COP 3502C Programming Assignment # 1
Dynamic Memory Allocation
Read all the pages before starting to write your code

Overview

This assignment is intended to make you do a lot of work with dynamic memory allocation! Don't wait until the weekend it's due to start it!

Your solution should follow a set of requirements to get credit.

What should you submit?

Write all the code in a single file main.c file and submit it on codegrade.

Please include the following commented lines in the beginning of your code to **declare your authorship of the code:**

/* COP 3502C Assignment 1

This program is written by: Your Full Name */

Compliance with Rules: UCF Golden rules apply towards this assignment and submission. Assignment rules mentioned in syllabus, are also applied in this submission. The TA and *Instructor can call any students for explaining any part of the code in order to better assess your authorship and for further clarification if needed.*

Caution!!!

Sharing this assignment description (fully or partly) as well as your code (fully or partly) to anyone/anywhere is a violation of the policy. I may report to office of student conduct and an investigation can easily trace the student who shared/posted it. Also, getting a part of code from anywhere will be considered as cheating.

Deadline:

See the deadline in Webcourses. The assignment will accept late submission up to 24 hours after the due date time with 10% penalty. After that the assignment submission will be locked. An assignment submitted by email will not be graded and such emails will not be replied according to the course policy.

What to do if you need clarification on the problem?

I will create a discussion thread in webcourses and I highly encourage you to ask your question in the discussion board. Maybe many students might have same question like you. Also, other students can reply and you might get your answer faster. Also, you can write an email to the TAs and put the course teacher in the cc for clarification on the requirements.

How to get help if you are stuck?

According to the course policy, all the helps should be taken during office hours. Occasionally, we might reply in email.

Problem: Assigned Seating

Objective

Give practice with structs and dynamic memory in C.

Give practice with functions in C.

Give practice following a design specification with structs and function prototype requirements.

Background Story

The demand for movies is higher than ever, and you are going to use this to your advantage to make some decent cash. You need your theater to stand out from its competitors, so you are going to introduce your Unacceptably Luminescent Tech Igniting Many Airplanes Though Eco-friendly (ULTIMATE) projector.

The projector in question can be seen for miles, and it works in the sunlight too. You are going to set up a large theater that can fit many people. For now, you need some software that can track who is in what seat efficiently. It seems the software you got from buying out a theater could only handle up to 400-seat theaters. You have *much* larger ambitions.

Problem Description

Write a program that can allow for people to buy consecutive seats in a row (if available) and look up who owns the seat at a particular spot in a row. You should assume that the theater starts with every seat available for purchase.

Input

The input format consists of a sequence of actions and requests, specified in the order in which they occur. Instead of having a fixed number of lines, the input will be sentinel driven; this means that a special word on a line by itself will specify that there is no more input to process.

Each line of input will be in one of the following formats:

- `BUY <ROW> <START> <END> <NAME>`
  - This means the user with the given `<NAME>` will try to purchase the seats in the given `<ROW>`. They want the seats with numbers ranging from `<START>` to `<END>` inclusive.
  - You are guaranteed that `<START>` will be at most `<END>`.
  - If any other user owns at least one seat in this range, the purchase should **NOT** go through.
- `LOOKUP <ROW> <SEAT>`
  - This means that we need to determine which user, if any, purchased the seat with the number `<SEAT>` in the given `<ROW>`.
- `EXIT`
  - This should terminate the program and clean up any memory.

<ROW>, <START>, <END>, and <SEAT> will all be positive integers in the range of 1 to 100,000, inclusive. The <NAME> will be an alphabetic single word string of at most 50 characters *(use scanf with %s)*. Finally, there will never be more than 100 successful BUY operations executed on a single row.

Output

For each BUY input, output a single line containing the word "SUCCESS" if the purchase was successful, and "FAILURE" otherwise. For each LOOKUP input, output a single line containing solely the name of the user that owns the seat, if the seat is owned and "None", if the seat is unowned.

Sample Input	**Sample Output**
BUY 1 5 10 ALICE BUY 1 7 7 Bob BUY 2 5 10 Carol LOOKUP 1 7 LOOKUP 1 10 LOOKUP 1 1 BUY 1 1 4 David LOOKUP 1 1 BUY 1 2 5 Eric BUY 10 1000 1000 SAM EXIT	SUCCESS FAILURE SUCCESS ALICE ALICE None SUCCESS David FAILURE SUCCESS
LOOKUP 1 1 LOOKUP 1 2 BUY 3 49998 50003 Guha BUY 1 50000 50000 Meade BUY 5 49999 50001 Ahmed LOOKUP 1 1 LOOKUP 1 49999 LOOKUP 50000 1 LOOKUP 3 50000 EXIT	None None SUCCESS SUCCESS SUCCESS None None None Guha

Sample Explanation

In the first sample, ALICE's buy is successful but Bob's isn't because he tries to buy seat 7 in row 1, which ALICE already has bought. Carol's buy is also successful in row 2.

ALICE does have seats 7 and 10 on the first row (because she bought seats 5 through 10 on that row), but does not have seat one on row 1.

David follows this up by buying all the seats with lower numbers on row 1 next to Alice's block of seats. Right after this buy, David has seat 1 in row 1. Eric can't buy the seats he wants in row 1 because they are already bought by both David and ALICE. Finally, SAM can buy seats in row 10 because they are all open to buy.

In the second sample, we start with 2 look ups, which can not be successful because no one has seats. Then, all three instructors buy seats in different rows. The next three look ups are all for seats that are unowned, while the last look up is for Guha's seat.

Required Structs and Function Prototypes

Here are three #defines for important constants to use:

```
#define INITSIZE 10

#define MAXLEN 50

#define MAXROWS 100000
```

The first represents the initial size of any row (in terms of number of orders).

The second represents the maximum length of a name for an order.

The third represents the maximum number of rows in the theater.

For safety reasons (or if you want to use 1-based indexing), feel free to add 1 to the last two.

The following struct definition should be used for an order:

```
typedef struct order {
    int s_seat;
    int e_seat;
    char* name;
} order;
```

The reason the row isn't stored in the order is that each row number will implicitly be equal to the index in the array storing the rows for the movie theater.

A second required struct will manage a single row. A vast majority of the logic of the assignment will be done in the functions associated with this struct:

```
typedef struct theaterrow {
    order** list_orders;
    int max_size;
    int cur_size;
} theaterrow;
```

Here are functions you **<u>are required to write</u>**. What is given is both the function signature and a description of what the function does:

```
// Returns a pointer to a dynamically allocated order storing the given
// start row, end row, and the name this_name. Note: strcpy should be
// used to copy the contents into the struct after its name field is
// dynamically allocated.
order* make_order(int start, int end, char* this_name);

// This function will allocate the memory for one theaterrow, including
// allocating its array of orders to a default maximum size of 10, and
// setting its current size to 0.
theaterrow* make_empty_row();

// Assuming that order1 and order2 point to orders on the same row, this
// function returns 1 if the orders conflict, meaning that they share // at
least 1 seat in common, and returns 0 otherwise.
int conflict(order* order1, order* order2)

// Returns 1 if the order pointed to by this_order can be added to the
// row pointed to by this_row. Namely, 1 is returned if and only if
// this_order does NOT have any seats in it that are part of any order
// already on this_row.
int can_add_order(theaterrow* this_row, order* this_order);
```

```
// This function tries to add this_order to this_row. If  successful,
// the order is added and 1 is returned. Otherwise, 0 is returned and
// no change is made to this_row. If the memory for this_row is full,
// this function will double the maximum # of orders allocated for the
// row (via realloc), before adding this_order, and adjust max_size and
// cur_size of this_row.
void add_order(theaterrow* this_row, order* this_order);

// If seat_num seat number in row number row is occupied, return a
// pointer to the owner's name. Otherwise, return NULL.
char* get_owner(theaterrow** theater, int row, int seat_num);

// If seat_num in the row pointed to by this_row is occupied, return a
// pointer to the string storing the owner's name. If no one is sitting
// in this seat, return NULL.
char* get_row_owner(theaterrow* this_row, int seat_num);

// This function returns 1 if the seat number seat_no is contained in
// the range of seats pointed to by myorder, and returns 0 otherwise.
int contains(order* myorder, int seat_no);

// Frees all memory associated with this_order.
void free_order(order* this_order);

// Frees all memory associated with this_row.
void free_row(theaterrow* this_row);
```

Finally, in the main function, the whole theater will be declared as follows (you can choose any variable name you want):

```
theaterrow** amc_= calloc(MAXROWS+1, sizeof(theaterrow*));
```

This allocates an array, where each element in the array is a **pointer to a theater row**. The memory associated with this variable should be freed, in main, at the very end of the program.

Implementation Requirements/Run Time Requirements

1. Please use either the malloc or calloc functions to dynamically allocate memory for all situations that require dynamically allocated memory as specified in the assignment write up. You may need to use realloc if the number of orders in a row exceeds the max_size

2. For full credit, the functions `can_add_order`, `add_order`, `get_owner`, and `get_row_owner`, should all run in linear time in the length of the relevant row upon which they operate. (Here linear time means, your code should not try to access all the items of the array multiple times while processing one of these functions.).

3. Your code must compile and execute on CodeGrade.

4. Please **do not** use **memory leak detector** header file and atexit line in your code like shown in Lab2 as this assignment is dealing with large number of malloc and using the mentioned leak detector will make the execution of your code extremely slow. Instead, use valgrind command to check whether you have any memory leak in your code or not and where is it happening. See the page on the dynamic memory allocation module about valgrind.

Deliverables

Please submit a single source file, `main.c`, via CodeGrade.

Rubric (subject to change):

According to the Syllabus, the code will be compiled and tested in Codegrade Platform for grading. If your code does not compile in Codegrade, we conclude that your code is not compiling and it will be graded accordingly. We will apply a set of test cases to check whether your code can produce the expected output or not. Failing each test case will reduce some grade based on the rubric given bellow. If you hardcode the output, you will get -200% for the assignment.

1. If a code does not compile the code may get 0. However, some partial credit maybe awarded, and it cannot be more than 35% even the code is almost complete
2. If you modify or do not use the required structure, you will get 0
3. If you use file i/o instead of using scanf, then you will get 0
4. Not using dynamic memory allocation for storing data will receive 0
5. There is no grade for a well indented and well commented code. But a bad indented code will receive 20% penalty. Not putting comment in some important block of code -10%
6. Implementing required functions and other requirements: 30%
    a. 5 pts for using the required structs (2.5+2.5)
    b. Implementing and using the 10 required functions: 25 pts (2.5 each)
7. Freeing up memory properly with zero memory leak (if all the required malloc implemented): (20%)
8. Passing test cases: 50%
9. Note that we will use more test cases while grading and the test cases will follow the assignment input and output specifications. Passing sample test cases cannot guarantee that your code will pass grading test cases as well. You should test your code thoroughly with your own test cases to make sure your code is solving the problem accurately.
10. There may be some partial credit if your code compiles, you take inputs properly, use dynamic memory allocation property, etc.

Some Steps (if needed) to check your output AUTOMATICALLY in a command line in repl.it or other compiler with terminal option:

You can run the following commands to check whether your output is exactly matching with the sample output or not.

Step1: Copy the sample output to sample_out.txt file and move it to the server

Step2: compile your code using typical gcc and other commands.

//if you use math.h library, use the -lm option with the gcc command. Also, note that scanf function returns a value depending on the number of inputs. If you do not use the returned value of the scanf, gcc command may show warning to all of the scanf. In that case you can use "-Wno-unused-result" option with the gcc command to ignore those warning. So the command for compiling your code would be:

 # gcc main.c -Wno-unused-result -lm (use -g as well if you plan to use valgrind and want to see the line numbers with the memory leak)

Step3: Execute your code and pass the sample input file as a input and generate the output into another file with the following command

$ ***./a.out < sample_in.txt > out.txt***

Step4: Run the following command to compare your out.txt file with the sample output file

```
$cmp out.txt sample_out.txt
```

The command will not produce any output if the files contain exactly same data. Otherwise, it will tell you the first mismatched byte with the line number.

Step4(Alternative): Run the following command to compare your out.txt file with the sample output file

```
$diff -y out.txt sample_out.txt
```

The command will not produce any output if the files contain exactly same data. Otherwise, it will tell you the all the mismatches with more details compared to cmp command.

 # diff -c myout1.txt sample_out1.txt //this command will show ! symbol to the unmatched lines.

Some hints:

- **Make sure you have a very good understanding of Dynamic memory allocation based on the lecture, exercises, labs, and recordings**
- **The core concepts of the example of dynamically allocating array of structure pointer, dynamically allocating array of strings, and the Lab2 code would be very useful before starting this assignment. You should really watch these specific recording before starting this assignment.**
- **Start the assignment as soon as possible.**
- **Break it down by drawing and designing,**
- **At first see whether you can read all the inputs properly**
- **Then gradually implement functions one by one and test your code gradually.**
- **Do not wait till the end to test your code.**
- **Do not hesitate to take help during all of our office hours.**

Good Luck!