```c
// Glenn Hewey
// CECS 424
// Lab 2 - Dynamic memory allocator (heap manager) in C
// Due: March 9 ,2018

#include "stdio.h"
#include "stdlib.h"

struct Block {
    int block_size; // # of bytes in the data section
    struct Block *next_block; // in C, you have to use •struct Block• as the type
};

const int OVERHEADSIZE = sizeof(struct Block);
const int MINSIZE = sizeof(void*);
struct Block *free_head;
// points to memory allocated with malloc inorder to deallocate at end of program
struct Block *head;

// initialize heap to specified size
void my_initialize_heap(int size){
    free_head = (struct Block*) malloc(size);
    (*free_head).block_size = size - OVERHEADSIZE;
    (*free_head).next_block = NULL;
}

void remove_front() {

    // temp head
    struct Block *front = free_head;

    // point free head to the next block
    free_head = (*free_head).next_block;

    // set the next pointer of the block you just removed equal to null
    (*front).next_block = NULL;
}

void remove_back() {
    struct Block *cursor = free_head;

    // keeps track of the previous block when traversing the free list
    struct Block *back = NULL;

    // traverse free list till you find the tail
    while((*cursor).next_block != NULL)
    {
        back = cursor;
        cursor = (*cursor).next_block;
    }

    // back will not equal null if you traversed the free list
    // remove the block from the free list
    if(back != NULL){
        (*back).next_block = NULL;
    }

}
```

```
// Remove a block from the free list depending on if it is
// in the front back or in the middle of the free list
void my_remove(struct Block* nd){

        // if current block is the head of the free list
        if(nd == free_head){
            remove_front();
        }

        // if current block is the tail of the free list
        if((*nd).next_block == NULL){
            remove_back();
        }

        // if the current block is in the middle of the free list
        struct Block* cursor = free_head;
        while(cursor != NULL)
        {
            if((*cursor).next_block == nd)
                break;
            cursor = (*cursor).next_block;
        }

        if(cursor != NULL)
        {
            struct Block* tmp = (*cursor).next_block;
            (*cursor).next_block = (*tmp).next_block;
            (*tmp).next_block = NULL;
        }
}

void my_free(void* addr){

    // set the blocks next equal to the free head
    struct Block* block = (void*)addr-OVERHEADSIZE;
    (*block).next_block = free_head;

    // set free_head equal to the the block we just freed
    free_head = block;
}

// Rounds the size requested up to a multiple of the MINSIZE = 8
(sizeof(void*))
int RoundUpToMultiple(int size){
    int remainder = abs(size) % MINSIZE;

    if (remainder == 0)
        return size;

    return size + MINSIZE - remainder;
}

void *my_alloc(int size){

    // Round up size
    size = RoundUpToMultiple(size);

    int found = 0;

```

```c
    // Walk free_head
    struct Block *cur = free_head;
    while(cur != NULL){
        if ( size <= (*cur).block_size ){
            found = 1;
            break;
        } else {
            cur = (*cur).next_block;
        }
    }

    // if there is a block with enough space. found will be set to 1
    // if not return 0

    if(found == 1){

        // current free block is large enough to fit the size being allocated
        // AND the excess space in the data portion is sufficient to fit another block with over head
        if( ((*cur).block_size - size - OVERHEADSIZE - MINSIZE) > 0 ){

            // Split

            //Calculate the left over size after the split
            int leftOver = (*cur).block_size - size;

            // create a block address of new block is equal to the cur addr + overhead + size
            struct Block* block = (void*)cur+OVERHEADSIZE+size;

            // Set the next block equal to the current blocks next block
            (*block).next_block = (*cur).next_block;

            // Set the new blocks size equal to the leftover size - the required overheadsize
            (*block).block_size = leftOver - OVERHEADSIZE;
            free_head = block;

            // Set the size of the current block equal to the size requested
            (*cur).block_size = size;

            // Remove the next pointer does not point to anything
            // because it is not a part of the free_list anymore
            (*cur).next_block = NULL;

        }else{

            //Remove from free list
            my_remove(cur);

        }
        // return addr to new block
        // add 16 bytes to get the address of the data portion of the block
        return (void*)cur+OVERHEADSIZE;
    }else{
        return 0;
    }
}

int main()
```

```
{
    void* a;
    void* b;
    void* c;
    void* d;
    void* e;
    int n = 5;

    my_initialize_heap(1000);

    switch(n){
        case 1:
            //Test 1
            printf("\tTest 1\n");
            a = my_alloc(sizeof(int));
            printf("a: %p\n", a);
            my_free(a);
            b = my_alloc(sizeof(int));
            printf("b: %p\n", b); // The addresses should be the same.
            break;
        case 2:
            //Test 2
            printf("\tTest 2\n");
            a = my_alloc(sizeof(int));
            b = my_alloc(sizeof(int));
            printf("a: %p\n", a);
            printf("b: %p\n", b); // address should be 24 (0x18) bytes apart
            break;
        case 3:
            //Test 3
            printf("\tTest 3\n");
            a = my_alloc(sizeof(int));
            b = my_alloc(sizeof(int));
            c = my_alloc(sizeof(int));
            printf("a: %p\n", a);
            printf("b: %p\n", b);
            printf("c: %p\n", c);
            my_free(b);
            d = my_alloc(sizeof(double)); // should be the same as b's address
            printf("d: %p\n", d);
            e = my_alloc(sizeof(int)); // address should be 24 (0x18) bytes apart from c
            printf("e: %p\n", e);
            break;
        case 4:
            //Test 4
            printf("\tTest 4\n");
            a = my_alloc(sizeof(char));
            b = my_alloc(sizeof(int));
            printf("a: %p\n", a);
            printf("b: %p\n", b); //Should equal b in test 2
            break;
        case 5:
            //Test 5
            printf("\tTest 5\n");
            a = my_alloc(sizeof(int[100]));
            b = my_alloc(sizeof(int));
            printf("a: %p\n", a);
            printf("b: %p\n", b);
```

```
            my_free(a);
            printf("b: %p\n", b); // Should not change
            break;
        default:
            printf("\n");
    }
    free(head);
    return 0;
}

```