structs allocated from the right (in which case the data would have to be allocated at the very end of the array) or if the array is already full.
    - If the result is not NULL, then the method first checks if there is room after (in this case before since traversal is from right to left) to see if there is enough space to allocate memory.
- **struct memEntry * findMemSmall(unsigned int size, struct memEntry * head)**- traverses the memory Array from left to right to find the right most *memEntry* struct whose size is less than or equal to 30 bytes. Used in *addFromLeft*. ***Overall takes O(n) time in the worst case***.
    - The result is NULL if there are no structs allocated from the left (in which case the data would have to be allocated at the head of the array) or if the array is already full.
    - If the result is not NULL, then the method first checks if there is room after to see if there is enough space to allocate memory.

- **struct memEntry * addFromLeft(char * memblock, unsigned int size)** - This method first calls *findMemSmallest* to retrieve a pointer to the rightmost memEntry struct whose size parameter is < 30 bytes. Depending on the result of *findMemSmallest*, the method will check the memory following the address of the pointer, and see if there is enough space to allocate memory there.

- **struct memEntry * addFromRight(char * memblock, unsigned int size) -** This method first calls *findLargest* to retrieve a pointer to the leftmost memEntry struct whose size parameter is > 30 bytes. Depending on the result of *findLargest*, the method will check the memory following the address of the pointer, and see if there is enough space to allocate memory there.

- **struct memEntry* findAnyMem(char* memblock, unsigned int size)** - Because even after addFromRight or addFromLeft there can be space left for a user requested input, this method goes through the linked list linearly once through to see if there is free space enough for allocation. **O(n) time**

- **struct memEntry * initialize(char * memblock, unsigned int size)** - initializes default values to a memEntry struct.

- **void* mycalloc(unsigned int size, char* file, unsigned int line)** - mallocs the user requested input and then uses memset to initialize all values to 0.
- **void mergeBlocks(struct memEntry * ptr)** - upon free, merges with ptr->next if it is free, and increases the size of ptr by the size of the next block and the size of the memEntry (no longer is able to refer to ptr->next). After this, merges with ptr->prev, if it is free, and increases the

size of ptr->prev by the size of the next block and the size of the memEntry (no longer is able to refer to ptr).

ALGORITHM/LOGIC OVERVIEW

We have used a linked list data structure to complete the sorting for this assignment.

1. **MALLOC** - There are two ways that an allocation can go:
   For user data requested <= 30 bytes
   Main > mymalloc > initalize > addFromLeft > findMemSmall/create new memEntry > return pointer to user
   For user data requested > 30 bytes
   Main > mymalloc > initialize > addFromRight > findMemBig/create new memEntry > return pointer to user
2. **FREE** - Once all the checks are done (as described in the definition of the function above), it traverses the linked list of memEntries and finds a match. If found, deletes it.
3. **CALLOC** - Takes arguments, nitems and sizenitems which corresponds to the definition according to man pages. Essentially works the same as malloc but before returning a pointer to data, uses memset to initialize every byte to 0.

TEST CASES AND OUTPUT FORMAT w/ EXAMPLES:

- Possible Errors that are caught with this code:
  - **Freeing Null pointers** - "User tried to free a NULL pointer at line 9 in file test3.c"
  - **Double Free** - "User tried an illegal free at line 10 in file test3.c"

- Will indicate **whether the memory is allocated** or will print out an error message otherwise indicating the line number of the .c file in which it was written. - "address of data: 0x602b87\n we have successfully freed memEntry at 0x602b87"
- Will indicate **whether a malloc was successful** by showing the address in memory where it was allocated, as well as how much space is left in bytes. - "there is 4858 space \n malloced new pointer at address 0x6023a7"
- Will inform the user if a reassigned block was so large that it was split into two blocks, one of them being big enough to hold the data they are requesting and the other one initialized as free (able to hold at least 1 byte of data for user)