COMP 2710: Project 1 – Phase 3: Implementation (30 points)

Goals of Project 1 – Phase 3:

- To design and implement a non-trivial application using the function-oriented development approach.
- To use singly linked lists
- To perform separate compilation.
- To gain experience with unit and system testing.
- To develop a reasonably user-friendly application

1. Overview

1.1 (20 points) Implement the simple text-based game in C++

- ***Singly linked list.*** Your underlying data structures must include at least one singly linked list.

- ***Basic Data Structures.*** Recall that basic data structures should include: Menu, System, Character, Scoreboard, Scorefile, Encounter, and Puzzle.

- ***Main Function.*** The main function in your program must be short. This is because a short main function generally improves the quality of your code. Please refer to the following webpage for further discussions on why we should keep our main function short.

http://programmers.stackexchange.com/questions/85657/why-should-main-be-short

1.2. (10 points) Test Results: *After developing the text-based game*, actually try all of your test cases (both system and unit testing). Actually show the results of your testing (a copy and paste from your program output is fine – don't stress too much about formatting as long as the results are clear). You should have test results for every test case you described. If your system doesn't behave as intended, you should note this.

Note: You have developed test drivers in phase 2 of this project. Your driver output will substitute for unit test results.

2. Programming Environment

Write a short program in C++. Compile and run it using the g++ compiler on a Linux box (computer labs in Shelby, your home Linux machine, a Linux box on a virtual machine, or using an emulator like Cygwin).

3. Programming Requirements

Write a program called <username>-project1.cpp

3.1 Function-Oriented Approach

You must use an structure-oriented approach to solve this problem (in other words, you will need write function definitions and use those functions, you can't just throw everything in main()). A well-done implementation will produce a number of robust functions, many of which may be useful for future programs in this course and beyond.

Remember good design practices include:

- A function should do one thing, and do it well
- Functions should NOT be highly coupled

Some potential functions:

- A few functions to deal with the Menu issues, handling basic user screw-ups (choosing an option out of bounds).
- A few functions for the system, which instantiates the other functions and runs encounters. Receives input from the Menu functions.
- A few function for Encounters, providing the basic framework for a generic encounter (with additional information feed in from system as the encounters occur).
- A Puzzle function which operates with Encounter.
- A HighScores data type which loads, sorts, and collects the high scores.
- A few functions for Characters, keeping track of all the data relating to the character.

3.2 **Singly Linked List**

Your underlying data structures should include a singly linked list. You will lose points if you do not use singly linked lists**.**

3.3 **Comments**

Use comments to provide a heading at the top of your code containing your name, Auburn Userid, and filename. Also describe any help or sources that you used (as per the syllabus).

Follow the comment standard posted on the web or some alternate, approved standard.

3.4 File Names

You will lose points if you do not use the specific program file name, or do not have a comment block on **EVERY** program you hand in.

3.5 Reuse well-written classes

You are welcome to reuse well-written functions from earlier COMP 2710 homework assignments or projects (but indicate where they came from).

3.6 No Global variables
You may not use global variables or global functions – all your data/operations must be contained within the functions of your project.

3.7 Welcome Message and Main menu
The program first prompts the user for a name by printing “What’s your name?”, and then prints a welcome message to the user.

Then, the user should be given a menu of options, such as the following:

1) Start a New Game of Dunstan and Dragons!
2) View top 10 High Scores
3) Quit

You DO NOT need any graphical user interface for this simple, text-based game. If you want to implement a visualization of some sort, then that is extra.

3.8 Scoreboard
Starting a new game will perform as described above. Viewing the top 10 High Scores will require those scores to be read from a file. The scores need to be sorted from highest to lowest. If a game results in a score higher than the existing high scores, you should replace the lowest score with the new higher score. The user should always have the ability to quit and this should exit the program normally.

For the high scores, you will have to use some simple file input/output. When you have to update the high scores, it is easier to just overwrite the scores file than to try to modify an existing scores file.

3.9 Usability Concerns and Error-Checking
Your program's output need not exactly match the style of the sample output (see the end of this file for one example of sample output). However, please note that since the program is using lots of randomization you may not get the same results each time you test.

You should appropriately prompt your user and assume that they only have basic knowledge of the system.

You should provide enough error-checking that a moderately informed user will not crash your program. This should be discovered through your unit-testing. Your prompts should still inform the user of what is expected of them, even if you have error-checking in place.

4. Separate Compilation:
You must use separate compilation and create a makefile for this project.

What is Make?

Make is a program that looks for a file called "makefile" or "Makefile", within the makefile are variables and things called dependencies. There are many things you can do with makefiles, if all you've ever done with makefiles is compile C or C++ then you are missing out. Pretty much anything that needs to be compiled (postscript, java, Fortran), can utilize makefiles.

Format of Makefiles -- Variables

First, lets talk about the simpler of the two ideas in makefiles, variables. Variable definitions are in the following format:

```
VARNAME = Value
```

So lets say I want to use a variable to set what compiler i'm going to use. This is helpful b/c you may want to switch from cc to gcc or to g++. We would have the following line in our makefile

```
CC = g++
```

This assigns the variable CC to the string "gcc". To expand variables, use the following form:

```
${VARNAME}
```

So to expand our CC variable we would say:

```
${CC}
```

Format of Makefiles -- Dependencies

Dependencies are the heart of makefiles. Without them nothing would work. Dependencies have the following form:

dependecy1: dependencyA dependencyB ... dependencyN

  command for dependency1

Check out the following links for an introduction and tutorials about makefile:
https://opensource.com/article/18/8/what-how-makefile
http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

5. Deliverables

5.1 Files to be Submitted

Please create and submit a single compressed (.tar.gz) file named "project1.tar.gz" through the Canvas system online. Next section shows you how to compress files in Linux.

- **Makefile** (this should make your project)
- **Multiple source code files.** These files are source code of your project implementations. For example, your files may include encounters.cpp, encounters.h, puzzle.cpp, puzzle.h, scoreboard.cpp, scoreboard.h, menu.cpp, menu.h, testdrivers.cpp, testdriver.h, system.cpp, system.h

  The above files are an example for separate compilation; you may create and name your own multiple source code files. You are not encouraged to use two simple files like "project1_demo.cpp" and "project1_functions.cpp" files.

- **project1_results.pdf** (This file includes all process information, especially the results of testing)

Note: You will lose points (at least 5 points and up to 10 points) if you do not submit a single compressed file and name your compressed file in the format described in this section.

5.2 Create your compressed file

To create a compressed tar.gz file from multiple files or/and folders, we need to run the tar command as follows.

```
tar -czf project1.tar.gz <COMP2710_project1_folder>
```

where <COMP2710_project1_folder> is a folder that contains Makefile, header file, and other source code of your project; `project1.tar.gz` is the single compressed file to be submitted via Canvas. For example, my single compressed file to be submitted can be created using the following command:

```
tar -czf project1.tar.gz ./comp2710/project1
```

where ./comp2710/project1 is a folder that contains files for my project 1.

6. Grading Criteria

6.1 (20 points) Implementation

- (2 points) submitted file names
- (2 points) Use comments to provide a heading at the top of your code containing your name, Auburn Userid, and filename. Please describe any help or sources that you used (as per the syllabus).
- (4 points) a singly linked list

- (5 points) makefile
- (6 points) game functions
- (1 point) coding style

6.2 (10 points) Test Results

- (2 points) Scoreboard
- (3 points) Main menu and game menu
- (5 points) Game functions (e.g., Move, read, search, view character)