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 work with dynamic memory allocation, pointers, and arrays of pointers. The difficulty level is not actually very high, but it might take sometimes to do it depending on your understanding of 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.

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 from Webcourses. The assignment will accept late submission up to 24 hours after the due date time with 10% penalty (note that this is not a default rule for all the future assignments!). After that deadline, the 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 Description: Monster Campus Parking System

Monster University is facing significant challenges with their parking system due to heavy demand and limited availability, particularly during peak hours. To better understand the usage patterns of various parking garages and ensure that only authorized vehicles can park, they need to develop a more effective monitoring system. Recognizing your exceptional coding skills, they have sought your expertise to create this system

The campus features multiple parking garages, each with a defined capacity. The university also maintains a record of registered vehicles along with their owner's details. Your task is to develop a system that efficiently tracks vehicles entering and exiting the garages while monitoring the status of each garage in real-time.

Requirements:

A. Struct Definitions: You must use the following struct definitions. You are free to add more fields if you anticipate.

```
typedef struct RegisteredVehicle{  //for one registered vehicle
    char* license_plate; //to be used for string license plate# of the vehicle
    char* owner_name;    //to be used for storing owner name
} RegisteredVehicle;

typedef struct Garage{  //for one garage
    char* garage_name;  //to be used for garage name
    int total_capacity; //total capacity of the garage
    int current_count; // number of vehicles in the garage in a particular time
    RegisteredVehicle** parked_vehicles; //list of parked vehicles in a particular time
} Garage;

typedef struct Campus{
    Garage** garages; //list of garages in campus
    int total_garages; //number of garages in the campus
    RegisteredVehicle** registered_vehicles; //list of registered vehicles
    int total_registered_vehicles; //total number of registered vehicles
} Campus;
```

B. Commands:

The parking system should support the following commands:

a. PARK <LP> <G>: This command parks the car with license plate LP to garage G.
   i. If the garage is full, it should print "FULL"
   ii. Otherwise, print "PARKED" after successfully parking the vehicle at garage G
   iii. It is guaranteed that the license plate <LP> and garage <G> will be available in the system before processing this command

b. UTILIZATION_REPORT: this prints the utilization report in the following format.

```
Garage: <G1>, Capacity: <c1>, Occupied: <o1>, Utilization: <u1>%
Garage: <G2>, Capacity: <c2>, Occupied: <o2>, Utilization: <u2>%
………….
Garage: <Gn>, Capacity: <cn>, Occupied: <on>, Utilization: <un>%
```

Where, <Gx> is the garage name, <cn> is the capacity of garage Gx, <on> is count of vehicles at <Gx>, and <ux> is the percentage of utilization of the garage Gx. <ux> has to be displayed in 2 decimal places. Use double data type in this calculation.

After printing the above summary, you should print the name of garage which is utilized least at that time based on utilization percentage, on a new line: “Least Utilized: <Gn>”. If multiple garages have the same minimum percentage of the utilization, then you should print the one that came first in the inputs.

c. RESIZE <G> <nc>: This command resizes the capacity of garage <G> with the new capacity <nc>
    i. If the new capacity is smaller than the number of vehicles in the garage, then it should print “FAIL. TOO SMALL TO ACCOMMODATE EXISTING VEHICLES.”
    ii. Otherwise, it should print “SUCCESS” after resizing the garage
    iii. This command may require to use realloc
    iv. It is guaranteed that the garage <G> will be available in the system before processing this command

d. SEARCH_OWNER <OWNER>: This prints the license plates of the vehicles and the garages they are parked for the vehicles owned by <OWNER>. It prints the data in the following format:

    <LP1> <G1>
    <LP2> <G2>
    <LP3> NOT ON CAMPUS

   Here, LPx is the license plate of the vehicles owned by <OWNER> and Gx is the garage where that car is parked. The list should be displayed in the same order they are entered in the inputs. If a car is not parked on campus, then it should display “NOT ON CAMPUS” for that vehicle.
   If there is no registered car by this owner, then it should display “NO REGISTERED CAR BY THIS OWNER”
   [A good idea would be using the displayVehiclesByOwner() function to process this command]

e. RELOCATE <LP> <G>: This command relocates the vehicle with license plate <LP> to garage <G>. It is guaranteed that <G> will be different than the current garage of the vehicle. It prints the following message:
    i. If the target garage is not found, it should print “<G> NOT FOUND”.
    ii. If the vehicle is not in any garage, it should print “<LP> NOT IN CAMPUS”
    iii. If the target garage is full, it should print “[G] IS FULL.” and should not remove the vehicle from the current garage.
    iv. If the vehicle is successfully relocated, then print “PARKED” and in another line, it should print “RELOCATION SUCCESSFUL”
    v. A good idea would be using the parkVehicle and removeVehicleFromGarage function in this process.

f. COUNT_TOTAL: This command shows the total number of vehicles parked on the campus.

    <count>

g. REGISTER_VEHICLE <LP> <OWNER>: This command registers a new vehicle with license plate <LP> with the owner <OWNER> to the campus system. It is guaranteed that each license plate number will be unique in the input. This command may require realloc!
   The command will output the message “REGISTERED” after successfully adding the vehicle to the campus system.

h. REMOVE_VEHICLE_GARAGE <LP>: It removes the vehicle with license plage LP from the garage it is located at that time. After removing the vehicle, it prints:

    “REMOVED FROM <G>”, where <G> is the garage where the vehicle was removed from.

   If the vehicle was not found in any garage, then it prints:

    “NOT FOUND IN CAMPUS”

i. REMOVE_GARAGE <G>: It removes the garage <G> from the campus. While removing, it frees the memory and sub memory allocated to this garage. However, it does not remove the vehicles from the campus system. The vehicles are still registered in the system. Just, they are removed from the garage. After removing, it prints:

<G> REMOVED.

If the garage is not found, then it prints:

<G> NOT FOUND

j. There could be many more useful commands. But to reduce the length of the assignment, they are not added to the assignment.

C. Function Implementations:

- You are required to implement the following functions with the prototypes mentioned below and must use them while processing the commands/queries and other operations in your code:
    1. **Garage* createGarage(const char* name, int capacity):** Creates a new garage with the specified name and capacity and initializes other variables and makes parked vehicles array based on the capacity.
    2. **RegisteredVehicle* createRegisteredVehicle(const char* license, const char* owner):** Create a new vehicle with the given license plate and owner.
    3. **void registerVehicle(Campus* campus, const char* license, const char* owner):** Registers a new vehicle on campus.
    4. **void parkVehicle(Garage* garage, RegisteredVehicle* vehicle):** Parks a vehicle in a specified garage. If the garage is full, it prints "FULL". Otherwise, if the parking is successful, it prints "PARKED".
    5. **int removeVehicleFromGarage(Garage* garage, const char* license):** Removes a vehicle from a specified garage. The function returns 1 if the removal is successful. Otherwise, it returns 0.
    6. **RegisteredVehicle* searchVehicleByLicense(const Campus* campus, const char* license):** Searches for a vehicle by its license plate across all garages on campus. If the vehicle is found, it return the vehicle. Otherwise, it should return NULL.
    7. **int countTotalVehicles(const Campus* campus):** Returns the total number of vehicles parked across all garages on campus.
    8. **int resizeGarage(Garage* garage, int new_capacity):** Resizes a garage to a new capacity. If the new capacity is smaller than the current number of vehicles in that garage, then you should print a massage "FAIL. TOO SMALL TO ACCOMMODATE EXISTING VEHICLES.\n". The function returns 1 or 0 depending on whether the task was successful or not.
    9. **int relocateVehicle(Campus* campus, const char* license, const char* target_garage_name):** Relocates a vehicle from one garage to another. If the vehicle not found, it prints the message like point 7 above. If the target garage is not found, it should print "[G] NOT FOUND". If the target garage is full, it should print "[G] IS FULL." and should not remove the vehicle from the current garage. The function returns 1 or 0 depending on whether the task was successful or not.
    10. **void displayVehiclesByOwner(const Campus* campus, const char* owner_name):** Displays all vehicles owned by a specified person if they are in campus. It means, same owner can have multiple vehicles and all of them can be on campus as well.

    10. **int removeGarage(Campus* campus, const char* garage_name):** Removes a specified garage from the campus and frees the memory. Note that while removing a garage, you should not remove the remove the vehicles from the system. The function returns 1 or 0 depending on whether the task was successful or not.
    11. **void generateGarageUtilizationReport(const Campus* campus):** Generates a report on the utilization of all garages. For each garage, it should print: "Garage: [G], Capacity: [GC], Occupied: [GO], Utilization: [GU]%", where [G] is the garage name, [GC] is the capacity of the garage, [GO] is the count of vehicles in the garage, and [GU] is the percentage of fullness of the garage. After printing the above summary, you should print the name of garage which is utilized least at that time.
    12. **You must also write necessary functions to free all the memory**
    13. **You are free to add any other functions if you wish (but not required).**

D. Additional Restrictions:

a. Only one instance of each vehicle should be created. It means, there should be only one malloc for one vehicle. If you need to use this vehicle again, for example relocate a vehicle to different garage, then you should assign the already created vehicle, instead of making another copy of the vehicle for the new garage.
b. All the strings input strings the assignment can be maximum 21 characters long, comprises of upper case letters, digits, and underscores.
c. For any strings, if it is not a statically allocated array, then you must malloc this with the exact amount of space necessary and then store the string with strcpy function. Any other approach for any version of your submission will result in -100 penalty in this assignment.
d. You must free all the memory to receive full credit.
e. The leak detector you have used for your lab 2 is great for day to day testing. However, including that leak detector slows down your code and it can take pretty long if the input size is big and if you have several dynamic memory allocation on your code. So, to test your code, we will use valgrind command inside codegrade. If you want to know more about valgrind command, you can watch a video posted on the DMA module. For your own testing, you can still use memory leak detector, however, **DO NOT INCLUDE** leak_detecotor_c.h on your code for programming assignment.
f. You do not need to comment line by line, but comment every function and every "paragraph" of code.
g. You don't have to hold any particular indentation standard, but you must indent and you must do so consistently within your own code.
h. Your code must work on codegrade and the output format must match to receive credit for a particular test case

Input Specification (Input must be from standard input through scanf. Do not use file i/0. File i/o will get zero)

The first line will contain 3 positive integers, g ($1 \le g \le 10000$), representing the number of garages in the campus so far, v ($1 \le v \le 10^5$), representing the number of registered vehicle in the system so far, and c ($1 \le c \le 10^5$), the number of commands to be processed.

The following *g* lines will contain the information of the g number of garages in the campus. Each line for a garage will contain a string and an int. The string represents a garage name (a single word string of in between 1 and 21 characters, all upper-case letters, digits or underscores) and the int represents the capacity. The value of the capacity will be between 1 to 10000.

The next v lines will contain the information of the v number of registered vehicles. Each line for the vehicle contains two strings. The first string is the license plate number (single word strings of in between 1 and 21 characters each, all upper case letters, digits or underscores). The second string present the owner of the vehicle and this follows the same string specification like license plate.

The following c lines will contain c commands. The commands will follow the specification mentioned above.

Note: the count of all the vehicles in the campus won't exceed 10^6.

Output Specification (Output must be on standard console output. No file i/o is allowed. File i/o will get zero)

The output will be based on the commands processed. Each output should be in a different line.

Example Input and Output with explanation (note that you should prepare more test cases and test your code with more inputs)

Example Input	Example Output (standard output)
3 15 20 GARAGE_A 3 GARAGE_B 4 GARAGE_C 3 ABC123 AHMED XYZ789 MIRAZIZ DEF456 AISHA XYZ787 HUDSON XYZ689 WILLIAM XYZ779 MUHAMMAD XYZ589 KEVIN DEF455 ARJUN DEF756 MATTHEW DGF456 RICARDO ABC1335 JAMES ABD123 ANEESHA ABD723 ANEESHA ACC123 RYAN ADC1234 ZENA PARK DEF756 GARAGE_C PARK ABD723 GARAGE_C PARK ABC1335 GARAGE_A PARK XYZ589 GARAGE_C UTILIZATION_REPORT PARK DEF455 GARAGE_C RESIZE GARAGE_A 5 SEARCH_OWNER ANEESHA RELOCATE DEF756 GARAGE_A COUNT_TOTAL PARK ACC123 GARAGE_C RESIZE GARAGE_C 2 RESIZE GARAGE_C 5 PARK DEF455 GARAGE_C UTILIZATION_REPORT REGISTER_VEHICLE ABC777 NUSAIR PARK ABC777 GARAGE_B REMOVE_VEHICLE_GARAGE ABC1335 REMOVE_GARAGE GARAGE_C UTILIZATION_REPORT	PARKED PARKED PARKED PARKED Garage: GARAGE_A, Capacity: 3, Occupied: 1, Utilization: 33.33% Garage: GARAGE_B, Capacity: 4, Occupied: 0, Utilization: 0.00% Garage: GARAGE_C, Capacity: 3, Occupied: 3, Utilization: 100.00% Least Utilized: GARAGE_B FULL SUCCESS ABD123 NOT ON CAMPUS ABD723 GARAGE_C PARKED RELOCATION SUCCESSFUL. 4 PARKED FAIL. TOO SMALL TO ACCOMMODATE EXISTING VEHICLES. SUCCESS PARKED Garage: GARAGE_A, Capacity: 5, Occupied: 2, Utilization: 40.00% Garage: GARAGE_B, Capacity: 4, Occupied: 0, Utilization: 0.00% Garage: GARAGE_C, Capacity: 5, Occupied: 4, Utilization: 80.00% Least Utilized: GARAGE_B REGISTERED PARKED REMOVED FROM GARAGE_A GARAGE_C REMOVED Garage: GARAGE_A, Capacity: 5, Occupied: 1, Utilization: 20.00% Garage: GARAGE_B, Capacity: 4, Occupied: 1, Utilization: 25.00% Least Utilized: GARAGE_A

Deliverables

You must submit only main.c on Codegrade:

Rubric (subject to change):

According to the Syllabus, the code must work on codegrade to receive credit. If your code does not compile on codegrade, we conclude that your code has compiler error 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. Note that we will apply more test cases while grading. So, passing the sample test cases might not guarantee that your code will also pass other test cases. So, thoroughly test your code.

1. If a code does not compile the code may get 0. However, some partial credit maybe awarded. A code having compiler error cannot get more than 30% even most of the codes are correct
2. If you modify or do not use the required structure: you may get 0
3. Not using dynamic memory allocation for storing data will receive 0
4. 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%
5. Implementing required functions and other requirements: 35%
6. Freeing up memory properly with zero memory leak (if all the required malloc implemented): (15%)
7. Passing test cases: 50%

Some hints:

- **Read it completely and check how the given inputs are producing the sample output.**
- **Make sure you have a very good understanding of Dynamic memory allocation based on the lecture, recording, exercises, and labs**
- **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.**
- **Start the assignment as soon as possible**
- **Break it down by drawing and designing,**
- **Write each function and test whether your data is loaded 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.**

Some Steps (if needed) to check your output AUTOMATICALLY in a command line in repl.it or other compiler with terminal option (This is very useful to test your code, passing inputs from file and check whether your code is generating the expected outputs or not):

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.

Good Luck!