```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>

struct Block {
    int block_size;
    struct Block* next_block;
};

const int OVERHEAD_SIZE = sizeof(struct Block); // refers to size & pointer of a block
const int POINTER_SIZE = sizeof(void*);
struct Block* free_head;

void my_initialize_heap(int size) {
    void* heap_buffer = malloc(size);
    if (heap_buffer == NULL) {
        printf("Error: Failed to initialize heap. Out of memory.\n");
        exit(EXIT_FAILURE);
    }

    free_head = (struct Block*)heap_buffer;
    free_head->block_size = size - OVERHEAD_SIZE;
    free_head->next_block = NULL;
}

void* my_alloc(int size) {
    if (size <= 0) {
        printf("Error: Size must be greater than 0.\n");
        return NULL;
    }
    // size must be a multiple of POINTER_SIZE. So, if pointer_size is 4, size must be at least 4, 8, 12...
    int adjusted_size = size + (POINTER_SIZE - size % POINTER_SIZE) % POINTER_SIZE;

    // Iterator
    struct Block* curr = free_head;
    struct Block* prev = NULL;

    bool found = false;
    // Iterate through each node, if a node has equal or more space than necessary to hold size, use that node.
    while (curr != NULL) {
        if (curr->block_size >= adjusted_size) {
            found = true;
            // Determine if the current block can be split.
            if (curr->block_size >= adjusted_size + OVERHEAD_SIZE + POINTER_SIZE) { // Splittable
                // Create a pointer to the newly split block's position then assign its structure members.
                struct Block* new_block = (struct Block*)((char*)curr + OVERHEAD_SIZE + adjusted_size);
                new_block->block_size = curr->block_size - adjusted_size - OVERHEAD_SIZE;
                new_block->next_block = curr->next_block;

                // Update Curr's block size as a result of splitting.
                curr->block_size = adjusted_size;

                // Adjust the linked list, depending on whether curr is the head or not.
                if (curr == free_head) {
                    free_head = new_block;
                } else {
                    prev->next_block = new_block;
                }
            } else { // Not splittable
                // If curr is the head, curr's next block is the new head.
                if (curr == free_head) {
                    free_head = curr->next_block;
                }
                // If curr is not the head, the previous block points to curr's next block.
                else {
                    prev->next_block = curr->next_block;
                }
            }
            // Since we found a block, no need to keep searching.
            break;
        }
        // Haven't found an available space yet.
        else {
            prev = curr;
            curr = curr->next_block;
        }
    }
    // Return a pointer to the allocated data, if possible.
    if (found) {
        return (char*)curr + OVERHEAD_SIZE;
    } else {
        return NULL;
    }
}

void my_free(void* data) {
    struct Block* freed_block = (struct Block*)((char*)data - OVERHEAD_SIZE);
    freed_block->next_block = free_head;
    free_head = freed_block;
}

void menuOptionOne() {
    int *numOne = my_alloc(sizeof(int));
    printf("Address of int A: %p\n", numOne);
    my_free(numOne);

    int *numTwo = my_alloc(sizeof(int));
    printf("Address of int B: %p\n", numTwo);
}

//Allocate two ints and print their addresses; they should be exactly the size of your overhead plus the
//larger of (the size of an integer; the minimum block size) apart.
void menuOptionTwo() {
    int *numOne = my_alloc(sizeof(int));
    printf("Address of int A: %p\n", numOne);
    int *numTwo = my_alloc(sizeof(int));
    printf("Address of int B: %p\n", numTwo);
    printf("Verifying Results...\n");
    int overheadPlusLarger = OVERHEAD_SIZE + sizeof(void*);
    printf("Size of overhead + larger of (the size of an integer; the minimum block size): %d bytes\n", overheadPlusLarger);
    printf("Address B - Address A: %ld bytes \n", (uintptr_t)numTwo - (uintptr_t)numOne);
}

//Allocate three ints and print their addresses, then free the second of the three. Allocate an array of 2
//double values and print its address (to allocate an array in C, allocate (2 * sizeof(double)); verify
//that the address is correct. Allocate another int and print its address; verify that the address is the
//same as the int that you freed.
void menuOptionThree() {
    int *numOne = my_alloc(sizeof(int));
    printf("Address of int A: %p\n", numOne);
    int *numTwo = my_alloc(sizeof(int));
    printf("Address of int B: %p\n", numTwo);
    int *numThree = my_alloc(sizeof(int));
    printf("Address of int C: %p\n", numThree);
    my_free(numTwo);

    printf("After freeing int B...\n");
    double *arr = my_alloc(2 * sizeof(double));
    printf("Address of array of 2 double values: %p\n", arr);

    int *numFour = my_alloc(sizeof(int));
    printf("Address of int D (should be the int B): %p\n", numFour);
}

//Allocate one char, then allocate one int, and print their addresses. They should be exactly the same
//distance apart as in test #2.
void menuOptionFour() {
    char *charOne = my_alloc(sizeof(char));
    printf("Address of char A: %p\n", charOne);
    int *numTwo = my_alloc(sizeof(int));
    printf("Address of int B: %p\n", numTwo);

    int overheadPlusLarger = OVERHEAD_SIZE + sizeof(void*);
    printf("Size of overhead + larger of (the size of an integer; the minimum block size): %d\n", overheadPlusLarger);
}

//Allocate space for a 80-element int array, then for one more int value. Verify that the address of
//the int value is 80 * sizeof(int) + the size of your header after the array's address. Free the array.
//Verify that the int's address and value has not changed.
void menuOptionFive() {
    int *arr = my_alloc(80 * sizeof(int));
    int *numOne = my_alloc(sizeof(int));
    printf("Address of array: %p\n", arr);
    printf("Address of int A: %p\n", numOne);
    printf("Address of int value: %p\n", (void*)((uintptr_t)arr + 80 * sizeof(int) + OVERHEAD_SIZE));
    printf("Value of int A: %d\n", *numOne);

    printf("Difference between array and int A: %ld\n", (uintptr_t)numOne - (uintptr_t)arr);

    my_free(arr);

    printf("After freeing array...\n");
    printf("Address of int value: %p\n", numOne);
    printf("Value of int A: %d\n", *numOne);
}

int main() {
    int menuChoice = 0;
    int runAgain = 1;
    my_initialize_heap(1000); // Initialize heap once outside the loop

    while (runAgain == 1) {
        printf("\n1. Allocate an int \n2. Allocate two ints \n3. Allocate three ints \n4. Allocate one char \n5. Allocate space for an 80-element int array \n6. Quit \nChoose a menu option: ");
        scanf("%d", &menuChoice);
        printf("\n---Test Case %d---\n", menuChoice);

        switch(menuChoice) {
            case 1:
                menuOptionOne();
                break;
            case 2:
                menuOptionTwo();
                break;
            case 3:
                menuOptionThree();
                break;
            case 4:
                menuOptionFour();
                break;
            case 5:
                menuOptionFive();
                break;
            case 6:
                printf("Done!\n");
                runAgain = 0;
                break;
            default:
                printf("Invalid choice. Please select a valid option.\n");
        }
    }
    return 0;
}
```

```
PS C:\Users\amnes\Documents\GitHub\CECS-342\lab3> ./lab3

1. Allocate an int
2. Allocate two ints
3. Allocate three ints
4. Allocate one char
5. Allocate space for an 80-element int array
6. Quit
Choose a menu option: 1

---Test Case 1---
Address of int A: 000002456593D530
Address of int B: 000002456593D530

1. Allocate an int
2. Allocate two ints
3. Allocate three ints
4. Allocate one char
5. Allocate space for an 80-element int array
6. Quit
Choose a menu option: 2

---Test Case 2---
Address of int A: 000002456593D548
Address of int B: 000002456593D560
Verifying Results...
Size of overhead + larger of (the size of an integer; the minimum block size): 24 bytes
Address B - Address A: 24 bytes

1. Allocate an int
2. Allocate two ints
3. Allocate three ints
4. Allocate one char
5. Allocate space for an 80-element int array
6. Quit
Choose a menu option: 3

---Test Case 3---
Address of int A: 000002456593D578
Address of int B: 000002456593D590
Address of int C: 000002456593D5A8
After freeing int B...
Address of array of 2 double values: 000002456593D5C0
Address of int D (should be the int B): 000002456593D590
```

```
1. Allocate an int
2. Allocate two ints
3. Allocate three ints
4. Allocate one char
5. Allocate space for an 80-element int array
6. Quit
Choose a menu option: 4

---Test Case 4---
Address of char A: 0000002456593D5E0
Address of int B: 0000002456593D5F8
Size of overhead + larger of (the size of an integer; the minimum block size): 24

1. Allocate an int
2. Allocate two ints
3. Allocate three ints
4. Allocate one char
5. Allocate space for an 80-element int array
6. Quit
Choose a menu option: 5

---Test Case 5---
Address of array: 0000002456593D610
Address of int A: 0000002456593D760
Address of int value: 0000002456593D760
Value of int A: 0
Difference between array and int A: 336
After freeing array...
Address of int value: 0000002456593D760
Value of int A: 0

1. Allocate an int
2. Allocate two ints
3. Allocate three ints
4. Allocate one char
5. Allocate space for an 80-element int array
6. Quit
Choose a menu option: 6

---Test Case 6---
Done!
```