The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 214– Programming Principles and Practice

Assignment 4

Date Due: Wednesday October 26, 2022, 6:00pm

Total Marks: 50

General Submission Instructions

- Assignments must be submitted using Canvas.
- Programs must be written in C conforming to the C11 standard. Everything we teach you is compliant with the C11 standard. Things we haven't taught you might not be, so use only the features of C that we have taught. If you try things you find on other websites (where you should definitely **not** be looking for solutions) they may not be C11 compliant. Everything you need to solve programming problems can be found in the course materials, or in the assignment itself.
- Include the following identification in a comment at the top of all your code files: your name, NSID, student ID, instructor's name, course name, and course section number.
- Late assignments are accepted with a penalty depending on how late it is received. Assignments are not accepted more than 2 days late (the assignment submission will close 48 hours after the due date passes and you will not be able to submit). See the course syllabus for the full late assignment and assignment extension policies for this class.
- **VERY IMPORTANT**: Canvas is very fragile when it comes to submitting multiple files. We insist that you package all of the files for all questions for the entire assignment into a single **ZIP** archive file. This can be done with a feature built into the Windows explorer (Windows), or with the **zip terminal command** (LINUX and Mac), or by selecting files in the Mac finder, right-clicking, and choosing "Compress ...". We **cannot accept** any other archive formats other than `ZIP` files. This means no `tar`, no `gzip`, no `7zip` (`.7zip`), and no WinRAR (`.rar`) files. **Non-ZIP archives will not be graded**.
- Instructions on "how to create zip archives" can be found here:
  `https://canvas.usask.ca/courses/62306/pages/how-to-zip-slash-compress-your-files`

Background

Command Line Arguments

This assignment requires the use of command line arguments to C programs. To learn about command line arguments, read Appendix C in the textbook before proceeding. It's not long, only about 3 pages!. You will use command line arguments in questions 1 and 2.

Dynamically-allocated Pointer Arrays

From textbook chapter 11, we know that we can declare a pointer array of a fixed size like this:

```
<pointer type> varname[<size>];
```

where `<pointer type>` is a pointer type, such as `int*` or `char*`, and `<size>` is a positive integer value. Used within in a function, such a declaration would use automatic allocation on the stack for the storage space for the `<size>` pointers. For example, an array of 10 pointers to integers would be declared thusly:

```
int* int_ptr_array[10];
```

and those 10 pointers would be in automatically allocated memory on the stack. But what if we didn't know how many integer pointers we needed until runtime? We would have to dynamically allocate such a pointer array with the desired number of pointers. Let's suppose we need to allocate an array of n pointers to integers at runtime. It would be done like this:

```
int** dy_int_ptr_array = (int**) malloc(sizeof(int*) * n);
```

The type of `dy_int_ptr_array` must be `int**` because it is a pointer to the first element of an array whose elements are of type `int*`. A pointer to something of type `int*` must therefore have type `int**` (a pointer to a pointer to an integer). Each element of both arrays `int_ptr_array` and `dy_int_ptr_array` is of type `int*`, but the type of `dy_int_ptr_array` cannot be an array type because it is dynamically allocated, so we have to use the type that such an array type would decay into, which is `int**`.

Now, suppose we have a `typedef` for a structure:

```
typedef struct _mystruct {
   ...
} MyStruct;
```

A automatically-allocated array of pointers to such structures would be declared thusly (similar to the declaration of `int_ptr_array`, above):

```
MyStruct* struct_ptr_array[10];
```

In Question 1 of this assignment, you'll need to create a **dynamically allocated** array of pointers to a structure type. Given the examples in this section it should not be a big leap to figure out how to dynamically allocate an array of pointers to a structure type, but exactly how to do that is left as a problem for Question 1. Use the example of allocating `dy_int_ptr_array`, above, as a guide — the pattern is the same, you just need to substitute in the right types.

Reminder: declaring and allocating a pointer array as shown above only creates space to store the **pointers** in the pointer array. It does not allocate memory for any data items that the pointers in the pointer array are expected to **point to**, nor does it initialize the pointers in the pointer array to point to anything valid. These things need to be done **after** dynamically allocating the pointer array.

Question 1 (18 points):

Purpose: To practice with structures and pointer arrays.

For this question you will read in the contents of a datafile, storing the **entire** contents of the file at once in a **array of pointers to structures** (a pointer array where each element is a pointer to a structure type).

Data File Format

Data files will contain information about loot items in a hypothetical game, formatted thusly:

- The first line of the data file contains a positive integer N that is the number of loot items that the file contains data for.
- The remainder of the file consists of N lines. Each line contains the data items for **one** loot item in the following order:
  - the **name** of the item consisting of not more than 30 characters that contains no spaces;
  - the **item level** of the item (positive integer);
  - the **value** of the item (positive integer);
  - the **rarity** of the item (a string of not more than 9 characters that contains no spaces). This can be one of four possible strings: `Common`, `Uncommon`, `Rare`, or `Legendary`;
  - the **durability** of the item (positive integer); and
  - the **probability** of the item being awarded from a loot chest (a real number not less than 0.0 and not more than 1.0).
- The data items on one line of the file may be separated by any amount of non-newline whitespace.
- The **total length** of any line in the file, including all whitespace and newline characters, is not more than 80 characters.

A sample input file, `loot.txt`, that conforms to the data file format above has been provided.

Program Requirements

Write a program that accepts exactly one command line argument: a filename. Your program must read in the data from the datafile (in the format described above), ask the user to enter a rarity level, and then display a list of the loot items and their properties that have the selected rarity.

Your program must work with any input data file that is in the format described above. Use exclusively the file I/O paradigm for reading tabular files as described in Appendix B of the textbook (even for the first line of the file!).

Item names in the datafile may contain underscores as placeholders for spaces within the item name, as seen in previous assignments. After reading the item names and storing them, replace the underscores with spaces using the `underscores_to_spaces()` function provided with Assignment 3, Question 3.

Your program must be readable. We will be assessing the readability and aesthetic presentation of this program. Your program must use a consistent whitespacing style. This includes spacing around and between operators and operands within lines, and blank-line spacing between lines of code and comment blocks. Use previous instructor solutions to assignments as a guide. Usually one space around/between operators and operands is preferred. Indentation style should be consistent throughout the code. One of the curly brace styles described in Section 1.3 of the textbook should be used consistently throughout the program code.

We have provided you with starter code in `asn4q1-starter.c` that contains comments that describe in greater detail how to complete the program. Wherever you see a comment that starts with "TODO", you need to do something!

When using C standard library functions, employ error checks as established in the textbook even where not explicitly asked for in the TODO comments in the starter code.

Note: As provided, `asn4q1-starter.c` will not compile without error. You will need to complete it to a certain extent before it will compile.

Sample Output

Here we show the expected output of several runs under different conditions. In all cases, lines shown in green are not program output but are the commands executed to run the program. Red text shows text entered by the user during the program's execution.

```
$ ./asn4q1
asn4q1: Error: Expected exactly one argument, the filename to read.
Got 0 arguments instead.
```

```
$ ./asn4q1 loot.txt other.txt
asn4q1: Error: Expected exactly one argument, the filename to read.
Got 2 arguments instead.
```

```
$ ./asn4q1 loot.txt
Successfully read 15 loot items from loot.txt.

List loot of what rarity? (Common, Uncommon, Rare, Legendary): Legendary
Pervon's Finger Bone 20 500000 Legendary 27 0.000010
Spatula of Justice 19 340123 Legendary 38 0.000098
Censer of Tarrasque Summoning 17 42000 Legendary 29 0.000240
```

```
$ ./asn4q1 loot.txt
Successfully read 15 loot items from loot.txt.

List loot of what rarity? (Common, Uncommon, Rare, Legendary): Uncommon
Dust of Deliciousness 7 600 Uncommon 1 0.150000
Scroll of Conjure Milk 3 125 Uncommon 1 0.100000
Hideous Halberd 8 2424 Uncommon 97 0.023000
The Shortest Bow 6 1989 Uncommon 18 0.007600
Scroll of Fog of Enlightenment 8 250 Uncommon 1 0.075000
```

```
$ ./asn4q1 loot.txt
Successfully read 15 loot items from loot.txt.

List loot of what rarity? (Common, Uncommon, Rare, Legendary): uncommon
```

```
$ ./asn4q1 doesnotexist.txt
doesnotexist.txt: No such file or directory
asn4q1: Error: was not able to open doesnotexist.txt for reading.
```

Testing

We require no testing output to be submitted for this question, but you are advised to test your code thoroughly for at least all of the situations demonstrated in the sample output, above.

Question 2 (18 points):

Purpose: To practice with dynamically allocated 2D arrays (1D views of 2D arrays).

A *magic square of size n* is an n by n matrix in which the sum of each row and column of the matrix are the same. For this question you'll be writing a program that reads one or more square matrices from separate data files and reports whether or not they are indeed magic squares.

Data File Format

The data file format for a magic square is quite simple. A data file for a magic square of size n contains $n^2 + 1$ integers. The first integer is the size of the magic square. The remaining n^2 integers are the entries in the magic square matrix in *row-major order*; that is, the file contains the numbers for the first row, from left-to-right, then the second row, left-to-right, the third row, and so on up to the n-th row. In other words, the matrix entires in the file appear in exactly the same order as they would appear in memory when stored in a 1D array that is a 1D view of a 2D array (see textbook Section 12.5.2 and 12.6). There may be any amount and type of whitespace between each integer in the file – thus the input files are essentially sequential files consisting of $n^2 + 1$ integers.

Three sample datafiles have been provided. If you want to generate your own magic squares, you can use `https://www.dcode.fr/magic-square`

Program Requirements

Your program must store magic squares in the following structure type:

```
typedef struct _magic_square {
    int size;       // Size of the magic square.
    int *square;    // A pointer to the first element of a 1D view of
                    // a 2D array with size rows and size columns
                    // containing the values in the entries of the
                    // magic square in row-major order.
                    // (dynamically allocated -- see Section 12.6
                    // of the textbook)
} MagicSquare;
```

Your program must consist of four functions:

1. A function called `read_magic_square` which takes one parameter: a filename. This function should read the data in the given filename (if it exists) and return a new dynamically allocated `MagicSquare` type that is initialized to contain the data in the file read.
2. A function called `test_magic_square` which takes one parameter: a pointer to a `MagicSquare` structure. The function must return 0 (to be interpreted as "false") if the magic square pointed to by the parameter is not a magic square, and must return 1 (to be interpreted as "true") if it **is** a magic square.
3. A function called `free_magic_square` which takes one parameter: a pointer to a `MagicSquare`. This function must free all dynamic memory associated with the magic square pointed to by the parameter.
4. A `main()` program which accepts one or more command line arguments that are filenames. For each filename given, use the functions described above to read the file, determine whether or not the resulting `MagicSquare`'s are indeed magic squares, and report the findings (see the sample output, below). This function is responsible for verifying that there is at least one command line

argument and terminating with an appropriate error message if no command line arguments are supplied. This function must also use `free_magic_square()` to free all dynamically allocated memory associated with magic squares.

Your solution must also satisfy the following additional requirements:

- Your program must utilize all conventions for error-checking for library functions established in the textbook and in this course.
- Your program must contain satisfactory function header documentation and inline commenting as per standards established on previous assignments. We will be assessing the quality of function documentation and inline comments for this question.
- Your program must be readable. We will be assessing the readability and aesthetic presentation of this program using the same criteria as Question 1.

Sample Output

```
$ ./magicsquare sq4.txt sq5.txt sq9.txt
sq4.txt is a magic square.
sq5.txt is NOT a magic square.
sq9.txt is a magic square.
```

Note: the first line in green text is not output, it is the command to run the program. The remaining lines are the output.

Question 3 (14 points):

Purpose: To practice designing a compound data structure comprised of dynamically allocated C arrays and structures.

The notorious pirate captain, Tractor Jack, has hired you as their "IT person". Help Jack design a data structure to hold information about his plunder from raiding towns along the Saskatchewan River.

Jack would like to keep track of the plunder for each of his raids. The information to be stored about a single raid consists of:

- The name of the raid's commander (may contain spaces, not more than 99 characters)
- The name of the river town that was raided (may contain spaces, not more than 99 characters).
- The date and time of the raid, consisting of:
  - The day of the month (positive integer not less than 1 and not more than 31)
  - The month (positive integer not less than 1 and not more than 12)
  - The year (positive integer)
  - The hour (non-negative integer not less than 0 and not more than 23)
  - The minute (non-negative integer not less than 0 and not more than 59)
- The plunder obtained from the raid, consisting of:
  - The number of sacks of wheat plundered (non-negative integer)
  - The number of sacks of barley plundered (non-negative integer)
  - The number of sacks of other grains plundered (non-negative integer)

Jack also wants to be able to look up which raids were plundered by a given ship in his fleet. The data for one ship consists of:

- The name of the ship (may contain spaces, arbitrary length)
- A list of the raids stored in the ship's hold (arbitrary length).

Finally the complete data structure consists of:

- The number of ships in Jack's fleet. (positive integer)
- A list of ships (arbitrary length).

In all cases where an arbitrary length is specified, this means that the length could be anything but once the data structure is created, those lengths are fixed. These arbitrary but fixed lengths will be known at runtime (see step 2 below).

Your Tasks

1. Design a data structure to organize and store the information as described above. We have given you a small start in the provided starter code, `asn4q3-starter.c`. There you will see a type definition for a complete database — a structure type named `ShipDatabase`. In order to define the members of the `ShipDatabase` structure, you'll need to define other types in which to store the data described above. You can achieve a satisfactory solution using structures, and arrays (which might include an array of structures!). However, remember that the number of ships and the number of raids that each ship can store are not fixed, so your data structure needs to be designed so that it can be dynamically allocated. You can add whatever fields you need to the `ShipDatabase` structure, but don't change its name. Read step 2 before you start designing!!!!
2. Write a function called `new_database()` which takes three parameters: the number of ships in the fleet, a pointer to the first element of an an array of length equal to the number of ships that stores the number of raids that can be stored in the cargo hold of each ship, and a pointer to

the first element of a pointer array with length equal to the number of ships that holds the name of each ship. This function must allocate space for a new ship database that has the specified number of ships, and where each ship can hold plunder up to the specified number of raids. The name of the ship must be stored in the database, but no other information in the database needs to be initialized. The function should return a pointer to the newly allocated database. We have gotten you started by providing you with a function header and empty definition for the `new_database()` function in `asn4q3-starter.c`. You can write whatever you want inside the function definition, but you cannot modify the header, and you must return a pointer to a `ShipDatabase` structure.

Remember: the `new_database()` function **only** needs to create a ship that can hold the specified amount of data. It does not need to initialize the database with any actual data (except for the name of the ship)

3. Write a function called `scuttle_database()` which takes one parameter: a pointer to a ship database. This function must free all memory associated with the database. It should return nothing. Again, we have gotten you started by providing you with a function header and empty definition for the `scuttle_database()` function in `asn4q3-starter.c`. You can write whatever you need to in the function definition, but you may not modify the function header.
4. You can use the provided `main()` function in `asn4q3-starter.c` to test your implementation of the functions in steps 2 and 3 to **some** extent. If a crash occurs, it's safe to assume that there is a problem with your functions, very likely an error in the use of dynamic memory allocation, dynamic memory freeing, or some other misuse of pointers. However, an error-free run does not prove that your design is acceptable or that your code is correct. It only proves that what you are doing does not cause a runtime error. In other words, running the main program can prove incorrectness, but such a test alone cannot not prove correctness.

There is not one unique correct answer to this problem. There are a small number of variations that are acceptable. However, an acceptable design must allow for the creation of a database with any number of ships (at least 1) and any valid raid capacities for each ship.

Your program must be readable. We will be assessing the readability and aesthetic presentation of this program using the same criteria as Question 1.

Your program must make the standard error checks established in the textbook when using C library functions.

Your program must be documented throughout. We will assess inline comments and comments that describe the data structure(s) that you define, but not function header comments as these are already present in the starter code.

Implementation Notes

You might be concerned that the code we have asked you to write doesn't really **do** much. Don't be. This is largely an assessment of your ability to design a reasonable compound data structure and manage its memory.

Files Provided

asn4q1-starter.c: The starter code for question 1.
loot.txt: Sample input data file for question 1.
sq4.txt, sq5.txt, sq9.txt: Sample input data files for question 2.
asn4q3-starter.c: The starter code for question 3.

What to Hand In

Hand in a `.zip` file archive which contains the following files:

asn4q1.c: The code for your solution to question 1.
asn4q2.c: The code for your solution to question 2.
asn4q3.c: The code for your solution to question 3.

VERY IMPORTANT: Canvas is very fragile when it comes to submitting multiple files. We insist that you package all of the files for all questions for the entire assignment into a single **ZIP** archive file. This can be done with a feature built into the Windows explorer (Windows), or with the **zip terminal command** (LINUX and Mac), or by selecting files in the Mac finder, right-clicking, and choosing "Compress ...". We **cannot accept** any other archive formats other than `ZIP` files. This means no `tar`, no `gzip`, no `7zip` (`.7zip`), and no WinRAR (`.rar`) files. **Non-ZIP archives will not be graded**. We will not grade assignments where these submission instructions are not followed.

Grading Rubric

The grading rubric can be found on Canvas.