CMPT 214: Programming Principles and Practice

Midterm Examination (Take-Home)

Due November 7, 2020 by 11:59pm, **Absolutely No Extensions**

This exam consists of 7 pages containing a total of 2 questions worth a total of 40 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.
- 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) 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.
- **tl;dr:** you may consult official CMPT course materials, but otherwise you must treat this exam like you are in an invigilated exam room, discussing the exam with nobody else, and not consulting internet resources.

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.
- 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 Saturday, November 7** (about 6 hours before the deadline). Start early, so you can ask questions early. **Do not expect replies after 6pm, Saturday, Nov. 7.**

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

The grading rubric is the same for all questions on this exam. You may view the grading rubric on the exam's submission page on Canvas.

Hint: If you get a 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):

The dread pirate, Tractor Jack, is obsessed with loot. He is known for looting wheat, barley, and several other grains all along the banks of the Saskatchewan River (allegedly — allegations have not been proven in court). In his downtime, Jack loves video games, and wants to try writing his own. For practice, he wants to write some code to generate some random loot, but he wants the loot to be different from the usual boring grains he loots when he is working. The loot for his game will be that of a fantasy role-playing game. Since pirates always make their crew do all the work, he's demanded that you do his practice for him.

Jack has already gotten a start on writing his loot generator. Let's have a look at what Jack has provided us.

Provided file: `items.h`

At the top of `items.h` you'll see the definition of a structure that is given the type name `Loot`. A `Loot` item holds information about one randomly generated loot item. The `Loot` structure has three members:

`base_type_name`: A pointer to the first element of a character array that holds the string that is name of the *base type* of the loot item.

`name`: A pointer to the first element of a character array that holds the string that is name of the loot item. The name of an loot item is generated randomly from three components: a prefix, the item's base type name, and a suffix. These three components are all strings. A randomly chosen prefix, a randomly chosen base type name, and a randomly chosen suffix, are concatenated together into a single string to form the item's name.

`rarity`: An unsigned integer with value between 0 and 3 (inclusive) that indicates the rarity of the item with 0 being the most common and 3 being the least common.

The remainder of the `items.h` consists of static declarations of four pointer arrays, where each element of each array is a pointer to a string. Also defined along with the pointer arrays are macros that give the length of each pointer array. The arrays are:

`prefixes`: This pointer array stores strings that are potential item prefixes. The prefix component of the name of a randomly generated loot item is chosen from among the strings referenced in this array.

`base_types`: This pointer array stores strings that are potential base type names for randomly generated loot items. A base type name component of the name for a randomly generated loot item is chosen from among the strings referenced in this array.

`suffixes`: This pointer array stores strings that are potential suffixes for randomly generated loot items. The suffix component of the name for a randomly generated loot item is chosen from among the strings referenced in this array.

`rarity`: This pointer array stores strings that are names of the rarity levels 0 through 3. If k is a rarity level between 0 and 3, then the name of that rarity level may be obtained by writing `rarity[k]`.

Provided module: `randomindex.h`, `randomindex.c`

The `randomindex` module provides you with random numbers for selecting random item name prefixes, base type names, and suffixes. `randomindex.h` contains a prototype for one function that is implemented in `randomindex.c`. It is not necessary for you to understand how the functions `randomindex.c` work, in fact you should never need to look at `randomindex.c`, though you will need to compile it into your program.

All you need to do is look at `randomindex.h` and see how to **call** the function prototyped there. The function provided by the `randomindex` module is called `random_index()`. It's prototype is:

```
unsigned int random_index(unsigned int n);
```

It takes an unsigned integer n as an argument, and returns a random number between 0 and $n - 1$.

You can use the `random_index()` function to obtain a random index into the `prefixes`, `base_types` and `suffixes` arrays to obtain random name components for randomly generated `Loot` items.

Your Tasks

Complete the following tasks. **In completing these tasks you may not modify `randomindex.c` or `randomindex.h` at all. You may not modify any of the existing code in `items.h` but you may add additional code to `items.h`.**

(a) Create a new file, `items.c` to go along with the provided `items.h` to form a program module.

(b) In `items.c` write a function called `create_random_loot_list` that takes as a parameter a positive integer N and returns a pointer to the first element of a new dynamically allocated array of N randomly generated `Loot` structures. The `name` of each `Loot` structure must be randomly generated by selecting a random prefix, base type name, and suffix, and concatenating them with appropriate spacing. The `base_type_name` member of each `Loot` structure should be the same base type name randomly chosen to form part of the item's `name` member. The rarity of each `Loot` item should be a randomly generated number between 0 and 3 (inclusive).

(c) In `items.c` write a function called `print_loot()` that takes as a parameter a pointer to a structure of type `Loot`. This function should print a description of the item to the console in the following format:

```
<name>, <rarity_name> <base_type_name>
```

Where `<name>` is the `name` member of the `Loot` item, `<rarity_name>` is the name of the rarity level corresponding to the `rarity` member of the `Loot` item (you can look up the rarity name in the `rarity` pointer array in `items.h`), and `<base_type_name>` is the `base_type_name` member of the `Loot` item. This function should return nothing.

As an example, if the name of the item is `Firey Axe of Danger Sense`, the base type name of the item is `Axe` and the rarity of the item is 2, then `print_loot` should print:

```
Firey Axe of Danger Sense, Epic Axe
```

(d) In `items.c` write a function called `destroy_random_loot` that takes as a parameter a pointer to the first element of a dynamically allocated array of `Loot` structures (i.e. a pointer returned by `create_random_loot_list`) and frees **all** of the dynamically allocated memory used by the array and its component elements. This function returns nothing.

(e) Create a new file called `question1.c`. In this file, write a `main()` function that uses the functions in the `items` module to do the following:

- creates an array of five random `Loot` items;
- prints the descriptions of each of the randomly generated loot items;
- destroys (de-allocates) the array of randomly generated loot item.

Test your `main()` program compiling it together with the `items` and `randomindex` modules.

Question 2 (20 points):

Shadow Moon is recruited by Wednesday to be his body guard, errand boy, and driver while they travel around the United States of America. The purpose? It is all hush hush really. One of the rules of the job Wednesday told Shadow was "no questions". Despite the secrecy, this job did allow Shadow to visit many cities by accompanying Wednesday around the country. In his notebook, Wednesday took note of which cities he could directly fly to from a given city in case he and Shadow Moon had to disappear quickly. Wednesday's handwriting is terrible, so Shadow Moon thought storing the travel log digitally would be easier to read.

Shadow Moon wants to be able to query a city name and display the cities he can directly fly to from a given city. Unfortunately, Shadow has no idea where to start with this and instead is hoping to recruit you to help him by throwing his travel log at you!

Input File Format:

The first line of the input file contains two positive integers separated by a space:

- the first integer is N, the total number of entries (pairs of cities) in the travel log;
- the second integer, K, is the number of unique city names in the travel log.

Following the first line, the next K lines of the travel log each contain the name of one of the K unique cities names:

```
sourcecity1
sourcecity2
sourcecity3
...
sourcecityK
```

where `sourcecityX` is the name of a city as a string that contains no spaces and is not longer than 30 characters.

Following this list of unique cities in the travel log is a list of N pairs of source and destination cities, as follows:

```
sourcecity1 destinationcity1
sourcecity2 destinationcity2
sourcecity3 destinationcity3
...
sourcecityN destinationcityN
```

where `sourcecityX` and `destinationcityX` are names of cities as strings containing no spaces but are separated by a single space. Every source and destination city name is one of the previously listed unique city names. A city may appear more than once as a source or as a destination, but no city will appear more than once as the destination for the same source city.

Moreover, no source city will have more than 6 different destinations.

A sample input file is provided called `travel-log.txt`, but your program must work for any input file that follows the file format given above.

Your Tasks

Complete the following tasks. **In completing these tasks you may not modify `travel-log.txt` at all.**

(a) Create a new file, `question2.c`.

(b) In `question2.c` design a data structure that will be used to store the contents of the input file. *Be sure to include comments explaining your data structure design — this is necessary for grading.* The design of the data structure is entirely up to you, but we recommend storing an array of elements (of a type of your choosing) that each contain information about a different unique source city and includes a reference to a data structure (also of your choosing) that stores a list of each city name that is a possible destination from that source city. See Figure 1 for a conceptual representation of our suggestion.

Exactly how you store the data and exactly what data you need store is for you to decide, but you'll be graded on the appropriateness of your decisions. You should read the rest of the tasks, below, before designing your data storage so that you are aware of any requirements that must be met, and can design the data storage so that it best supports fulfilling those requirements.

(c) In `question2.c`, write a function called `read_cities` that takes as a parameter a pointer to a file that has been freshly opened for reading that contains a travel log as described in the "Input File Format" section, above. This function must return a pointer to an instance of the data structure you designed in part (b) that has been populated with the data from the travel log file.

It is in this function that you should ensure all relevant memory is allocated for your data structure (hint: if you aren't using dynamic memory allocation in some way, you've probably made some bad choices).

You may write additional functions to support the operation of `read_cities()` if you choose.

(d) In `question2.c`, write a function called `display_cities()` that takes as a parameter a pointer to a populated instance of the data structure you defined in part (b), and any other parameters that you deem essential. This function should print out to the console each unique source city followed by every destination city that can be flown to directly from it:

```
<src>: <dest>, <dest>,
<src>: <dest>, <dest>, <dest>,
<src>: <dest>,
...,
<src>: <dest>, <dest>, <dest>, <dest>, <dest>,
```

Note that trailing commas at the end of an output line are acceptable.

The `display_cities()` function returns nothing.

You may write additional functions to support the operation of `display_cities()` if you choose.

(e) In `question2.c` write a function called `destroy_data_structure()` that takes as a parameter a pointer to a data structure retruned by returned by `read_cities` and frees **all** of the dynamically allocated memory used by all components of the data structure. This function returns nothing.

You may write additional functions to support the operation of `destroy_data_structure()` if you choose.

(f) In `question2.c` write a `main()` function to:

- open the input file `travel-log.txt` for **reading** (you may hard-code the filename);
- call `read_cities()` to create a populated data structure structure;
- call `display_cities()` that displays the populated structure;
- destroy (de-allocates) the data structure by calling `destroy_data_structure()`.

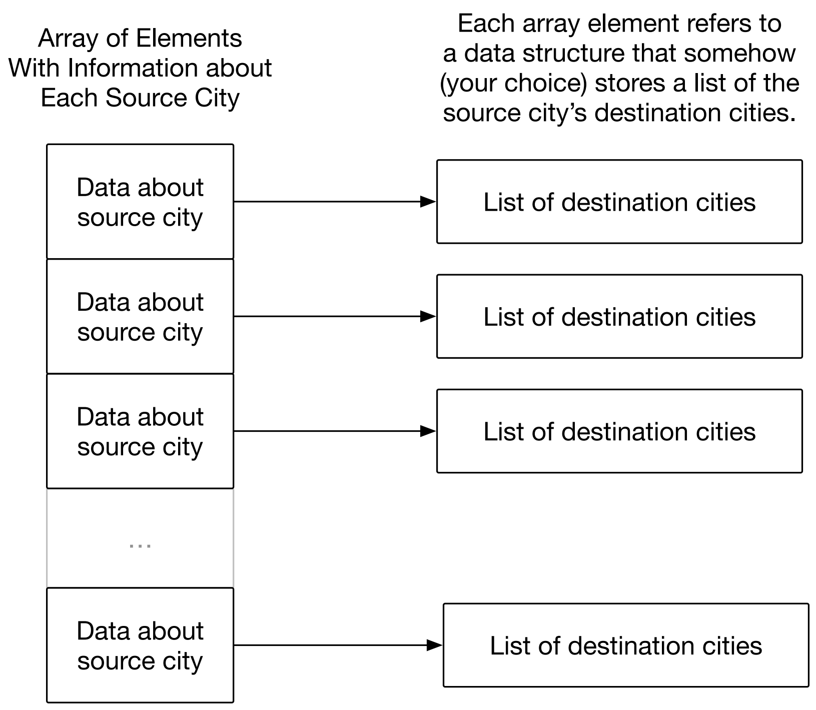
Array of Elements With Information about Each Source City
Each array element refers to a data structure that somehow (your choice) stores a list of the source city's destination cities.
Data about source city
Data about source city
Data about source city
...
Data about source city
List of destination cities
List of destination cities
List of destination cities
List of destination cities

Figure 1: A conceptual idea of how the data should be organized. We suggest an array of data elements (of a type that is your choice/design) that stores necessary information about each source city, including a reference to a data structure (also of your choice/design) that stores a list of the destination cities for that source city. *Hint: the array elements might store* ***other*** *information about the source city, in addition to the list of destination cities...*

Hand-in Checklist

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

- Question 1:
  - ☐ `items.c`
  - ☐ `items.h`
  - ☐ `question2.c`
  - ☐ `randomindex.c` (please hand in the provided file, unmodified)
  - ☐ `randomindex.h` (please hand in the provided file, unmodified)
- Question 2:
  - ☐ question2.c