COMP322: Assignment 3- Winter 2013

Due at 11:30pm EST, 22nd March 2013

1 Introduction

Rock Paper Scissors is a two person game where each player, simultaneously puts their hand in a shape that represents either a rock, a piece of paper, or a scissor. A player wins according to the following rules:

1. A player who plays rock beats a player who plays scissors (because a rock would smash scissors)
2. A player who plays scissors beats a player who plays paper (because scissors could cut paper)
3. A player who plays paper beats a player who plays rock (because paper can cover a rock?)
4. If the players play the same thing, the game is a tie.

Rock paper scissor (RPS) is a critical game for settling important disputes between children such as who gets the last cookie. You can read more information about it at `http://en.wikipedia.org/wiki/Rock-paper-scissors`

As the game is fully symmetric (all 3 choices are equally good), the game tends to evolve into a game of "I think my opponent will play paper. So I should play scissor. But if my opponent thinks I'll play scissor, then he will play rock, so actually I should play scissor" and so on forever. There actually is a deep strategy involved in playing this game and people play this game competitively in tournaments. There are openings and advanced strategies, more of which can be found at `http://rpsgame.com/strategy.html` and `http://www.worldrps.com/`

For this assignment, we will implement a Rock Paper Scissor tournament. In doing so, we will create several types of players using inheritance and classes in C++.

Part 1: RPSPlayer.cc

On the course webpage, you will find a header file defining an interface or abstract class called `RPSPlayer.h`. An `RPSPlayer` stores as private properties two vectors representing the list of moves that have been made by the players so far in the game. The method `chooseMove` is a *pure virtual function*. This means that you can not actually create an object of type RPSPlayer because the method `chooseMove` is not defined. However, classes that inherit from `RPSPlayer` will implement `chooseMove` in a different way. In this way, we can create several `RPSPlayer`, all of which share some similarity, but can choose a move using a different method but via the same interface.

For example, one kind of RPSPlayer (called `HumanPlayer` for example) might ask the user to input a choice. Another player called `SmartPlayer` might analyze the prior moves

made in the game and come up with a choice based on that. (For example, noticing that the opponent always plays rock, one might decide to choose Scissors.)

You should start by doing the following:

1. Implement the non virtual method `evaluatePlay` inside the RPSPlayer. This method should return an `RPSResult` representing whether the first move won, the second move, or if it was a tie. Note that `RPSResult` is an enum (see below). This method is a static method meaning you do not call it on an object.

2. Implement the non virtual method `addRound` inside the RPSPlayer. This method takes as input 2 moves, and stores them inside the class.

An enum is a simple way to make a new type in C++. The point is to gain type safety and readability. If you write something like the following

```
enum DayOfWeek { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
         THURSDAY, FRIDAY, SATURDAY, SUNDAY};
```

you can now use DayOfWeek as a type. This enables you to write a function returning a DayOfWeek. The alternative would be an int code (where for example 0 represents Sunday, 1 represents Monday, etc), but this approach is safer as it means you can only assign to a variable of type DayOfWeek one of those 7 values, which you can do for example, by the following:

```
DayOfWeek favourite = FRIDAY;
```

Part 2: Implementing players

You should now implement the class `RPSPlayer` by creating at least 2 classes that inherit from RPSPlayer but have a different technique for `chooseMove`. An example player who always chooses rock, is posted on the course webpage in a file `RockPlayer.cc`. You should provide at least 2 more classes. For the benefit of simplicity of marking, you should put all these classes into a file `Players.cc`. If you need to add private properties to help you choose a move, this is allowed (and encouraged!) Some example players you could make `HumanPlayer`, `BeatOpponentsPreviousMovePlayer`, `FileReaderPlayer`, etc.

The usefulness of polymorphism and inheritance comes when you deal with pointers. One can NOT write:

```
RockPlayer rock;
//.....set rock
RPSPlayer player = rock; //truncates rock
```

However, you *can* write:

```
RPSPlayer player1 = new RockPlayer();
player1->chooseMove();
```

and due to polymorphism, the chooseMove that is executed is based on whatever type of player player1 points to. Since ALL implementing classes are required to provide `chooseMove` we know this will exist. The convenience of this will be seen in the next part, but the idea is you can write all your subsequent code ignoring the exact kind of player you are using (after the initial set up).

Part 3: Implementing a tournament

You should now implement a rock paper scissor tournament with 128 players in it. Put this code in a program `RPSTournament.cc`. The tournament should consist of a variety of kinds of players and you should use an stl library class to store them. Note that it is important that you use `vector<RPSPlayer*>` for example as opposed to `vector<RPSPlayer>`. If you use the second, you would run into issues because the same truncation would occur. The downside of this is you need to be responsible for memory management of all these pointers. As a side note, there are ways to create what are called *smart pointers* to avoid this sort of issue. See `http://www.codeproject.com/Articles/15351/Implementing-a-simple-smart-pointer-in-c` for an example. Boost library also has something similar implemented.

The way the tournament should work is the following:

1. First create all the players and add pointers to them to the stl library container.
2. Pair each player with another player and have them play a round of rock paper scissor. The winner moves to the next round. (If there is an odd number of players at some point, you should automatically move one player to the next round.)
3. If a particular matchup is a tie, let the players play again until the tie is resolved. If after 3 plays, they are still tied, then they both are eliminated. (This is how you could end up with an odd number of players left)
4. Repeat the second and third steps until there is just one player left in the tournament (or zero if the finals end in a tie).

Notice when setting up this tournament, that you have successfully abstracted the particulars of which kind of player you are dealing with since you are only using the `chooseMove` method and focusing on the tournament implementation rather than worrying about what kind of player you are dealing with.

2 Submitting your assignment

What To Submit

`RPSPlayer.cc`
`Players.cc`
`RPSTournament.cc`
`Any other files needed for compiling (e.g. header files`
`CompileCommand.txt` : In this file you should put the exact command you used

to compile your program with g++. This will vary depending on exactly how you connected your files together and will let the TA run your code more easily. `Confession.txt` (optional) In this file, you can tell the TA about any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit. On the other hand, it also may lead the TA to notice something that otherwise he or she would not.