10 points.
Due: Monday, December 13, 2021 at 11:59 pm

Section 1 Problem Statement

In this assignment, you'll develop code to help manage the International Rock, Paper, Scissors Society. The IRPSS has a rather unique approach to determining their hierarchy of members. Rather than holding regular tournaments, they allow new players to work their way into the hierarchy individually. A prospective Society member faces one established member near the middle of the road. If they win, they move on to face players who previously beat that first member. If they lose, the are sent to play members who previously lost. As the developers of this system, we won't raise questions about whether this is a very fair way to determine the hierarchy, as this approach conveniently mimics a binary search tree.

So, in this assignment, we will spend some time using C++ template features to develop a binary tree data structure for tracking the Society's hierarchy of members. We'll also practice using exception handling to settle matters when competitions get out of hand.

Section 2 Classes

Below, we'll describe each of the classes you should write to handle IRPSS hierarchies.
Note that for this program, a test `main.cpp` will be provided. For these last two programs (assignments 6 & 7), we will not use menu-based programs as we have in previous assignments.

The main focus of the programming effort this time will be in creating a binary tree data structure, which uses templating to allow re-use for different regional chapters of the IRPSS. We'll start off with writing very simple classes for members of the North American and South American chapters.
In both cases, a member will have an array of characters representing their "strategy" in matches. This is the order of moves they will do in an effort to win a best-of-n matchup. Each character has a value of `'r'`, `'p'`, or `'s'`.

Both classes will also override the < and << operators. Since these classes are very simple, you should implement them with their functions in the .h files.

There is one more important concern to discuss here. It turns out that some Rock, Paper, Scissors players form unofficial clubs, which often have fierce rivalries with one another. Two prominent clubs are the School of Rock and the Paper Tigers. Whenever two members of these opposing clubs face one another, a fight breaks out and they begin throwing punches instead of rock/paper/scissors. For the purposes of our program, Punches is a subclass of the exception type (provided in `Punches.h`). More details on how to handle these exceptions are given in the `IRPSSHierarchy` section.

(a) class NorthAmericanMember

Each member of the North American chapter should have a string for their name, and a vector of characters to store their strategy. In addition, they should store an int representing their strength in a fight, and two bools representing whether they are a member of the School of Rock or the Paper Tigers, respectively.

The public functions for `NorthAmericanMember` are listed below:

- `NorthAmericanMember(string _name, int _strength, vector<char> _strategy, bool _rockschool, bool _tiger)` (param'd constructor)
  As usual for parameterized constructor, this should set the values of the name, strength, strategy vector, and club membership variables for the class.

- `bool operator<(const NorthAmericanMember& defender)`
  Before you begin this comparison, check if one of the members is in School of Rock and the other in Paper Tigers (you can assume a member is never in both). If so, this function should throw a `Punches` object. Use the parameterized constructor, passing in `this->strength` as the first parameter, and `defender.strength` as the second parameter.

Otherwise, this operator continues on to simulate a match between players.
The North American chapter of the Society uses a best-of-5 system for its members. For this operator, then, you should run a loop of up to n iterations, where n is the length of the strategy vector(s). Note, you can assume that `this->strategy` and `defender.strategy` have the same length.
Inside this loop, you should compare corresponding entries of the strategy vectors. Each loop iteration is one "game" of rock, paper, scissors. As you might expect, 'r' beats 's', 's' beats 'p', and 'p' beats 'r'. If the two corresponding characters are the same, the game is a draw. Otherwise, the member who had the winning character wins the game.

The first player to win 3 games wins the match (as they have the best of 5). If you reach the end of the loop and neither player has three wins, then the player who won more games wins the match. If you reach the end of the loop and the players are tied in games won, then "defender" wins the match. If "defender" wins the match, return true (because "this" is less-than "defender"). If "this" wins the match, return false.

- `ostream& operator<<(ostream& out, const NorthAmericanMember& member)`
  There isn't a lot to be output here. First, output the member name. If the member is in School of Rock, then output a space and the string `"(School of Rock)"`. If the member is in Paper Tigers, then output a space and the string `"(Paper Tigers)"`.
  Finally, output an end-line character.

(b) class SouthAmericanMember

Like members of the North American chapter, each South American member should have a string for their name, a vector of characters to store their strategy, and two bools representing whether they are a member of the School of Rock or the Paper Tigers, respectively. In addition, they should store an int representing their strength in a fight, *as well as* an int representing their speed in a fight (members of this chapter tend to use both speed an strength in their fights, whereas the North American members to to just stand close and punch it out).

The public functions for `SouthAmericanMember` are listed below:

- `SouthAmericanMember(string _name, int _strength, int _speed, vector<char> _strategy, bool _rockschool, bool _tiger)` (param'd constructor)
  Like the NorthAmericanMember constructor, this should set the values of the name, strength, strategy vector, and club membership variables for the class. Of course, it should also set the speed.

- `bool operator<(const SouthAmericanMember& defender)`
  This is mostly the same as the North American version, with a few minor modifications. First, the South American chapter uses a best-of-seven system, so in this case the winner is the first to four wins. If you reach the end of the loop and neither player has four wins, use the same approach as before (player with more wins wins the match, or tie goes to defender).

  If the players are members of the opposing clubs, you should still throw a Punches exception. This time, use the third and fourth parameters for the Punches constructor, as follows:
  `Punches(this->strength, defender.strength, this->speed, defender.speed)`

- `ostream& operator<<(ostream& out, const SouthAmericanMember& member)`
  This should be the same as the output operator for NorthAmericanMember

(c) class IRPSSHierarchy

As mentioned above, this class will carry out the operations of a binary search tree. It should be a class template that accepts a single meta-parameter, which is the type of data to be stored. Each `IRPSSHierarchy` object should store a variable "member" of type `T`, where `T` is the type given by the meta-parameter (see the lecture slides for some examples of using the type within a class template). The class should also store two `IRPSSHierarchy` pointers, for "left" and "right." If you would like to simplify memory management, feel free to use C++'s smart pointers here.
Note: Because we are dealing with a class template, you may simply write all the code in the .h file. We will

not require an `IRPSSHierarchy.cpp` file.

The public functions for `IRPSSHierarchy` are listed below:

- `IRPSSHierarchy(T _member)` (param'd constructor)
  The constructor should take a variable of type T as parameter. It should store this parameter into its own "member" variable, and initialize its pointers to null.
- `Insert(T new_member)`
  To perform an insertion, the `IRPSSHierarchy` should compare its "member" variable to `new_member`. If `new_member < this.member`, then the `new_member` belongs on the "left." Otherwise, the new member advances to the "right". Once you've decided whether the new member goes to the left or right, check the corresponding pointer. If it is null, you should create a new `IRPSSHierarchy` object, passing in the `new_member` to its constructor. Else, call the Insert function through that pointer, passing along the new member to face the next player in the hierarchy.

  Don't forget, when you make a comparison, it's possible an exception will be thrown. You should use a try-catch block, as discussed in the previous lecture, to catch any Punches thrown. If you catch a Punches exception, first print out the string you get when calling the exception's built-in `what()` function.
  Now, in the event of a fight breaking out, the IRPSS rules state that the offending parties should move their business to a boxing ring reserved for such occasions at the nearest regional headquarters. If the new member, or "challenger" wins against the old member, or "defender," they simply replace the old member in the hierarchy and do not advance any further. If the new member loses, or the fight ends in a draw, then the new member loses the right to join the IRPSS, and will not be added to the hierarchy.
  Thus, the second step in handling a Punches exception is to determine the winner of the boxing match. The exception has two public `int` variables, called `challenger` and `defender`. Compare these, and apply the rules outlined above. If the challenger wins, replace the `IRPSSHierarchy`'s member variable with the new member, otherwise return without making any changes.
- `ostream& operator<<(ostream& out, const IRPSSHierarchy& hierarchy)`
  The output operator will perform what is known as a pre-order print. First, if the "left" pointer is not null, output the "left" `IRPSSHierarchy` to the stream. Then, output `hierarchy.member` (recall, the Member classes you wrote already have the << operator written). Finally, if the "right" pointer is not null, output the "right" `IRPSSHierarchy`.

Section 3 Main

As mentioned previously, you will not need to write a `main.cpp` file for this program. We will provide a `main.cpp` for testing your code.

Feel free to add more tests if there are specific things you want to test out. The provided `main.cpp` should cover most cases, but you may find some case you wish to test more thoroughly.

In addition, feel free to add your own "Member" classes for Society chapters on other continents, with different rule sets, to see how class templates can be used to perform the same operations with many different classes.

Section 4 Submission

Submit a zip file, which contains header files `IRPSSHierarchy.h`, `NorthAmericanMember.h`, and `SouthAmericanMember.h`.
You are **not** required to submit `main.cpp` or `Punches.h`, as we will use our own copies of those during grading.
You are **not** required to submit a `IRPSSHierarchy.cpp` file, since IRPSSHierarchy is implemented as a class template.

Your zip file should be named *NetID*`_asg6.zip`, where *NetID* is your Net ID (the username you use on Canvas, *not* the 10-digit number on your wiscard).

In each header file, you should include a comment at the top of the file with your name and NetID.

Section 5 Rubric

Item	**Points**
Program builds and runs without compile errors	3
The two 'Member' classes output the correct data with the « operator	1
The 'Member' classes implement the < comparison operator correctly, including throwing exceptions	2
IRPSSHierarchy correctly implemented as class template, and can use with either 'Member' class equally well	1
IRPSSHierarchy allows new members to be added, and handles any exceptions thrown	2
IRPSSHierarchy correctly implements a pre-order traversal in the « operator.	1
Total	**10**