```
////////////////////////////////////////////////////////////
//
//  CECS 424 lab1 (Heap Implementation))
//
//  author: Lyndon Kondratczyk
//
//  version: 9/14/15
//
//  A simple heap allocation program using first-fit selection
//
////////////////////////////////////////////////////////////


#include <stdio.h>
#include <stdlib.h>
#include <math.h>

const int msize = 8; //increased from sizeof(int) to avoid seg. faults
const int psize = sizeof(void*);
void* free_head;
void* buffer; //separate var for this since free_head changes reference

void my_initialize_heap(int);
void* my_alloc(int);
void my_free();
void extra_test_run();
void class_test_run();
void program();
void std_deviation(int*, int);
void print_contents(int);
void populate_block(void*, int);
void free_block(void*, int);

int main(){
    extra_test_run();
    class_test_run();
    program();
    return 0;
}

/*
Allocates memory to the buffer pointer, which should be freed after use

@param size The size of memory to allocate
*/
void my_initialize_heap(int size){
    int overhead = msize + psize;
    //avoid segmentation faults
    while(size % psize != 0){
        size ++;
    }
    buffer = malloc(size);
    //initialize free head and initial block info
```

```
    free_head = buffer;
    *((int*)free_head) = (int)(size - overhead);
    *((void**)(free_head + msize)) = NULL;
}

/*
Makes a request to reserve memory in buffer using heap allocation

@param size The amount of memory being requested

@return The starting point of memory reserved, or NULL of no memory available
*/
void* my_alloc(int size){
    //avoid segmentation faults
    while(size % psize != 0){
        size ++;
    }

    if((free_head != NULL)){
        //set "walking" variables
        void* current = free_head;
        void* next = current + msize;
        void* previous = NULL;

        //walk until memory found or end of free list reached
        while(*((int*)current) < size && (*((void**)next) != NULL)){
            previous = current;
            current = *((void**)(next));
            next = current + msize;
        }

        //memory found, now test if new block needed
        if(*((int*)current) >= size) {

            //enough space for new block if true
            if(*((int*)current) >= size + psize + psize + msize){
                //create new block
                void* new_block = current + size + msize + psize;
                *((void**)(new_block + msize)) = *((void**)(next));
                *((int*)new_block) = *((int*)current) -
                        (int)(size + msize + psize);
                //update current block size
                *((int*)current) = size;

                //switch free_head
                if (previous == NULL)
                    free_head = new_block;
                //switch previous pointer
                else
                    *((void**)(previous + msize)) = new_block;
            } else{ //not enough space for new block
                //link previous's next pointer to current's next
                if (previous != NULL)
```

```
                    *((void**)(previous + msize)) =
                            *((void**)(current + msize));
                //link free_head to free_head's next
                else
                    free_head = *((void**)(free_head + msize));
            }
            //clear current's next pointer (inf any)
            *((void**)(current + msize)) = NULL;
            //return a pointer to current's data section
            return (void*)(current + msize + psize);
        }
        //not enough space
        return NULL;
    }
    //no free list
    return NULL;
}

/*
Returns a reserved block of data to the free list

@param data A pointer to the data segment of the heap (hopefully)
*/
void my_free(void* data){
    if(data != NULL){
        *((void**)(data - psize)) = free_head;
        free_head = data - msize - psize;
    }
}

/*
Coolects user data for calculating the standard deviation
*/
void program(){
    printf("\nPlease input the value of n (integer)\n");
    int size = 0;
    scanf("%i", &size);
    printf("%i entered \n", size);

    my_initialize_heap((int)((size * psize) + msize + psize));
    int* int_array = my_alloc((int)(size * psize));

    int i;
    for(i = 0; i < (int)(size); i++){
        int data = 0;
        printf("Please input integer number %d:\n", i);
        scanf("%i", &data);
        *(int_array + i) = (int)data;
    }
    std_deviation(int_array, size);
    free(buffer);
}
```

```
/*
Calculates the standard deviation of an int array

@param array An int array
@param size The size of the int array
*/
void std_deviation(int* array, int size){
    float accum = 0;
    //calculate mean
    int i;
    for(i = 0; i < size; i++){
        accum += *((int*)array + i);
    }
    float mean = (float)accum / (float)size;
    printf("mean = %f \n", mean);

    //calcutate sum((f(n) - mean)^2)
    accum = 0;
    for(i = 0; i < size; i++){
        accum += pow((((float)*((int*)array + i))) - mean, (float)2);
    }
    //calculate standard deviation
    float sigma = sqrt(((float)1/(float)size) * accum);
    printf("deviation = %f \n", sigma);
}

/*
A graphical test for visualizing the heap
*/
void extra_test_run(){
    /*
    printf("GRAPHICAL TEST\n");

    int size = 72;
    my_initialize_heap(size);

    void* entry1 = my_alloc(4);
    populate_block(entry1, 4);
    print_contents(size);

    void* entry2 = my_alloc(8);
    populate_block(entry2, 8);
    print_contents(size);

    void* entry3 = my_alloc(4);
    populate_block(entry3, 4);
    print_contents(size);

    void* entry4 = my_alloc(16);
    populate_block(entry4, 16);
    print_contents(size);

    my_free(entry1);
```

```
    free_block(entry1, 4);
    print_contents(size);

    my_free(entry2);
    free_block(entry2, 8);
    print_contents(size);


    my_free(entry3);
    free_block(entry3, 4);
    print_contents(size);

    entry2 = my_alloc(8);
    populate_block(entry2, 8);
    print_contents(size);

    entry3 = my_alloc(8);
    populate_block(entry3, 8);
    print_contents(size);

    entry4 = my_alloc(16);
    populate_block(entry4, 16);
    print_contents(size);

    entry1 = my_alloc(8);
    populate_block(entry1, 8);
    print_contents(size);

    free(buffer);
    */
}

/*
The minimum test requirements for the class with console outputs
*/
void class_test_run(){
    int size = 1024;
    my_initialize_heap(size);

    printf("Test 1:\n");
    int i = 0;
    int test;
    int* test1a = my_alloc(sizeof(int));
    printf("Allocated address 1: %ld \n", test1a);
    my_free(test1a);
    printf("Allocation 1 freed: \n");
    int* test1b = my_alloc(sizeof(int));
    printf("Allocated address 2: %ld \n\n", test1b);

    printf("Test 2:\n");
    int* test2a = my_alloc(sizeof(int));
    printf("Allocated address 1: %ld \n", test2a);
    int* test2b = my_alloc(sizeof(int));
```

```
    printf("Allocated address 2: %ld \n", test2b);
    printf("Address 2 - 1 = %ld - %ld = %ld \n",test2b, test2a,
            (test2b - test2a)*sizeof(int));
    printf("Size of int = %i, Size of overhead = %i, sum = %i\n\n", msize,
            (msize + psize), (msize + msize + psize));

    printf("Test 3:\n");
    printf("!!!Note, void* size is 8, so double * 2 used:\n");
    int* test3a = my_alloc(sizeof(int));
    printf("Allocated address 1: %ld \n", test3a);
    int* test3b = my_alloc(sizeof(int));
    printf("Allocated address 2: %ld \n", test3b);
    int* test3c = my_alloc(sizeof(int));
    printf("Allocated address 3: %ld \n", test3c);
    my_free(test3b);
    printf("Allocation 2 freed: \n");
    double* test3d = my_alloc(2 * sizeof(double));
    printf("Allocated address 4 (double double): %ld \n", test3d);
    int* test3e = my_alloc(sizeof(int));
    printf("Allocated address 5 (int): %ld \n\n", test3e);

    printf("Test 4:\n");
    char* test4a = my_alloc(sizeof(char));
    printf("Allocated address 1: %ld \n", test4a);
    int* test4b = my_alloc(sizeof(int));
    printf("Allocated address 2: %ld \n", test4b);
    printf("Address 2 - 1 = %ld - %ld = %ld \n",test4b, test4a,
            ((int*)test4b - (int*)test4a)*sizeof(int));
    printf("Size of int = %i, Size of overhead = %i, sum = %i\n\n", msize,
            (msize + psize), (msize + msize + psize));

    printf("Test 4:\n");
    int** test5a = my_alloc(sizeof(int) * 100);
    printf("Allocated address 1: %ld \n", test5a);
    int* test5b = my_alloc(sizeof(int));
    printf("Allocated address 2: %ld \n", test5b);
    printf("Address 2 - 1 = %ld - %ld = %ld \n",test5b, test5a,
            ((int*)test5b - (int*)test5a)*sizeof(int));
    printf("Size of array = %i, Size of overhead = %i, sum = %i\n",
            sizeof(int) * 100, (msize + psize),
            (sizeof(int) * 100 + msize + psize));
    my_free(test5a);
    printf("Allocation 1 freed: \n");
    printf("Allocated address 2: %ld \n", test5b);

    free(buffer);
}

/*
Prints the contents of the buffer

@param size The current allocated size of the buffer
*/
```

```
void print_contents(int size){
    int i;
    for(i = 0; i < size; i ++){
        if((buffer + i) == free_head){
            printf("[%i]: free_head \n", i);
        }
        printf("[%i]:%c \n", i, *((char*)buffer + i));
    }
}

/*
Populates a reserved block of data for visualization

@param dataBlock A pointer to the data to be populated (set to 'A')
@param size The size of the data block
*/
void populate_block(void* dataBlock, int size){
    int i;
    for(i = 0; i < size; i++){
        *(((char*)dataBlock + i)) = 'A';
    }
}

/*
"Frees" a reserved block of data for visualization

@param dataBlock A pointer to the data to be freed (set to 'Z')
@param size The size of the data block
*/
void free_block(void* dataBlock, int size){
    if(dataBlock == NULL){
        printf("oops\n");
    }
    else{
        int i;
        for(i = 0; i < size; i++){
            *(((char*)dataBlock + i)) = 'Z';
        }
    }
}
```