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, pointers, and arrays of pointers. The difficulty level is not actually very high, but it is **intricate** - 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. **Submit your main.c, in.txt, and the memory leak header and c file to Mimir platform.**

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 Mimir. 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: The Monster Trainers Need Your Help

Several small monster trainers have come to you for advice regarding expeditions they're planning into various regions. You are writing a program to estimate how many monsters they can expect to capture in each region.

- You've got a Small Monster Index that tells you the name, type, and relative commonality of all the small monsters in question.
    - (A monster's absolute commonality is the same in each region. A monster's *relative* commonality will change region by region as calculations are performed – we'll show you how that works shortly.)
- You've also got an atlas that tells you about the relevant regions and which small monsters are present in them.
- Each trainer tells you which regions they're visiting, and how many monsters they intend to capture per region.
- To estimate the number of a given monster M, a trainer will capture in a region R:
    - Divide the relative population of M in R by R's total relative population.
    - Multiply the result by the total number of captures the trainer intends per region.
    - Round this result to the nearest integer. .5 rounds up, so you can use round() and its friends. Note that this can result in a total slightly different than the trainer intended!

Data Structures

The structures you'll use for the monsters, regions, itineraries and trainers are shown in the sidebar. ***You must use these structures in your code. However, you are free to add more structures if you need. (Some of my unit test case depends on this structure and members and you should not change the structure name and member name)***

You'll need to allocate, read, compute upon, output from, and subsequently free:

- The monster index.
    - The names and elements of each monster in the monster index.
- The region atlas.
    - The names and monster lists of each region.
- A list of trainers.
    - The names and itineraries of each trainer.
    - The region list of each itinerary.

```
typedef struct monster {
	char *name;
	char *element;
	int population;
} monster;

typedef struct region {
	char *name;
	int nmonsters;
	int total_population;
	monster **monsters;
} region;

typedef struct itinerary {
	int nregions;
	region **regions;
	int captures;
} itinerary;

typedef struct trainer {
	char *name;
	itinerary *visits;
} trainer;
```

Input Specification

The input will be taken from in.txt file. There are blank lines in the sample inputs to make them more readable. They may or may not be present in the actual inputs; you should completely ignore blank lines. If you use fscanf, those blank lines will be automatically ignored. You can just write your code thinking that the line gaps do not exist.

The first line of the file contains the number of monsters *mcount* and the word monsters just as an indication that this number is count of monsters. You can read that string and ignore it.

The next *mcount* lines contain information of monsters where each line contains two strings and one integer. The first string is the monster name (single word string with maximum length is 50). The second string is the element of the monster (single word string with maximum length is 50). The integer represents the population of the monster (max value is 1 million)

After that the file contains the number of regions *rcount* and the word regions. You can read the word regions and ignore it.

After that the file contains information about *rcount* number of regions. Where the first line of a region contains the name of the region, next line contains the number of different monsters *rmcount* in that region with the word monsters. The next rmcount lines contain the name of different monsters in the region.

After the *rcount* number of regions, the file contains number of trainers *tcount* and the word Trainers. Then tcount number of trainers' information is provided. For each trainer the first line represents the trainer name (single word string with maximum length is 50), the next line contains how many captures with the word captures and then then next line contains number regions *trcount* with the word regions. After that the *trcount* lines contain the name of the region the trainer is visiting.

Output Specification

The output of the program must be written to out.txt file with exact same format as specified in the sample output. The output also need to be displayed in the console. The output contains each trainer name with the number of monster the trainer will capture from each region. Follow the output format as shown in the sample output. Note that we will test your code with diff command to match your output file with our output file. If they do not exactly match, you will loss significant grade for that specific test case. But there maybe some partial credit depending on how much it is not matching.

Print order should generally be consistent with input:

- Print the trainers in the order you got them.
- Within the trainers, print the regions in the order you got the visits.
- Within the regions, print the monster counts in the order they show up in the atlas for that region.
- Print blank lines between each trainer.

Example Input and Output

Example Input

8 monsters
StAugustine Grass 12
Zoysia Grass 8
WholeWheat Bread 6
MultiGrain Bread 10
Rye Bread 10
Cinnamon Spice 5
Pepper Spice 10
Pumpkin Spice 30

3 regions

Rome
4 monsters
StAugustine
Zoysia
WholeWheat
Pepper

Helve
5 monsters
StAugustine
WholeWheat
MultiGrain
Rye
Cinnamon

Aria
5 monsters
Zoysia
MultiGrain
Cinnamon
Pepper
Pumpkin

3 Trainers

Alice
5 captures
2 regions
Rome
Aria

Bob
4 captures
3 regions
Rome
Helve
Aria

Carol
10 captures
1 region
Aria

Example Output (*see the mapping example discussed in the next page to understand how this output is generated*)

Alice
Rome
2 StAugustine
1 Zoysia
1 WholeWheat
1 Pepper
Aria
1 Zoysia
1 MultiGrain
1 Pepper
2 Pumpkin

Bob
Rome
1 StAugustine
1 Zoysia
1 WholeWheat
1 Pepper
Helve
1 StAugustine
1 WholeWheat
1 MultiGrain
1 Rye
Aria
1 Zoysia
1 MultiGrain
1 Pepper
2 Pumpkin

Carol
Aria
1 Zoysia
2 MultiGrain
1 Cinnamon
2 Pepper
5 Pumpkin

Mapping Example

Here's the table of how each individual trainer's results are computed. It also shows how rounding issues can lead to trainers capturing more monsters than they intend!

Rome	*Raw*	*Divided*	*Alice*	*Round*	*Bob*	*Round*
Coefficient	1.00	36.00	5.00		4.00	
StAugustine	12.00	0.33	1.67	2.00	1.33	1.00
Zoysia	8.00	0.22	1.11	1.00	0.89	1.00
WholeWheat	6.00	0.17	0.83	1.00	0.67	1.00
Pepper	10.00	0.28	1.39	1.00	1.11	1.00
Total	36.00	1.00	5.00	5.00	4.00	4.00

Helve	*Raw*	*Divided*		Bob	*Round*
Coefficient	1.00	43.00		4.00	
StAugustine	12.00	0.28		1.12	1.00
WholeWheat	6.00	0.14		0.56	1.00
MultiGrain	10.00	0.23		0.93	1.00
Rye	10.00	0.23		0.93	1.00
Cinnamon	5.00	0.12		0.47	0.00
Total	43.00	1.00		4.00	4.00

Aria	*Raw*	*Divided*	*Alice*	*Round*	*Bob*	*Round*	*Carol*	*Round*
Coefficient	1.00	63.00	5.00		4.00		10.00	
Zoysia	8.00	0.13	0.63	1.00	0.51	1.00	1.27	1.00
MultiGrain	10.00	0.16	0.79	1.00	0.63	1.00	1.59	2.00
Cinnamon	5.00	0.08	0.40	0.00	0.32	0.00	0.79	1.00
Pepper	10.00	0.16	0.79	1.00	0.63	1.00	1.59	2.00
Pumpkin	30.00	0.48	2.38	2.00	1.90	2.00	4.76	5.00
Total	63.00	1.00	5.00	5.00	4.00	5.00	10.00	11.00

Specific Requirements

- You have to use dynamic memory allocation
- You have to use the provided structures without changing their name
- You may not use global variables
- You have to implement at least the following functions:
    - `monster* createMonster(char *name, char *element, int population):` This function returns a dynamically allocated monster filled with the provided parameters
    - `monster** readMonsters(FILE* infile, int *monsterCount):` This function returns an array of monster pointers where each monster pointer points to the dynamically allocated monsters with fill-up information from the provided input file. It can use the createMonster function in this process. This function also updates the passed variable reference pointed by monsterCount.

    - `region** readRegions(FILE* infile, int *countRegions, monster** monsterList, int monsterCount):` This function returns an array of region pointers where each region pointer points to a dynamically allocated region, filled up with the information from the file, and the region's monsters member points to an appropriate list of monsters from the monsterList passed to this function. This function also

updates the passed variable reference pointed by countRegions. As the readMonsters function has crated all the monsters using dynamic memory allocation, you are getting this feature to use/re-use those monsters in this process.

- *trainer* readTrainers(FILE* infile, int *trainerCount, region** regionList, int countRegions):* This function returns a dynamically allocated array of trainers, filled up with the information from the file, and the trainer's visits filed points to a dynamically allocated itinerary which is filled based on the passed regionList. This function also updates the passed variable reference pointed by trainerCount. As the readRegions function has crated all the regions using dynamic memory allocation, you are getting this feature to use/re-use those regions in this process.

- *void process_inputs(monster** monsterList, int monsterCount, region** regionList, int regionCount, trainer* trainerList, int trainerCount):* This function processes all the data and produce the output. During this process, you can create/use more functions if you want.

- `void release_memory(monster** monsterList, int monsterCount, region** regionList, int regionCount, trainer* trainerList, int trainerCount ):` This function takes all the dynamically allocated arrays and free-up all the memory. You can create/use more function in this process if you want.

- You have to use memory leak detector code as shown the lab as well as explained in webcourses
  - You must #include "leak_detector_c.h" in your code, and
  - You must call atexit(report_mem_leak) as the first line of your main().
  - leak_detector_c.h and leak_detector_c.c are available in webcourse.
- You do not need to comment line by line, but comment every function and every "paragraph" of code.
- You don't have to hold any particular indentation standard, but you must indent and you must do so consistently within your own code.

Rubric (subject to change):

According to the Syllabus, the code will be compiled and tested in Mimir Platform for grading. If your code does not compile in Mimir, 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. A code having compiler error cannot get more than 50% even most of the codes are correct
2. If you modify or do not use the required structure: 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: 30%
6. Freeing up memory properly with zero memory leak (if all the required malloc implemented): (20%)
7. Passing test cases: 50%

Some hints:

- **Make sure you have a very good understanding of Dynamic memory allocation based on the lecture, exercises, and labs**
- **The core concepts of the example of dynamically allocating array of structure pointer, dynamically allocating array of strings, and the Lab1 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 load 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. [If you start early and take help early, you have a good chance to join us during office hours easily. Otherwise, if you work on the last moment, you might not be able to enter the office hours for long queue. So, plan accordingly]**

Good Luck!