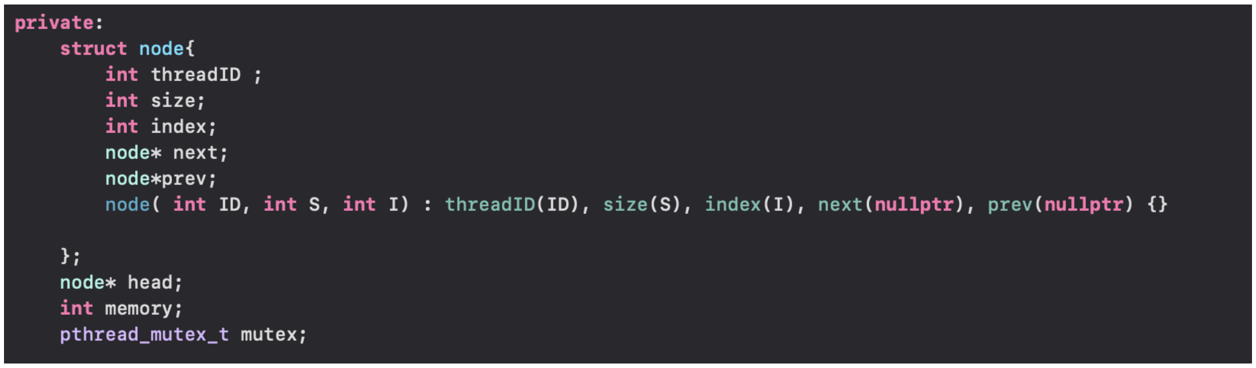
Mert Coşkuner 29120

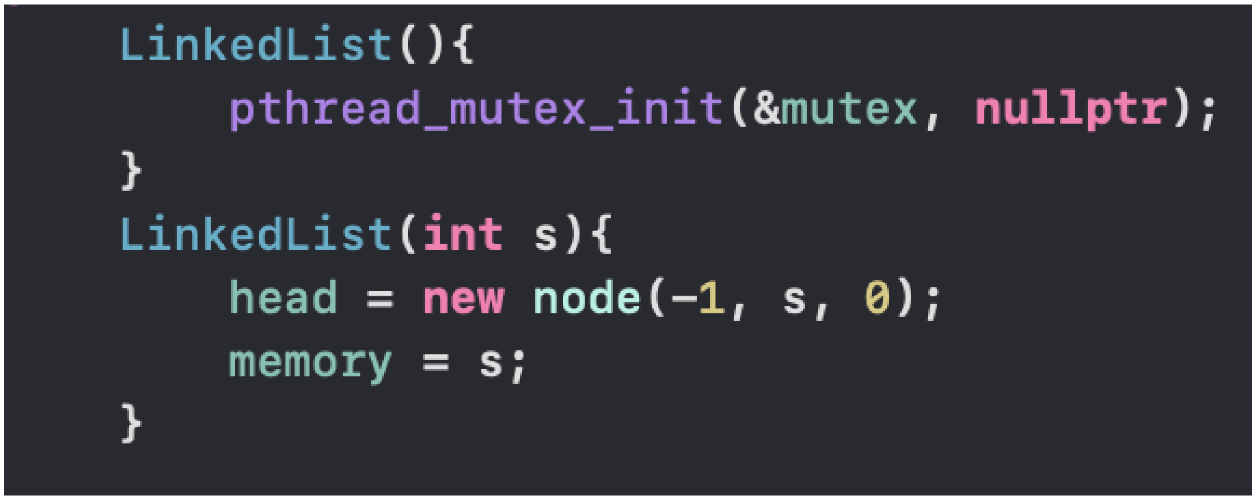
In this assignment, I used two classes to do my implementation. The first one is for my linked list structure and the second one is for managing the heap.

In a linked list node I implemented the following:
ThreadID, size, index, next and prev pointers.



```
private:
    struct node{
        int threadID ;
        int size;
        int index;
        node* next;
        node*prev;
        node( int ID, int S, int I) : threadID(ID), size(S), index(I), next(nullptr), prev(nullptr) {}

    };
    node* head;
    int memory;
    pthread_mutex_t mutex;
```

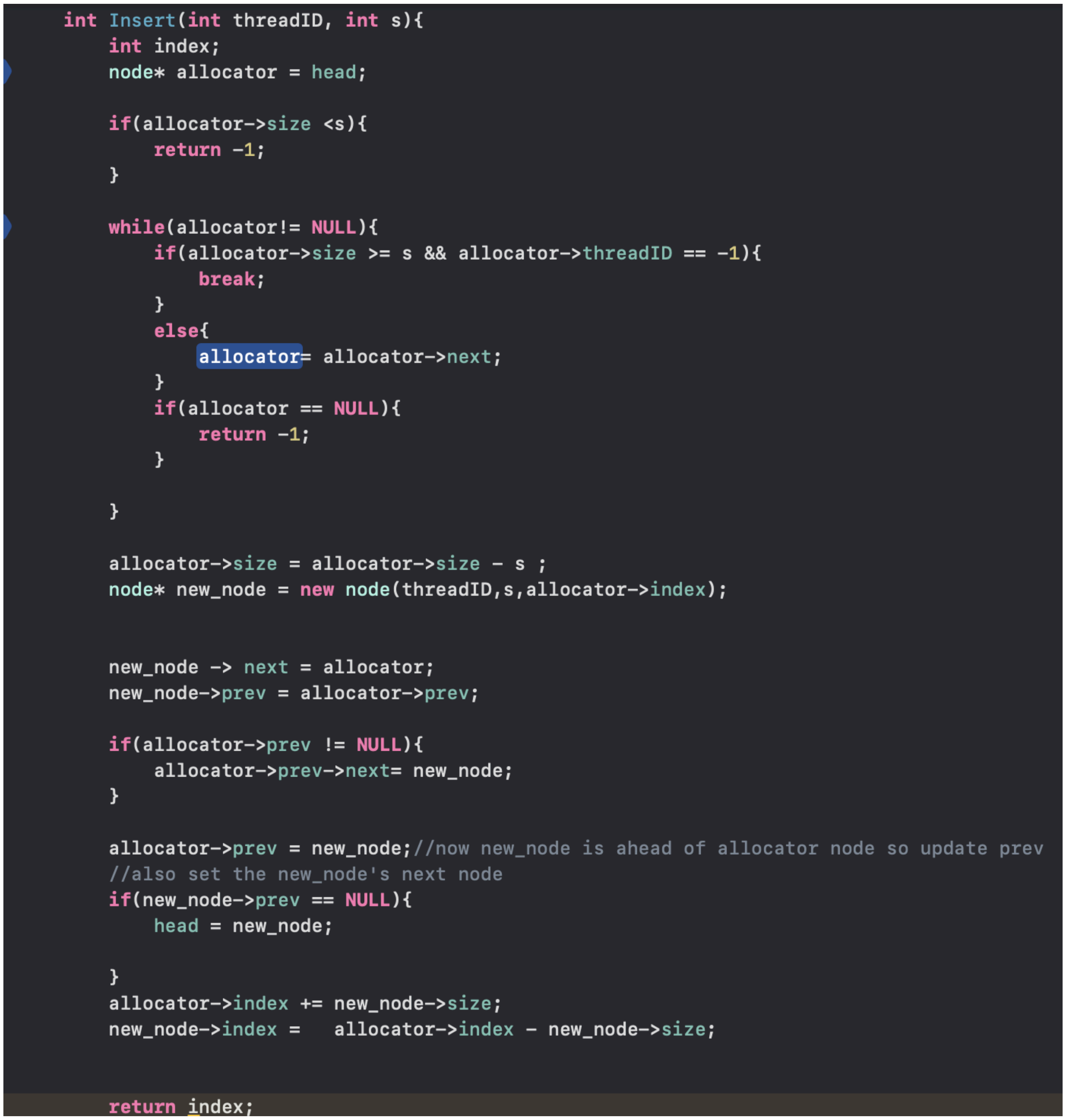
I used two types of constructors for specific purposes when we initialize the linked with the default constructor it starts with a mutex and if the constructor with a size variable initializes the linked list we start the simulation of the heap.



```
    LinkedList(){
        pthread_mutex_init(&mutex, nullptr);
    }
    LinkedList(int s){
        head = new node(-1, s, 0);
        memory = s;
    }
```

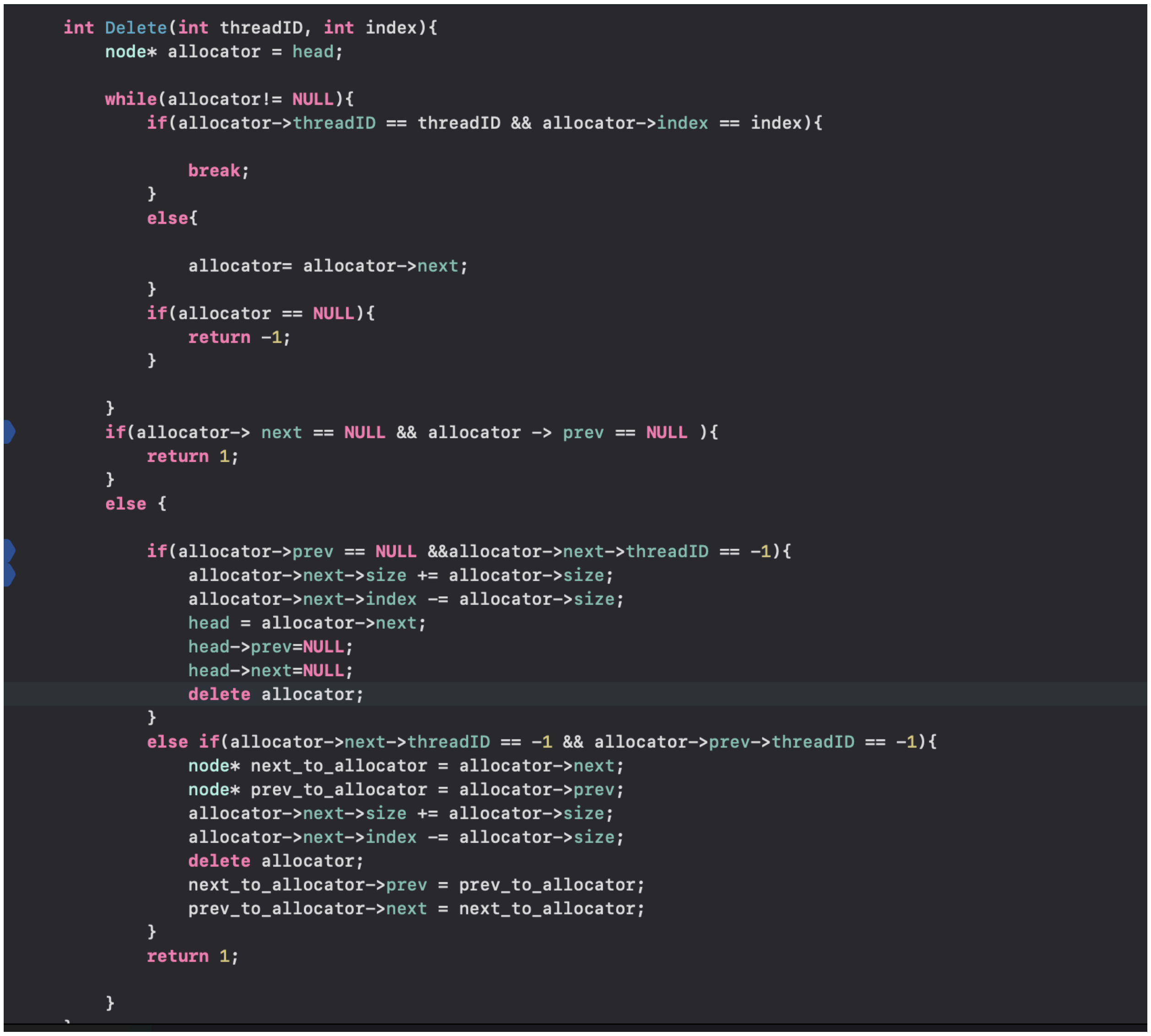
I have 3 functions inside my linked list class insert, delete, and print list.

Insert searches for the first free space node, finds it, and inserts the wanted space



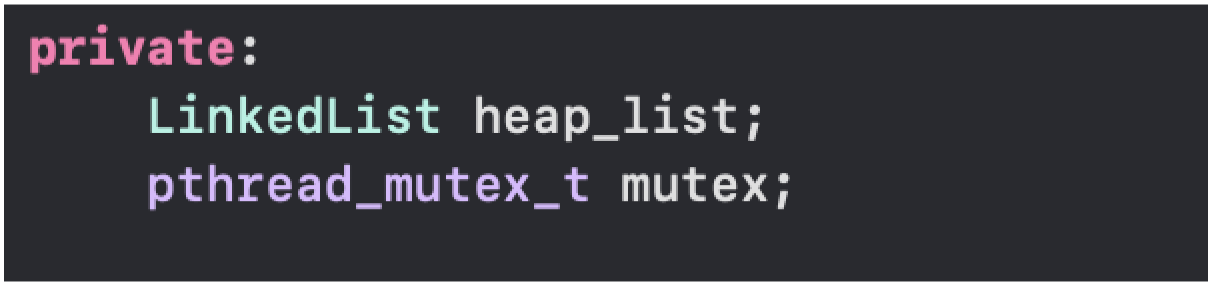
```
    int Insert(int threadID, int s){
        int index;
        node* allocator = head;

        if(allocator->size <s){
            return -1;
        }

        while(allocator!= NULL){
            if(allocator->size >= s && allocator->threadID == -1){
                break;
            }
            else{
                allocator= allocator->next;
            }
            if(allocator == NULL){
                return -1;
            }

        }

        allocator->size = allocator->size - s ;
        node* new_node = new node(threadID,s,allocator->index);


        new_node -> next = allocator;
        new_node->prev = allocator->prev;

        if(allocator->prev != NULL){
            allocator->prev->next= new_node;
        }

        allocator->prev = new_node;//now new_node is ahead of allocator node so update prev
        //also set the new_node's next node
        if(new_node->prev == NULL){
            head = new_node;

        }
        allocator->index += new_node->size;
        new_node->index =   allocator->index - new_node->size;


        return index;
```

Delete searches for the given and then find it. First, check if it is the only node if yes then the function returns 1. If not, it checks that the allocator is the head of the linked list or if it is between two free spaces for the coalescing operation.



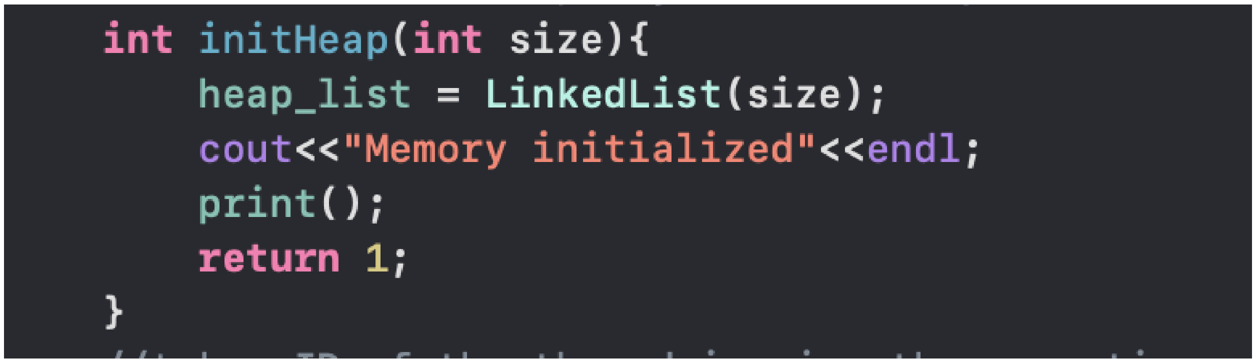
```
    int Delete(int threadID, int index){
        node* allocator = head;

        while(allocator!= NULL){
            if(allocator->threadID == threadID && allocator->index == index){

                break;
            }
            else{

                allocator= allocator->next;
            }
            if(allocator == NULL){
                return -1;
            }

        }
        if(allocator-> next == NULL && allocator -> prev == NULL ){
            return 1;
        }
        else {

            if(allocator->prev == NULL &&allocator->next->threadID == -1){
                allocator->next->size += allocator->size;
                allocator->next->index -= allocator->size;
                head = allocator->next;
                head->prev=NULL;
                head->next=NULL;
                delete allocator;
            }
            else if(allocator->next->threadID == -1 && allocator->prev->threadID == -1){
                node* next_to_allocator = allocator->next;
                node* prev_to_allocator = allocator->prev;
                allocator->next->size += allocator->size;
                allocator->next->index -= allocator->size;
                delete allocator;
                next_to_allocator->prev = prev_to_allocator;
                prev_to_allocator->next = next_to_allocator;
            }
            return 1;

        }
```

The second class is my heap manager class. I have 2 private members one of them is Linked List type and the second is the mutex lock.



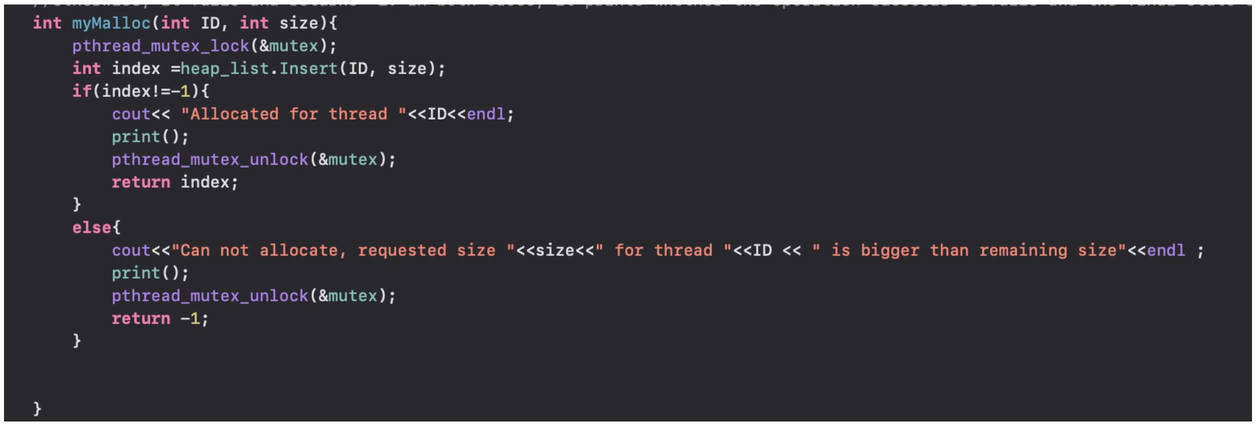
```
private:
    LinkedList heap_list;
    pthread_mutex_t mutex;
```

It has a default constructor that initializes the mutex variable. After that, I have my init function which assigns my linked list and handles the implementation.



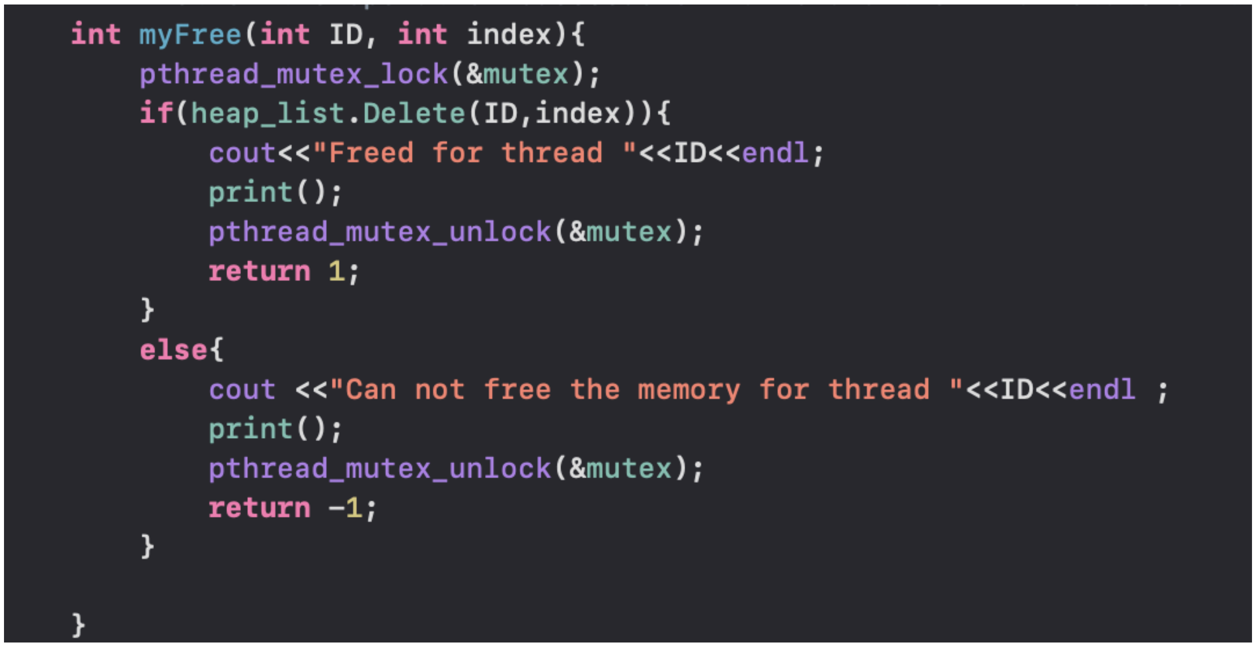
```
int initHeap(int size){
    heap_list = LinkedList(size);
    cout<<"Memory initialized"<<endl;
    print();
    return 1;
}
```

Then I have myMalloc function it obtains the lock and then uses the insert function of my linked list class if the returned value is 1 then we can allocate a space for the thread if not we can not allocate.



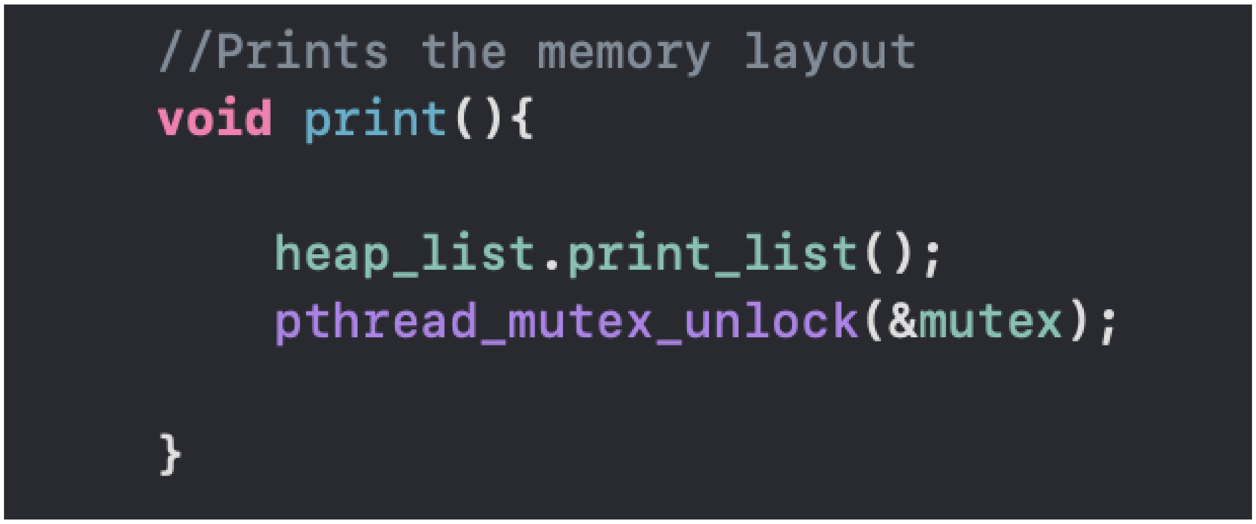
```
int myMalloc(int ID, int size){
    pthread_mutex_lock(&mutex);
    int index =heap_list.Insert(ID, size);
    if(index!=-1){
        cout<< "Allocated for thread "<<ID<<endl;
        print();
        pthread_mutex_unlock(&mutex);
        return index;
    }
    else{
        cout<<"Can not allocate, requested size "<<size<<" for thread "<<ID << " is bigger than remaining size"<<endl ;
        print();
        pthread_mutex_unlock(&mutex);
        return -1;
    }


}
```

Moreover, I have myFree function it obtains the lock and uses the delete function of the linked list class if it is successful we deallocate the memory if not there is an error message displayed.



```
int myFree(int ID, int index){
    pthread_mutex_lock(&mutex);
    if(heap_list.Delete(ID,index)){
        cout<<"Freed for thread "<<ID<<endl;
        print();
        pthread_mutex_unlock(&mutex);
        return 1;
    }
    else{
        cout <<"Can not free the memory for thread "<<ID<<endl ;
        print();
        pthread_mutex_unlock(&mutex);
        return -1;
    }

}
```

Lastly I have my free function this does not obtains the lock but it frees because it is assumed that it will be used in other function not lonely.



```
//Prints the memory layout
void print(){

    heap_list.print_list();
    pthread_mutex_unlock(&mutex);

}
```