The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 214– Programming Principles and Practice

Assignment 3

Date Due: October 14, 2022, 6:00pm

Total Marks: 35

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

This assignment requires that you use file I/O functions to read and write data to and from files. To learn how to do this, read Appendix B of the textbook before proceeding.

Problems

Question 1 (4 points):

Purpose: To practice writing data to text files with `fprintf()`, reading strings from the console with `scanf()`, and storing such strings.

For this question, start with the solution to Assignment 2 question 1 provided in the file `asn3q1-starter.c`. Your task is to modify the given program in the following ways:

1. In the `main()` function, after the user has input the monster starting and ending levels, monster type, and hero level, prompt the user to enter a filename. In the prompt, let the user know that the filename cannot contain spaces. Read the filename with `scanf()` and store it appropriately. You may assume that the user enters a filename that does not contain spaces, and that the length of the filename entered does not exceed 100 characters.
2. Modify the remainder of the program as needed so that instead of the table title, table header row, and table data rows being printed to the console, they are written to a file with the name entered by the user. This output should overwrite any existing file with the same name. When working with files, employ the error checking demonstrated in the examples in textbook Appendix B (there are additional error checks that could be done beyond what is in Appendix B but you don't have to use those, just make use of the kinds of checks demonstrated). Terminate the program gracefully if an error is detected using the error reporting conventions we have established previously in this course.

Implementation Notes

Do not modify the `base_xp()` and `adjuted_xp()` functions. You may modify anything else as needed.

Don't forget to follow the extra error reporting step if there is a problem opening the file for reading. See the section titled "Informative Errors using `perror()`" within Appendix section B.3.1.

Testing

You do not need to submit any proof of testing or program output for this question.

Question 2 (4 points):

Purpose: To practice reading numeric data from sequential files.

In this question you'll write a program that reads a sequential file containing the dollar values of sales transactions and determines the average transaction value.

File Format

The file format will be that of a sequential file of real numbers representing dollar values of sales transactions. We remind you that such a sequential file contains numbers separated by any amount and type of whitespace (see also Appendix B.4.2 in the text book). We will provide you with one test data file, but your program must work with any data file formatted in this way.

Program Requirements

Your program must do the following:

1. Open the data file `sales.dat` for reading. This file is a plain text file, even though it does not have a `.txt` extension. You may hard-code the filename. Use the error-checking and reporting conventions established in Appendix B of the textbook.
2. Read each dollar amount from the file and compute their sum. Use the error-checking and reporting conventions established in Appendix B.
3. Close the input file.
4. Display a message on the console reporting the average value of a sales transaction.

Implementation Notes

As stated above, when using library functions for file I/O, use the error-checking and reporting conventions established in Appendix B.

Remember that since you are computing an average, you don't need to store each individual sales transaction value read — you can simply keep a running sum and count of the values read. If you are trying to store the individual values in an array, you're doing more work than needed!

Question 3 (14 points):

Purpose: To practice reading combinations of strings and numbers from tabular files.

You will write a program that will solve the problem of reading some mixed type data from a tabular file and printing to the console in an attractive manner. We will begin by describing the file format, and then describe what the program should do.

Data File Format

The data file format for this question is a tabular file containing information about quests. Each line contains a quest name, followed by the appropriate character level at which to attempt the quest, followed by the number of experience points the quest is worth upon completion. In addition, valid data files have the following properties:

- A file may contain any number of lines, but always contains at least one line.
- A file contains no blank lines.
- Quest names are not longer than 50 characters and contain no spaces.
- Recommended character levels and experience point values are always non-negative integers not exceeding 1 million (remember, zero is non-negative).

Your program must be capable of processing any data file with the format described above. An example data file, `quests.txt`, is provided for you.

Program Requirements

Write your program so that it prints the data in the file to the console according to the following specifications.

- Print to the console the data in the input file `quests.txt` as a table. The table should have three columns with widths 50, 12, and 10 characters from left-to-right, respectively. There must be **two** additional spaces between columns. Data must be **left-aligned** in each column.
- The first row of the printed table must be a header row with headings `Quest Name`, `Quest Level`, and `Experience`. The header row should respect the same column widths and spacing as the data rows, as described in the previous bullet point.
- The remaining rows are data rows, one for each quest in the data file. The data from each quest comes from one line in the input file. Hint: You need to read in the data in the file, printing out table rows as you do so.
- Use the paradigm of calling `fgets()` to read an entire line from the input file, then calling `sscanf()` to break the line into columns as described in Appendix B.4.3 of the textbook.
- The length of the character array used to store a line from the file should be fixed, and must be the smallest length possible to store the longest possible line that could be read by `fgets()` for the file format in this question.
- The length of the character array used to store a quest name should be fixed, and must be the smallest length possible to store the longest possible quest name.
- Employ the error-checking and reporting conventions established for working with files, as demonstrated in the examples in textbook Appendix B. The program should terminate gracefully when an error is encountered using the conventions established earlier in this course.
- You'll notice that quest names in the provided data file contain underscores where there should be spaces. This makes it possible to obtain the strings easily with `sscanf()`. Before displaying the quest names, you must replace the underscores with spaces. To do this you must write a

function called `underscores_to_spaces()` that accepts one parameter, a character array containing a string, and replaces each underscore in the provided string with a space. You must call this function in your `main()` program to convert the underscores in quest names to spaces before printing them.
- Expected output for the given `quests.txt` is given below.

Implementation Notes

- It is not necessary to store the entire contents of the input data file in memory. It is sufficient to read the data for one quest, print it out, then read the data for the next quest, and so on.
- You may hard-code the filename `quests.txt` into your program, however, your program must still work correctly for any input data file that conforms to the described data file format, above. That is, if the hard-coded filename were changed to the name of another compliant data file, the program must still work correctly. We reserve the right to test your programs with our own compliant data files.
- We recommend getting the program working without substituting the underscores in quest names for spaces first, and then write and call the function to replace them.
- Consider **very** carefully the length of the character array you use to store file lines and quest names. To come up with the right array length for file lines, think very carefully what happens when you use `fgets()` to read a string, what it stores, and generally how strings are stored in character arrays (hint: the correct length is not 80!). Similarly, for the length of the array to store quest names, think about how `sscanf()` works and what it will store when extracting the quest name from the file line (hint: the correct length is not 50!).
- Although the description of this problem is quite long, your solution should be quite short (mine is 17 lines of actual code within `main()` not counting comments or blank lines, plus a few lines for the `underscores_to_spaces()` function). A solid understanding of Appendix B in the textbook is essential. Pay special attention to Appendix sections B.3 and B.4.3 and Section 8.7.3 where it talks about `fgets()`.

Testing

Once you have your program working when using the provided `quests.txt` as input, create your own input file named `quests-abc123.txt` where `abc123` is replaced with your NSID. Your input file must conform to the same file format as described above (remember if you want quest names with spaces in them to use underscores when writing them into the data file!). It must contain data for at least 3 but not more than 6 quests. Your data file must be unique and not re-use any data from the given file, so make up some fun, interesting quest names. There's no reason why your data file should be the same as that of another student. Test your program on your new data file. You can modify the hard-coded input filename in your code in order to test your program on your new data file. We don't care which filename is hard-coded there when you hand in the code.

Run your program twice, once with the provided input file and once with your `quests-abc123.txt` input file. Copy the output produced when the input file is `quests.txt` into a file called `asn3q2-quests-output.txt`. Copy the output produced when the input file is `quests-abc123.txt` to a file called `asn3q2-abc123-output.txt`. Include both files containing output in your submission (see the What to Hand In section at the end of this document).

Expected Output

The output should look exactly like this when `quest.txt` is used as input, and every program requirement is satisfied.

```
Quest Name                                          Quest Level   Experience
Destroy the worms in Princess Bubblegum's basement  1             10
Infiltrate the bandit's lair                        2             40
Defeat Goliad                                       20            4000
Locate the Lich's lair                              35            12250
Atone for Shoko's sins                              15            2250
Make an amazing sandwich                            7             490
Find the Ice King's Wizard Eye                      9             810
Get some pickles from Prismo                        27            7290
Rescue Wildberry Princess from the Ice King         12            1440
Win Wizard Battle                                   13            1690
Eat Marceline's fries                               3             90
Rescue Marceline from the Nightosphere              21            441
Discover Peppermint Butler's secrets                32            10240
Defeat the Ice King's penguin army                  27            7290
Discover the Ice King's secret past                 10            1000
Find out what Beemo does when he is alone           6             360
```

Question 4 (13 points):

Purpose: To practice using arrays of structure, strings, and passing arrays to functions.

For this question you will solve the same problem as in Assignment 2 question 5 (this was the question where we simulated a menu ordering system). The difference now is that instead of hard-coding the menu items and prices in the program, we will read them from a data file and store them in an array of structures.

Data File Format

The file format for the data file containing menu items is as follows:

- A menu data file contains at most 20 lines.
- A menu data file contains no blank lines.
- Each line of a menu data file is no more than 80 characters long (not including the newline character).
- Each line of a menu data file contains the name of a menu item and its price as a real number. These two data items are separated by whitespace.
- Menu item names contain no spaces.
- Menu item names do not exceed 60 characters in length.
- Menu item names may contain underscores which are to be replaced by spaces before being displayed to the user (similar to the quest names in Question 3 on this assignment).

Program Requirements

The program's behaviour must be identical to that of the program described in Assignment 2, Question 5. In other words, the program must work exactly the same way from the point of view of the user. However, the implementation will change because we are now reading the menu data from a file rather than hard-coding the data. The new implementation must meet the following requirements:

- Design an exciting unique menu (you can use the same menu as you used for Assignment 2, Question 5 if it was indeed unique) and create a menu data file for it, following the data file format described above. There's no reason two students should hand in the same menu. Name your data file `menu-abc123.txt` where abc123 is your NSID.
- Add new function, `read_menu_from_file`, which accepts the following parameters:
  - A string that is the path to the menu data file to read.
  - An array of `MenuItem` structures into which to store the data read. The `MenuItem` structure and type definition is given in the provided starter code.
  - An integer denoting the maximum number of menu items to read.

  and returns:

  - The actual number of menu items read from the file.

  The function must read up to the maximum number of specified menu items and store the menu data in successive elements of the array. You must report any errors that occur when reading the file. Also you must report an error if the file contains more menu items than the maximum number of items specified by the 3rd parameter. On detection of any error in this function, terminate the program. When reading menu item names, use the `underscores_to_spaces()` you wrote for Question 3 to change the underscores in the menu item names strings to spaces. Document the function parameters in a comment block before the function header.

- The `print_menu()` function from Assignment 2 Question 5 must be modified so that it takes an array of MenuItem structures and the number of menu items stored in that array as additional arguments and prints out the menu data that was read from a file. Otherwise the function should behave in exactly the same way as in Assignment 2, Question 5.
- The main program must be completed by following the TODO comments in the provided starter code for this question.
- Make appropriate use of the provided `#define`-d sizes at the top of the provided starter code
- The program must work correctly for any file that conforms to the data file format for this question.

Testing

Run your program once using your unique menu data file, `menu-abc123.txt`, inputting at least 3 valid menu choices, and at least one menu choice that is a valid integer but an invalid menu choice. Copy a transcript of the output of this run to a text file named `asn2q5-output.txt` and submit it with your completed program. Also hand in the `menu-abc123.txt` data file you created (see also "What to Hand In" later in this document).

Sample Output

The output should have the same form as for Question 5 of Assignment 2.

Files Provided

asn3q1-starter.c The starter code for question 1.
sales.dat The data file for question 2.
quests.txt Sample input data file for question 3.
asn3q4-starter.c The starter code for question 4.

What to Hand In

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

asn3q1.c The code for your solution to question 1.
asn3q2.c The code for your solution to question 3.
asn3q3.c The code for your solution to question 3.
quests-abc123.txt Your unique input data file that conforms to the file format specifications for question 3 (replace `abc123` with your NSID).
asn3q3-quests-output.txt The output from the test of your question 3 solution when the input file is `quests.txt`
asn3q3-abc123-output.txt The output from the test of your question 3 solution when the input file is `quests-abc123.txt` (replace `abc123` with your NSID).
asn3q4.c The code for your solution to question 4.
asn3q4-output.txt The transcript of the tests of your solution to question 4.
menu-abc123.txt The data file you created for question 4.

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.