COSC 220 - Computer Science II
Lab 11

Dr. Joe Anderson

Due: 6 December 2019

1 Objectives

1. Gain familiarity with inheritance and polymorphism
2. Gain familiarity with c++ standard library containers
3. Gain familiarity to state-machine program control flow

2 Tasks

1. Download the `TextGame.zip` file from the course webpage and unzip it into a folder called `Lab 11`.
2. Study the code and review the program structure:
   (a) You can compile the code with the provided `Makefile`.
   (b) The main game loop (in `main.cpp`) tracks the current `GameState` object
      i. This class uses a `std::map` object to hold the user choices. You can think of a `std::map<T1,T2>` as a set of objects of type `std::pair<T1,T2>` where `T1` and `T2` are the types of each object in the pairs.
      ii. See `http://en.cppreference.com/w/cpp/container/map` for the documentation.
   (c) Each iteration of the main game loop executes the `printOptions` method of the current state
   (d) The user input is then passed to the `handleInput` method of the current state, and the return value is stored as the new `currentState`
   (e) The parent class is `GameState` which declares some functions which must be implemented in *every* class that inherits from it. One example of such a state is the `TravelState`. Study this state and get a feel for how it is used in the program.
3. Open the `idlestate.h` file and implement it to allow the user to either stay resting, or to get up and resume walking
   (a) You may need to remember which direction the user was walking, and whether there was a crossroad option. To this end, you may need to add private variables to the `IdleState` class
4. A better solution to the "resume" problem is to adjust the flow of the program as follows:
   (a) In the main function, declare a `std::stack<GameState*>` to hold a stack of pointers to `GameState` objects
      i. See `http://en.cppreference.com/w/cpp/container/stack` for the documentation.

(b) Change the `handleInput` function to no longer return a `GameState` object, but rather take a `std::stack<GameState*>&` parameter. Instead of returning the next state, we will permit the current state to push new states onto the main stack as needed.

(c) On each iteration of the main loop, use the stack `.top()` and `.pop()` to get and remove the top pointer of the stack, storing it as the new `currentState`.

(d) Now when we transition from `TravelState` to `IdleState`, we can simply push the old travel state (rather, a copy of it) onto the stack, *and then* push the new `IdleState`.

5. Go through a similar procedure to implement the `CombatState`

   (a) To start, give the user only two options: attack or flee

   (b) You will need a private variable to hold the battle data

      i. For now, only store some number for the monster health

      ii. Also store some number of damage to deal to the monster

   (c) If the user chooses to attack, subtract the attack amount from the monster's health

   (d) If the monster drops to zero health, end the combat and congratulate the user. Otherwise, push an updated `CombatState` object onto the stack.

6. To trigger the combat, use the `combatProbability` to "interrupt" the main game loop with that probability, sending the user into combat

7. Be sure to set `combatProbability` of the `CombatState` to zero, so that we can't get attacked while already in combat!

8. On your own:

   (a) Try creating a class hierarchy to hold the "players" of the game: a `LivingEntity`, with subclasses `Monster` and `Player`

      i. In the respective constructors, you may hard-code certain attributes (attack damage, health values) or try randomizing them!

      ii. Use the `Monster` in the `CombatState` as a private variable to store the monster along with the state

   (b) Adjust the `handleInput` to also take the player object as an argument, so that it can be modified (e.g. if the player takes damage or gets equipment)

   (c) Add monster counter-attacks, to take away from the player health during combat

   (d) Let the player re-generate health while they are resting

9. This Lab may be turned in by May 13.

10. If you are interested, you can see a more fully implemented form of this game at `https://github.com/jtanderson/TextAdventure`. Feel free to clone the code for yourself, make changes, and submit code to the main repository!

3 Submission

Upload your project files to the course canvas system in a single zipped folder: To zip on Linux:

1. To zip a single folder into a single archive called "myZip.zip":

```
zip -r myZip.zip folderName
```

2. To zip multiple files into a zip file called "myZip.zip":

```
zip myZip.zip file1.cpp file2.h file3 file4.cpp
```

Turn in (stapled) printouts of your source code, properly commented and formatted with your name, course number, and complete description of the code.

Also turn in printouts reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.