Aditya Patel, Yogesh Patel

myMalloc() design:

- The function starts off checking the requested size – namely making sure its not less than 1.

```
if(requestedSize < 1){
  return NULL;
}
```

- Next it checks to see if the block is uninitialized and if it is, it initializes it with a metadata block that is not in use and sets the size to the memory size minus the size of metadata.

```
if(myblock[0] == '\0'){
  metadata* temp = (metadata*) &myblock[0];
  temp->usage = notInUse;
  temp->size = BLOCKSIZE - sizeof(metadata);
}
```

- Next it will loop through all the blocks in the memory it traverses metadata by starting in the beginning and looking at sizes to get to the next block. According to the prompt we have a first fit algorithm so the first block it finds it will be placed into. And if nothing is found it returns null. If a block is found it returns a pointer to the start of the data portion of that block.

```
//Traverse forward find the first block that can fit the data and put it in.
int i =0;
metadata* blockPtr = (metadata*) &myblock[i];
for(i = 0; i<BLOCKSIZE; i += blockPtr->size + sizeof(metadata)){
  blockPtr = (metadata*) &myblock[i];
  //Checking if the block can accomodate requestedSize
  if(blockPtr->usage == notInUse && blockPtr->size >= requestedSize){
    blockPtr->usage = inUse;
    //if there isnt enough space to allow for metadata and 1 byte atleast it will
    //just return the whole block otherwise it will split it into 2. The block to give
    //and the block to set as notInUse for further use.
    if(blockPtr->size - requestedSize > sizeof(metadata)){
      metadata* nextPtr = (metadata*) &myblock[i + requestedSize + sizeof(metadata)];
      nextPtr->usage = notInUse;
      nextPtr->size = blockPtr->size - requestedSize - sizeof(metadata);
      blockPtr->size = requestedSize;
    }
    return &myblock[i + sizeof(*blockPtr)];
  }
}
//if nothing is found returns null
return NULL;
```

Aditya Patel, Yogesh Patel

myFree() Design:

- The function starts off by checking if the pointer passed to it is indeed a pointer to a chunk of memory by first separating the pointer into its respective metadata and data portions. It prints an error message then returns 0.

```
int myFree(void* ptr,char * file, int line){
  metadata* blockPtr = ptr - sizeof(metadata);
  char* dataPtr = (char*) ptr;
  //checks if it is a valid pointer to free
  if(blockPtr->usage != inUse){
    printf("Error: freeing non allocated memory %p from file: %s on line %d has made an error\n",ptr,file,line);
    return 0;
  }
```

- Otherwise the function then sets that ptr to not in use and locates the block previous to the one to free and the block after the one to free. This is done in order to combine them later if they are not in use. To find prev we originally traversed backward byte by byte from the ptr given and assumed each byte to be metadata in order to find the actual metadata. In reality this did not work when we filled the blocks with random data in our test cases so we changed it to only traverse forward.

```
//set pointer to not in use
blockPtr->usage = notInUse;

//find prevPtr
metadata* prevPtr = NULL;
if(blockPtr != (metadata*) &myblock[0]){
  int i;
  for(i = 0; i<BLOCKSIZE; i += prevPtr->size + sizeof(metadata)){
    prevPtr = (metadata*) &myblock[i];
    char* nextPtr = (char*) prevPtr + prevPtr->size + sizeof(metadata);
    if((metadata*) nextPtr == blockPtr){
      break;
    }
  }
}
//find nextPtr
metadata* nextPtr =(metadata*) (dataPtr + blockPtr->size);
```

- Once the next and prev are found both of there ussage are checked. If they are not in use they will be combined with the original block to free in order to provide more space and have less metadata and garbage that is not needed and also to ensure more data will fit.

```
//if nextPtr isnt in use it will combine them into 1 block.
if(nextPtr != NULL && nextPtr->usage == notInUse){
  blockPtr->size = blockPtr->size + nextPtr->size + sizeof(metadata);
}
//if nextPtr isnt in use it will combine them into 1 block.
if(prevPtr != NULL && prevPtr->usage == notInUse){
  prevPtr->size = blockPtr->size + prevPtr->size + sizeof(metadata);
  blockPtr = prevPtr;
  dataPtr = (char*) prevPtr + sizeof(metadata);
}
```

- Finally the free function will erase all data in the data portion of the block and return 1 for success

```
//Erasing all memory in the dataportion of the block.
int j = 0;
for(j = 0; j < blockPtr->size; j++){
  dataPtr[j] = '\0';
}
return 1;
```

Aditya Patel, Yogesh Patel

Workload Findings and Extras:

```
Average Workload Times in milliseconds:
        A: 1.349000
        B: 0.080000
        C: 0.120000
        D: 0.149000
        E: 0.108000
        F: 0.034000
```

In our testing we found that A took the longest time to complete. This was rather surprising as we figured smaller free and smaller mallocs should be quicker but in reality after thinking about it, it make sense those were the slowest because each malloc would take about the same time regardless of how many bytes you are mallocing and the same with free. To get to our predictions :

Firstly, we calculated our myMalloc function to be at the worst case O(n) because at the worst case it has to look through the full block size each time.

Then we calculated our myFree function to also be in the worst case O(n) because it has to look through all the blocks to find the previous pointer and it will be the second to last one each time.

With that in mind we predicted A, B to be about the same, C and D to be about the same and F be faster than E. A, B should be about the same because they are functionally doing the same thing (150 mallocs and 150 Frees). C and D are all mallocing 50 times total and freeing 50 times

total so they should be about the same time. And F should be faster than E since E is only mallocing in 1-64 byte chunks it will take more mallocs to fill to capacity in E while F can malloc 1 – 4092 bytes thus it can fill to capacity in less mallocs than E even with F filling its mallocs with random data.

Unfortunately, that was also not the case. Everything matched up with what we were expecting to see except for B since B seemed to be significantly faster than A. The only thing we can assume is that the compiler is able to figure out what we are doing and optimize it very well.