Dynamic Memory Management Code Explanation

Note: I do not claim ownership of this code; however, please be aware that my explanation of the code may contain mistakes or inaccuracies. Therefore, if you identify any errors or have any corrections, please let me know.

mymalloc.h Header File

https://github.com/SW33TSTUFF/Dynamic-Memory-Management-Simulation/blob/main/mymalloc.h

#ifndef MYMALLOC_H and #define MYMALLOC_H: These lines are known as include guards. They prevent the header file from being included multiple times in the same compilation unit, ensuring that the declarations are only included once.

These include guards are not necessary and you will still be fine without it.

It also includes the function declaration.

```
void *MyMalloc(size_t size);
void MyFree(void *ptr);
```

As you might have noticed by now, MyMalloc function expects datatype size_t as a parameter. So what is size_t in the first place? The size_t data type in C is an unsigned integer type used to represent the size of objects in bytes.
Watch this video to understand the uses of size_t over int.
https://www.youtube.com/watch?v=nBJuP_un20M

The header file also contains a struct called heapchunk, which is used as the metadata block for dynamic memory allocation. This struct stores information about the starting address, size of the allocated block, and the status of the block, indicating whether it is free or not.

mymalloc.c File

```
#include <stdio.h>
#include "mymalloc.h"

#define ARRAYMAX 25000
#define METADATAMAX 2500        // metadata size wasnt specified in the question. Took 2500

char heap[ARRAYMAX] = {0};      // mentioned array of 25000 bytes

size_t heapSize = 0;

const size_t metaDataStart = 22500;
heapchunk *metadata = (heapchunk *)(heap+metaDataStart);        // type cast becuase heap was initially initalized as char
size_t metaDataEnd = 0;
```

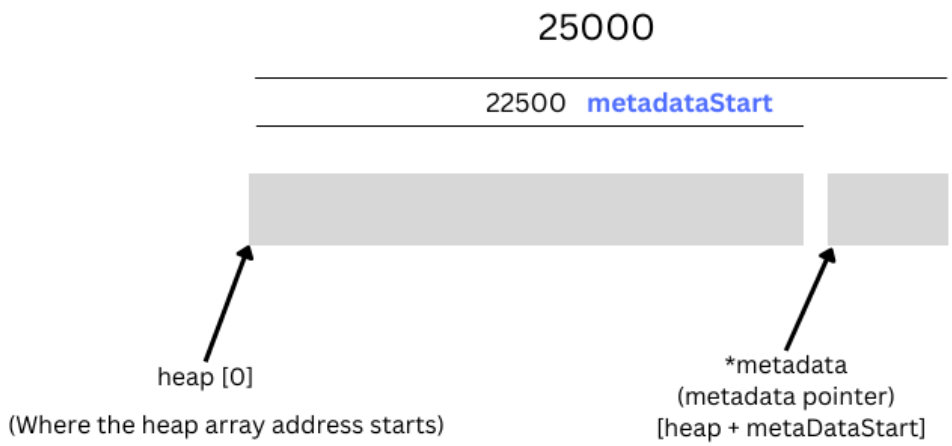
https://github.com/SW33TSTUFF/Dynamic-Memory-Management-Simulation/blob/main/mymalloc.h

The assignment requires us to create a global array of 25000 bytes. To achieve this, we will use a char array since the size of the char data type is 1 byte.

heapSize is a variable that keeps track of the amount of heap currently in use. This variable is useful for determining whether we have exceeded the heap limit.

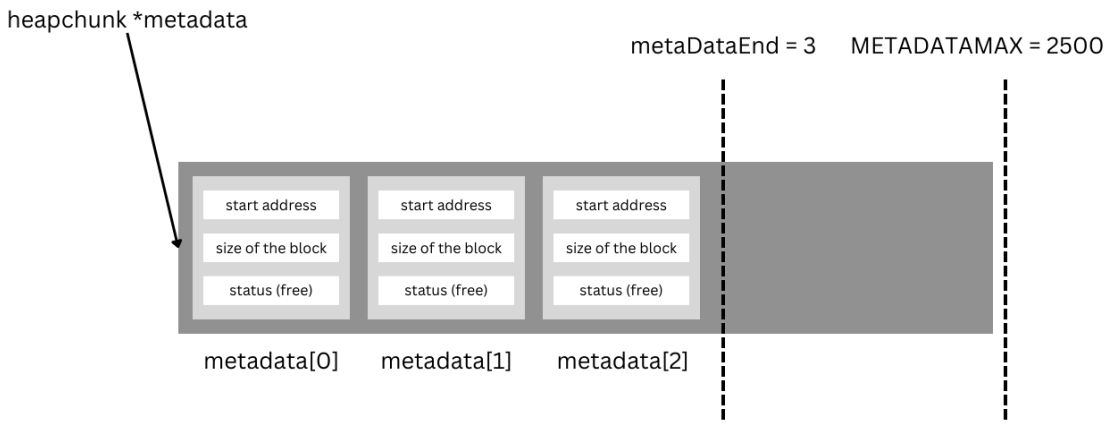
metaDataStart is a constant that represents the starting point of the metadata. It is set to 22500 because we have already allocated METADATAMAX as 2500.

To visualize this, the structure can be represented roughly as follows:




```
heapchunk *metadata = (heapchunk *)(heap+metaDataStart);
```

(heapchunk *) is just typecasting. We do this because heap was initialized as char and metaDataStart as unsigned int (size_t). When traversing through the metadata part, we want it to behave as follows.



MyMalloc() Function

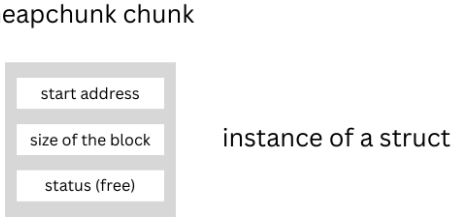
```
void *MyMalloc(size_t size){

    for(size_t i=0; i<metaDataEnd; i++) {

        if ((metadata+i)->status == false) {    // not taken

            if ((metadata+i)->size == size) {   // requested size is the same size as the current block
                // Allocate memory using current block
                (metadata+i)->size = size;
                (metadata+i)->status = true;
```

As you may have noticed, the for loop does not execute because metaDataEnd is initially set to 0. Therefore, we start from this point.

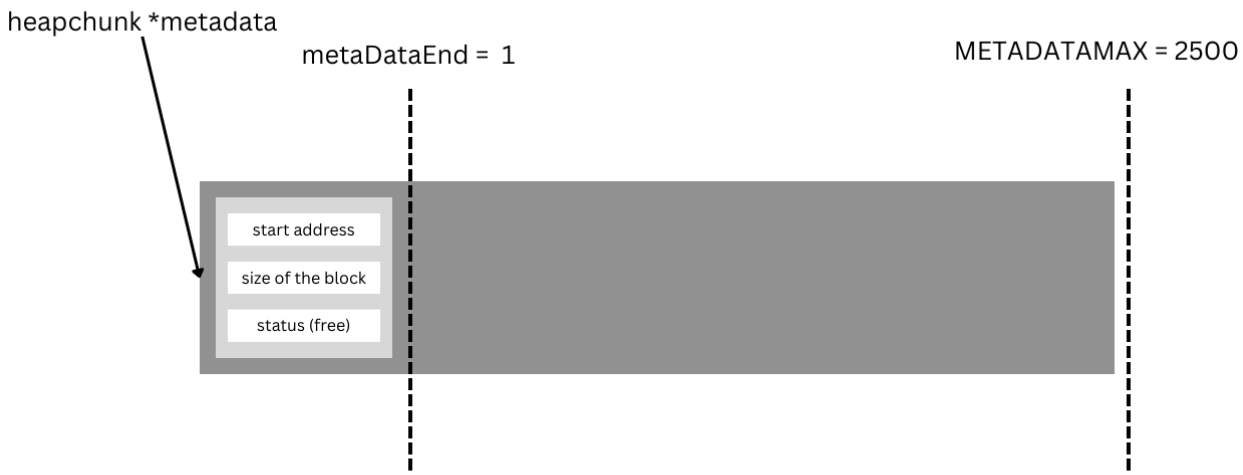
```
// Suppose the first 2 two options isnt available we simple allocate memory after the last allocated memry slot
if(heapSize + size <= ARRAYMAX-METADATAMAX)
{
    void *newAddress = heap + heapSize;
    heapSize = heapSize + size;

    heapchunk chunk;
    chunk.startAddress = newAddress;
    chunk.size = size;
    chunk.status = true;

    *(metadata+metaDataEnd) = chunk;        // simply enlisting that chunk
    metaDataEnd++;

    // printf("Memory allocated in next available space\n");
    return newAddress;
}
else {
    // printf("Memory not allocated\n");
    return NULL;
}
```

Here, we check if the requested size can still be allocated. If it can, we store the address in the heap. In the first run, when heapSize is set to 0, the return address will be heap[0] itself. However, we also need to store metadata.

To achieve this, we create an instance of the heapchunk struct, which serves as a metadata object.



Now we add this information to our metadata array and update the metaDataEnd accordingly.
.



The MyMalloc() function returns the new address if the allocation is successful. Otherwise, it returns NULL.

Now that metaDataEnd is no longer 0, we can proceed with the first part of the MyMalloc() function.

```
for(size_t i=0; i<metaDataEnd; i++) {
```

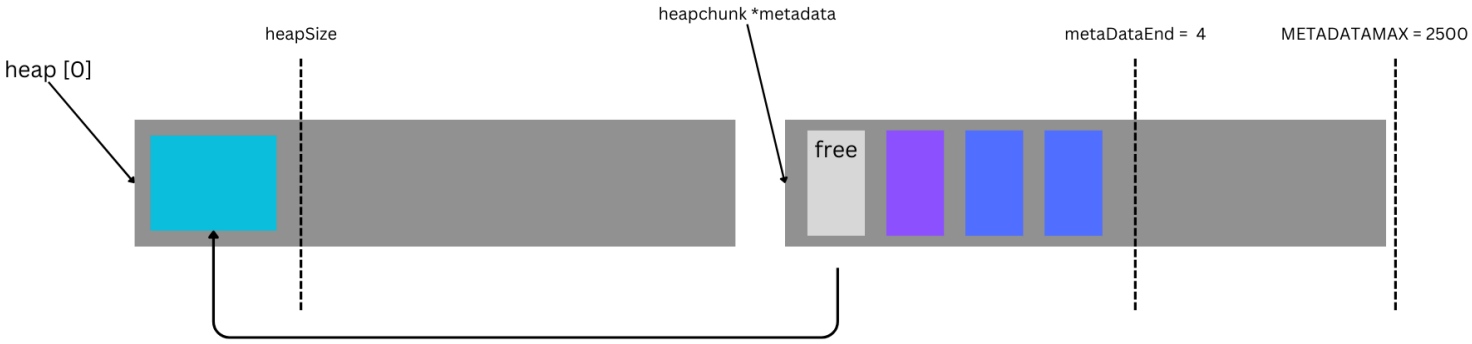
We go through the metadata chunks from the start to metDataEnd in search of free blocks. If this condition is not met in the for loop, we allocate after the last block like in the previous case.

While going through this loop if we find a block which match the exact size as the requested, we can easily modify the status to true (block taken) and return the address it starts.
Note: (metadata+i)->size = size; is not needed.

```
if ((metadata+i)->status == false) {    // not taken

    if ((metadata+i)->size == size) {   // requested size is the same size as the current block
        // Allocate memory using current block
        (metadata+i)->size = size;
        (metadata+i)->status = true;

        // printf("Memory allocated in a free slot which was the same size as the given size\n");
        return (metadata+i)->startAddress;

    }
```

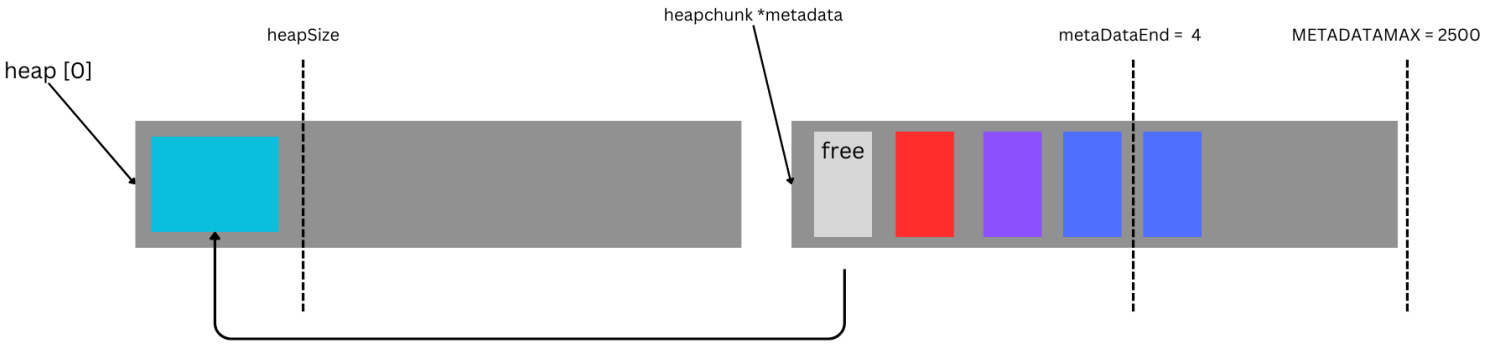
Let's say that when we traverse through this metadata chunks we find a size greater than our requested size? We can use it but this will later cause us issues like fragmentation. Where memory will be available but cannot be used due to scattering of memory.

So what we do here is that we shift the blocks to create 2 references of the splitted memory. Where one metadata chunk will hold the information regarding the requested memory and the other reference will hold the memory which is left after the requested memory.



Lets say that during our first iteration we found a free block but the requested size is less than the size labeled in the metadata chunk. The shifting of metadata chunks is initially done.

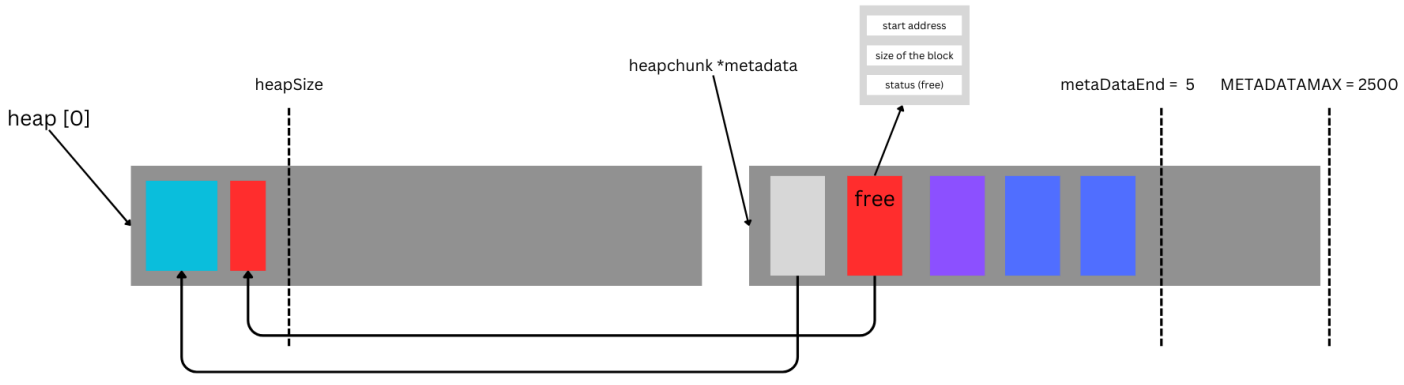
```
else if ((metadata+i)->size > size){    // Requested size is smaller than the current block
    // Create new slot for next free space and insert
    for (size_t j = metaDataEnd; j > i+1; j--){
        *(metadata+j) = *(metadata+j-1);
    }
```



```
size_t updatedSize = (metadata+i)->size - size; // new size by getting the metadata size - allocated block size
void *updatedLocation = (metadata+i)->startAddress + size;
(metadata+i)->size = size;
(metadata+i)->status = true;

heapchunk chunk; // create an instance of that chunk struct
chunk.startAddress = updatedLocation;
chunk.size = updatedSize;
chunk.status = false;


*(metadata+i+1) = chunk;
metaDataEnd++;

// printf("Memory allocated in a free slot\n");
return (metadata+i)->startAddress;
```

When traversing through the metadata chunks, if we find a size greater than the requested size, we can split the block into two references. One reference will hold the information regarding the requested memory, including its starting address and size. The other reference will hold the remaining memory after the requested memory.

By creating these two references, we can efficiently utilize the available memory without causing fragmentation. The new chunk holding the remaining memory is linked to the location of the block before shifting. This helps maintain the integrity of the memory allocation.



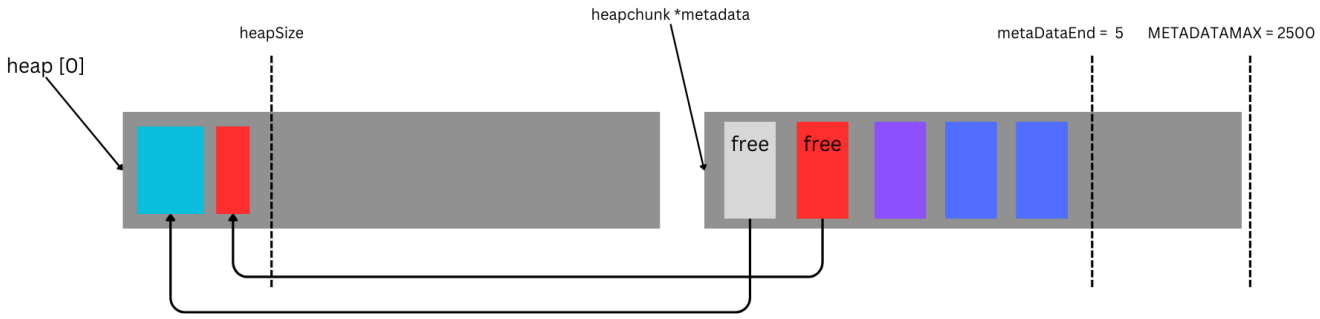
MyFree() Function

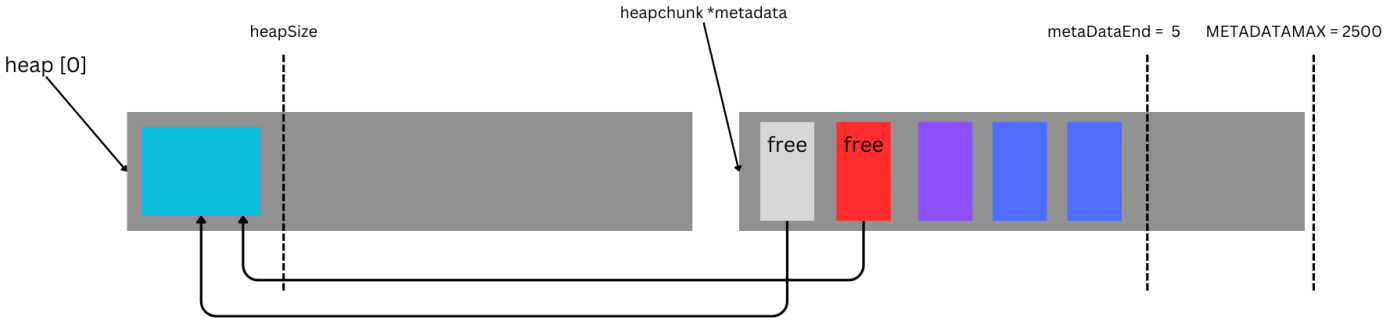
The first part of the code changes the free status.
The second part of the code handles merging of two consecutive free blocks.

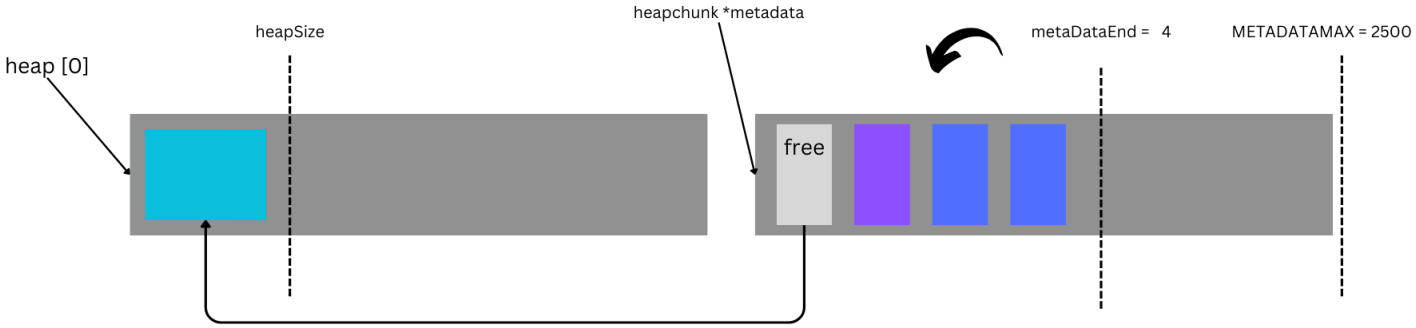
```
for (size_t i = 0; i < metaDataEnd-1; i++) {
    if ((metadata+i)->status == false) {

        heapchunk *preptr = metadata+i;
        heapchunk *ptr = (metadata+i) + 1;
```

If two consecutive metadata chunks are free, we only need one single chunk to hold the reference. So we shift the blocks by 1 and decrease the metaDataEnd by 1.

```
heapchunk *preptr = metadata+i;
heapchunk *ptr = (metadata+i) + 1;

if ((ptr)->status == false) {

    preptr->size = preptr->size + (ptr)->size;
    // After merging the chunks, it shifts the remaining chunks in the metadata array to fill the gap left by the merged chunks
    for(size_t j=0;j < metaDataEnd-2; j++){ // shifting by 1 becuase we now only need 1 metadata for the merge
        *(preptr+j)=*(preptr+(j+1));
    }
    metaDataEnd = metaDataEnd -1;
}
```




heap [0]
heapSize
heapchunk *metadata
metaDataEnd = 5
METADATAMAX = 2500
free
free


heap [0]
heapSize
heapchunk *metadata
metaDataEnd = 4
METADATAMAX = 2500
free