CMPT 214: Programming Principles and Practice

Final Examination (Take-Home)

Due December 11, 2020 by 11:59pm, **Absolutely No Extensions**

This exam consists of 12 pages containing a total of 3 questions worth a total of 52 marks.

Instructions

Exam Rules

This is a take-home exam. **Here's what this means for the purposes of this exam:**

- The exam is an **individual effort**.
- You may not discuss **any** details of this exam with anyone else, whether they are registered in the class or not.
- You may consult any official CMPT 214 class materials, such as the textbook, exercises, exercise solutions, assignments, assignment solutions.
- You may use a calculator, you may use a dictionary.
- You may use a C compiler, and a BASH shell (either on your own computer or Tuxworld).
- You may ask questions of the course instructors (see below).
- You may **not** consult any other resources beyond those mentioned above, including any Internet resources, or other people that are not course instructors, on any matter related to the exam.

  Exceptions: you may use the internet to gain access to a C compiler (e.g. `tuxworld`, or online compiler) or the LINUX BASH shell on `tuxworld` for the purpose of compiling and testing your code, and you may use an online dictionary for the purpose of looking up the meaning of a word.

Submitting Your Exam

The completed exam is to be submitted to Canvas. The submission page is in the same place that assignment submissions are found. Follow the **Assignments** link on the left menu, and click on the **Mid-term Exam** assignment. Submit your exam in the same way you have submitted your previous programming assignments.

- As always, only `.zip` archives containing your submitted files will be accepted (no .rar, no .7zip, etc.).
- Multiple submissions are permitted until the deadline, but only the most recent submission will be graded. **Download your submissions after uploading to double check it contains everything needed. Leave yourself time to do this!**
- You are solely responsible for ensuring that all files are correctly submitted.
- The gold standard for determining program correctness is whether or not is runs/works/crashes on the `tuxworld.usask.ca` LINUX server. Test your submissions on tuxworld. C Programs must be compliant with the C11 standard.
- At the end of this document is a **Hand-in Checklist** that tells you exactly what to submit.

Asking Questions

- Do not post questions about the exam in any public place. This includes Canvas discussions and Discord.
- Again, treat the asking of questions as if you are in an invigilated exam room and can only privately ask an instructor.
- If you have a question about the wording of a question, or clarification about what a question is asking you to do, send a single email addressed to both course instructors:
  - `eramian@cs.usask.ca`
  - `peggy.anderson@usask.ca`

  Your question will be answered by the next available instructor.
- **We will answer questions from the time the exam is released until 6pm on Friday, December 11** (about 6 hours before the deadline). Start early, so you can ask questions early. **Do not expect replies after 6pm, Friday, Dec. 11.**

Academic Honesty

Breaching the above rules of the take-home exam is **academic misconduct**. Any evidence of academic misconduct on this exam will be pursued to the fullest extent possible through formal academic misconduct hearings. This means that the Office of the Dean will be informed immediately, no informal resolution option will be given, and you will need to appear before the hearing board. You do not want to be before the hearing board.

This exam is an **individual undertaking** – cheating on an exam is considered serious academic misconduct by the university and can be met with disciplinary action, including suspension or expulsion. By submitting your exam solutions to Canvas, you are affirming that the submitted work is entirely your own.

Grading Rubric

You may view the grading rubric on the exam's submission page on Canvas.

Hint: If you get a C compiler warning when `-Wall` *and* `-Wextra` *are turned on, this is a strong sign that there is an error in your program, despite the fact that the compiler is able to compile it. Warnings mean that the compiler thinks you might have done something you didn't intend, but that the code it encountered is syntactically and semantically valid, and so it is able to compile the program anyway. Programs that compile with warnings are likely to lose marks.*

Exam Questions

Question 1 (20 points):

It is a time of celebration in the Candy Kingdom. Lucrative rock candy mines have been discovered not far away. Rock candy is the primary ingredient in producing building materials for the kingdom, and can be refined into purified candium, a deliciously sweet raw ingredient for the Candy Kingdom's primary export industry: gourmet artisan candy products.

Princess Bubblegum has ordered the creation of a mining camp to create a place to live for the miners who will soon be exploiting the newly discovered mines. But where to put it? The camp must be close to one of the mine sites, and it must be close to the river because the corn syrup that flows in the river is needed both in the mining process, and to provide life-sustaining high-fructose sustenance for the miners. The camp also needs to be as close as possible to the factory that refines the raw rock candy ore to minimize transportation costs.

In this scenario, you play the role of Peppermint Butler, who has been tasked with analyzing geosurvey data to calculate the most optimal location for the mining camp that simultaneously minimizes the distances between the camp, the nearest mine, the river, and the factory.

The geosurvey data has been already been collected. The Candy Kingdom Lands have been divided into an N by M grid. For each grid location, the distance to the nearest mine, the distance to the river, and the distance to the factory has been determined. This information has been stored in a data file. All you need to do now is write a program that will compute the optimal location for the mining camp. If (r, c) is the grid location at row r, column c (top left corner is $(0, 0)$), then the optimal location for the mining camp is the (r, c) that minimizes the following *suitability function*:

$$S(r, c) = R + M + 1.5 \times F$$

$S(r, c)$ is the suitability score for grid location (r, c) (smaller is better), R is the shortest distance from grid location (r, c) to the river, M is the shortest distance from grid location (r, c) to the nearest mine, and F is the distance from grid location (r, c) to the factory. The best location for the mining camp is the grid location (r, c) that minimizes $S(r, c)$.

Input File Format (Geosurvey Data)

The geosurvey data file is a plain text file in the following format. The data file begins with two integers, N and M, where N is the number of rows in the geosurvey grid, and M is the number of columns in the geosurvey grid.

Following this are $N \times M \times 3$ integers. The first three integers are the values of R, F, and M for grid location $(0, 0)$, in that order. The next three integers are the values of R, F, and M for grid location $(0, 1)$. The next three numbers are the same data for grid location $(0, 2)$, and so on, in row-major order. That is, after the data for grid location $(0, M-1)$, the next three numbers are the data for grid location $(1, 0)$.

There may be an arbitrary amount and type of whitespace (newlines, spaces, tabs) between any of the integers in the data file.

Hint: if you plan well, and choose a good method for reading this data from the file, the arbitrary whitespace should not cause any problems or special cases that have to be handled.

Two geosurvey files datafiles are provided. `candy-geosurvey-small.txt` contains test data to be used for testing your program. Later in this document we provide the expected output when using this input file. We also provide `candy-geosurvey.txt` which is the actual Candy Kingdom geosurvey data that you need to solve the problem for. Your program must work with any input file that conforms to the stated input file format.

Your Tasks

In a file called `question1.c`, do the following:

1. Write a function called `read_geosurvey()` that takes the name of a geosurvey data file as a parameter. This is the only parameter permitted. This function must read the geosurvey data into a data structure of your design, and return a pointer to a newly-allocated instance of that data structure containing the geosurvey data from the file.
2. Write a function called `camp_location()`. This function may have any parameters you deem necessary. This function should determine from the geosurvey data the grid location that has the optimal suitability measure $S(r,c)$ and communicate that grid location back to the calling function (you may not use global variables). You may design additional data structures as needed to support this function.
3. Write a function called `print_survey_map()`. This function **must** take a filename as a parameter but may have any other parameters you deem necessary. This function should write to the provided filename a visualization of the geosurvey. For each grid location, you'll write a `*` if is the optimal location for the mining camp, an `M` if it is the location of a mine, an `F` if it is the location of a factory, an `=` if the river runs through the grid location, and `.` if the location contains none of those things. A mine and a factory will never occupy the same location. However, a mine or a factory may appear in a river location, and printing those map symbols should be given priority over printing a river symbol. The best camp location could appear anywhere, even on a grid location with river, factory, or mine, and its map symbol should be given priority over any other symbol.

   Hint: There's an easy way to determine from the data whether a location contains a mine, river, or factory. But you have to figure it out. We won't tell you. It's pretty easy to reason out if you think about it.
4. Write a function called `deallocate_geosurvey()` that takes a pointer to a geosurvey data structure created by `read_geosurvey()` as a parameter (and no other parameters) and deallocates any and all dynamically-allocated memory associated with it.
5. Write a `main()` function that takes two command line arguments: the name of the input file to read the geosurvey data from, and the name of an output file to write the geosurvey map visualization to. Use the functions you wrote in parts (a) through (d) to read the input file, determine the best location for the camp, write out the visualization of the geosurvey, display a message to the console indicating the grid location that is the most suitable location for the mining camp, and then de-allocate the data structure. Sample output is given on the next page for `candy-geosurvey-small.txt`.

Sample Output for Question 1

```
# For the input file candy-geosurvey-small.txt, the console output
# from the program should be something like:

The best location for the mining camp is grid coordinate: 8, 12.

# and the contents of the output file should be:
................
................
................
..............F.
................
................
................
................
============*===
................
................
................
...........M....
................
................
................
```

Remember: your program also has to work for `candy-geosurvey.txt` and any other input file that satisfies the required file format.

Question 2 (20 points):

Dipper and Mabel's favourite game is Dungeons, Dungeons and More Dungeons. They'd really like to write a computer program to automate the task of creating random encounters for this game. A random encounter is a randomly generated list of monsters that the players have to defeat. The program will create random encounters by generating a list of monsters that together are an appropriate challenge for the game's players. Help out Dipper and Mabel!

Read all requirements for this question before writing any code. If you do not, you might make decisions early on that make later tasks harder. When designing data structures, you must think about the problem as a whole and how your data structures can best support all of the things it is needed for.

Program Overview

The program will consist of four modules:

statsdb: This module will manage *creature databases* of game creatures and their statistics. A *creature* could be a player or a monster.

encounter: The encounter module's main purpose will be to provide a function to generate a random encounter (a list of monsters) of a suitable challenge for a given creature database containing players by selecting monsters randomly from a given creature database containing monsters.

question2: This module will contain the main program. It will call on the other modules to load monster and player databases from data files, and generate random encounters.

randomindex: This module is given to you already written. It is the same module you received on the midterm. To remind you, its only purpose is to provide a function with the prototype:

```
        unsigned int randomindex(unsigned int n);
```

which generates a random number between 0 and $n - 1$. **Do not modify this module in any way.**

Input File Format (Creature Statistics Data)

The input data for this program are plain text files containing creature statistics that will be used to create the player and monster databases. The first line of each file contains a positive integer indicating the number of creatures, K, in the database. Each of the following K lines contain game statistics for one creature. Each line contains five pieces of information, separated by one or more spaces:

- The name of the creature, which contains no spaces and is less than 100 characters long.
- An integer indicating the creature's initiative rating.
- A non-negative integer indicating the creature's armour rating.
- A non-negative integer indicating the creature's attack rating.
- A non-negative integer indicating the creature's challenge rating.

Your Tasks

1. Write the `statsdb` module. This module must provide the following functions:

(a) A function called `create_stats_db()` which takes the name of a file containing creature statistics data (file format described above) as a parameter, and returns a pointer to a *creature database*: this is a data structure of your design that stores the information contained within the file. You may add additional parameters to this function if you think it is appropriate.

(b) A function called `destroy_stats_db()` that takes a pointer to a creature database as a parameter and de-allocates any and all dynamically allocated memory associated with it. You may add additional parameters to this function if you think it is appropriate.

(c) A function called `get_creature()` which, given a pointer to a creature database and a creature name as parameters, returns the creature statistics for the named creature. The return type should be a data structure of your design that holds the statistics for a creature. You may add additional parameters to this function if you think it is appropriate.

(d) A function called `average_challenge()` which, given a pointer to a creature database as a parameter, returns the average challenge rating of the creatures in the database. You may add additional parameters to this function if you think it is appropriate.

(e) A function called `print_creature_stats()` which, given a pointer returned by `get_creature()`, prints out to the console the statistics of the given creature in an aesthetically pleasing manner. The stats for a creature consist of all five of the data items that appear on a line of the input data file for a creature (described above). All the stats for the creature must be printed on a single line. You may add additional parameters to this function if you think it is appropriate.

Hint: This seems like a lot, but if your data structures are well-designed, most of these functions are quite short.

2. Write the `encounter` module. This module must provide the following functions:

(a) A function called `generate_random_encounter()`. The parameters to this function must be:

- A pointer to a creature database containing monsters.
- A descriptive name for the encounter (string).
- A positive integer indicating *c* the maximum number of creatures to add to the encounter.
- A positive integer called the *challenge rating threshold*.

Other parameters may be added if you think it is appropriate. The function must return a pointer to a data structure of your design called an *encounter* that contains the following:

- The descriptive name of the encounter.
- The number of creatures selected for the encounter.
- The names of each creature in the encounter. A creature name may appear more than once. For example if "goblin" appears three times, it means there are three goblins in the encounter.

How to construct the encounter: The function should randomly select monsters from the monster database and add their names to the encounter one-by-one until the encounter contains exactly *c* monsters, or until the most recently added monster causes the total challenge rating of the monsters in the encounter to exceed the given *challenge threshold*, whichever occurs first.

(b) A function called `print_encounter()` which, given a pointer to an encounter, prints the descriptive name of the encounter followed by a numbered list of monsters in the encounter and their statistics in an aesthetically pleasing manner (see the sample output, below). After printing all of the monsters' statistics, print the total challenge rating of the encounter (again, see sample output, below). You may add additional parameters to this function if you think it is appropriate.

3. Write the `question2.c` module. This module should contain the main program. The main program must accept two command line arguments. The first argument must be the name of an input file containing the creature stats of monsters. The second argument must be the name of an input file containing the creature stats of players. Sample input files, `monsters.db` and `players.db` have been provided for you.

   The main program must do the following:

   (a) Create creature databases from the given monster and player input file names.

   (b) Generate a random encounter of monsters with a challenge threshold equal to the **twice** the average challenge rating of the players. Give the encounter any descriptive name you like. Be creative!

   (c) Print the monsters in the random encounter to the console.

   (d) Destroy all data structures that were created.

4. Write a `Makefile` that will build your program. Consider building up the Makefile as you add modules, so that you can always build your program just by typing `make`.

Sample Output

```
Dark Cave Encounter:
Monster #1: Mind_Flayer, Init: 1, Armor: 15, Attack: 7, Challenge Rating: 7
Monster #2: Frost_Giant, Init: -1, Armor: 15, Attack: 9, Challenge Rating: 8
Monster #3: Mind_Flayer, Init: 1, Armor: 15, Attack: 7, Challenge Rating: 7
Monster #4: Skeleton, Init: 2, Armor: 13, Attack: 4, Challenge Rating: 1
Monster #5: Goblin, Init: 2, Armor: 15, Attack: 4, Challenge Rating: 1
Monster #6: Orc, Init: 1, Armor: 13, Attack: 5, Challenge Rating: 1
Monster #7: Giant_Rat, Init: 2, Armor: 12, Attack: 4, Challenge Rating: 1
Monster #8: Wyvern, Init: 0, Armor: 13, Attack: 7, Challenge Rating: 6
Total challenge rating: 32
```

Remember, encounters are randomly generated, so the likelihood of you observing this exact output is almost zero.

Question 3 (12 points):

After talking to your friend, Kimmy, about a script you made to organize your music collection, she expresses that she's been feeling as if she has been living under a rock for the last 15 years. She wants to expand her music repertoire and has no idea who to start with. Or *where* to start - there are so many genres to pick from! This is where you come in, given a directory/file structure representing a database of information about popular music songs categorized by genre, you will write a script that Kimmy can use that allows her to query this database based on a genre she selects. Once Kimmy has selected a genre, your script will output a text file with all artists under that genre, and output to her how many artists there are in the database for the given music genre. This will give her a good starting point on what some genres are out there, and who some artists are that fall under that genre.

Getting Started

To solve this problem you will write a BASH script in a file called `question3.sh`. To prepare, do the following:

- Create an empty directory where you will complete the problem.
- Within that empty directory, create your `question3.sh` file. It can be empty for now.
- Place the provided `music.zip` file in the same directory as `question3.sh`.
- Unzip `music.zip`. This should create a sub-directory called `music`. Within this a directory structure which we will call the *music database*.

After completing these steps you should see the following in the directory in which you placed `question3.sh` when you get a directory listing:

```
$ ls
music        music.zip   question3.sh
```

Thus, the only files in your working directory for this question should be `question3.sh`, `music.zip` and the `music` directory containing the *music database* directory structure.

Structure of the Music Database

The directory/file structure of the `music` directory created by unzipping `music.zip` is very similar, but not identical to, the directory/file structure you created in Assignment 7.

Within the `music` directory are directories named after music genres. Within most of the genre directories are directories named after artists, and then within each artist directory are `.txt` files with information about songs by that artist.

There are seven genre directories that are a little special: `hip-hop`, `metal`, `pop`, `punk`, `rap`, `rock`, and `indie`. These genre directories contain both artist directories (containing `.txt` files) and sub-genre directories (containing artist directories). You can distinguish between artist and sub-genre directories by the fact that a sub-genre directory name will always contain the main genre name as a sub-string. For example, `music/hip-hop/alternative-hip-hop` is a sub-genre because `alternative-hip-hop` contains the substring `hip-hop` (its main genre), but `music/hip-hop/eminem` is an artist directory, because `eminem` does not contain the main genre name `hip-hop` as a substring.

Familiarize yourself with the music database directory structure by exploring it a bit. Make sure to examine one of the seven special sub-genred genres and that you understand how they differ from the other genre directories.

Your Tasks

Write the BASH shell script `question3.sh` so that it does the following:

1. Change the current working directory to the the `music` subdirectory that contains the music database (assume that when the script starts, that the current working directory is the same directory that contains the `question3.sh` script file).
2. Display on the console the names of all the genre directories within the `music` directory (now the current directory) **in alphabetical order**.
3. Prompt the user to enter the name of one of the displayed genres. You may assume that the user always enters a valid, existing genre, but you must accept case-insensitive genre names, interpreting user responses as if they had been typed in all lower-case. For example, if the user enters "Pop", or "POP", this should be interpreted as the all-lowercase "pop".
4. If the user entered one of the seven special genre names that contain sub-genres, display **in alphabetical order** all the names of all of the sub-genre directores within the directory for the user's selected genre **as well as the main genre name** (exclude artist folder names!). Then prompt them to enter one of those (sub-)genre names.
5. Create a **new** file in the same directory as `question3.sh` (i.e. in the directory `../` since the current directory is `music`) that contains the names of all of the artist directories (and only artist directories) in the selected genre or sub-genre directory **sorted in alphabetical order**. If `<genre>` is the name of the selected genre or sub-genre, the name of the file written to should be `<genre>_artists.txt`. **If the output file already exists, overwrite it.**

   For example, if the user selected the genre `broadway`, then the name of the output file should be `broadway_artists.txt` and it should contain the names of all the artist folders within `music/broadway`.

   If the user selected the `dance-pop` sub-genre of `pop`, then the output file name should be `dance-pop_artists.txt` and it should contain all the artist folder names within `music/pop/dance-pop`.

   If the user selected `hip-hop` as the main genre, and then `hip-hop` again as the sub-genre (i.e. entered the main genre again), then the output file should be `hip-hop_artists.txt` and contain the names of the artist directories within the `music/hip-hop` directories.
6. Determine how many artists were in the selected (sub-)genre. Display a message indicating how many artists were found in the selected (sub-)genre.

Hints

- *You can assume that within a genre or sub-genre directory containing artist directories, that each artist directory has a unique name — no artists are duplicated within a genre or sub-genre directory.*
- *You don't need to do anything with the .txt files. You only need to look at directory names to solve this problem.*
- *You don't need to modify the files and directories within the music database in any way.*

Sample Output

This is a sample of how the output might look. Formatting is flexible, as long as the right information appears, and is presented in an aesthetically pleasing manner. For example, the list of genres doesn't need to be in two columns (this is not hard to achieve).

```
(base) Marks-MacBook-Pro:musicrecommender mark$ bash ./question3.sh
a-cappella                      mash-up
alternative-dance               metal
alternative-r&b                 neo-progressive
british-soul                    norwegian-progressive
broadway                        otacore
canadian-contemporary-r&b       pop
canadian-folk                   post-screamo
comic                           progressive-bluegrass
djent                           progressive-post-hardcore
edm                             punk
electro-jazz                    rap
electro-swing                   reggae-fusion
electronica                     rock
hip-hop                         uncategorized
indie                           video-game-music
---------------------------------------------
Select a genre from the above: indie
---------------------------------------------
canadian-indie
indie
indie-anthem-folk
indie-cafe-pop
indie-emo
indie-folk
indie-pop
indie-pop-rap
indie-poptimism
indie-punk
indie-rock
indie-soul
indietronica
seattle-indie
tulsa-indie
---------------------------------------------
Select a sub-genre from above: indie-emo
---------------------------------------------
The total number of artists in the genre indie/indie-emo were:        2
```

Hand-in Checklist

Your complete exam submission must consist of a **.zip** archive containing all of the following.

- Question 1:
  - ☐ `question1.c`
  - ☐ `candy-geosurvey.txt` (unmodified, as given)
  - ☐ `candy-geosurvey-small.txt` (unmodified, as given)
- Question 2:
  - ☐ `question2.c` (the main module)
  - ☐ `statsdb.c` and `statsdb.h` (the creature database module)
  - ☐ `encounter.c` and `encounter.h` (the encounter module)
  - ☐ `randomindex.c` and `randomindex.h` (the provided random number generator module)
  - ☐ `monsters.db` (unmodified, as given)
  - ☐ `players.db` (unmodified, as given)
  - ☐ `Makefile` (the Makefile you wrote to build your program)
- Question 3:
  - ☐ `question3.sh`