EGC-211P Section B

Take Home Assignment 5:

Submission Deadline to Dj: 24st September 23:59hrs

Create one file **rpskl.cpp**, to implement a class called **Tool**. It should have an int field called strength and a char field called type.

Create 5 more classes called Rock, Paper, Scissors, Spock, and Lizard, which inherit from Tool. Each of these classes will need a constructor which will take in an int that is used to initialize the strength field. The constructor should also initialize the type field using **'r' for Rock, 'p' for Paper, 's' for Scissors, 'k' for Spock, and 'l' for Lizard.**

These classes will also need a public method **bool fight(Tool&)** that compares their strengths in the following way (Rock paper scissors - Wikipedia), as listed in the table:

Rock-Paper-Scissors-Spock-Lizard;
+ means that the row player "beats", – means "is beaten" and O means tie

Opponent / Row player	rock	paper	scissors	Spock	lizard
rock	O	–	+	–	+
paper	+	O	–	+	–
scissors	–	+	O	–	+
Spock	+	–	+	O	–
lizard	–	+	–	+	O

For example: rock beats scissors, and lizard, paper folds (beats) rock and Spock, Scissors cuts (beats) paper and lizard, Spock beats rock and scissors, and lizard eats (beats) paper and Spock.

In our exercise, we have a slight variation, as follows:

A tool, when fights a stronger tool, halves its strength temporarily before fighting.

A tool, when fights a weaker tool, doubles its strength temporarily before fighting.

The strength field should not change in the *fight method*, which returns true if the original class (caller) wins in temporary strength and false otherwise.

You may also include any extra auxiliary functions and/or fields in any of these classes.

Implement Tool, Rock, Paper, Scissors, Spock and Lizard, and their fighting rules.

Marks will be cut if any of the following points are violated:

1. Should not have any duplicate code – fields or methods -- (checking for proper use of inheritance)
2. Data members are all private members of Tool class. Derived classes do not have any data member.
3. Setting strength should only happen through constructor.
4. Proper use of virtual functions (https://www.w3schools.com/cpp/cpp_virtual_functions.asp).
5. Tool class should not be instantiable (https://www.geeksforgeeks.org/cpp/pure-virtual-functions-and-abstract-classes/).

Split rpskl.cpp into .h and .cpp

Include meaningful code comments.

All the methods should receive and return parameters in terms of reference objects (not pointers, not values). You may store pointers internally in a class.

Entity to Play the game

Define a class Player (create player.h and player.cpp).
This class contains a Tool* attribute.
Tool* points to any one of the five types of tools.
Since a Player uses a Tool, this relationship is **association/aggregation**.

The Player class has the following additional fields (with corresponding getters and setters):

- id (an integer)
- name (a string)

Important notes about id:

- The id is internally generated by the code when a Player instance is created.
- Therefore, there must be no setter for id.
- Every player must have a unique id (no two players can share the same id).

The Player class must provide a public method:

choose(char type, int strength)

This method dynamically creates the corresponding Tool for the player to use in a fight.

A proper destructor must be defined to ensure that there are no memory leaks. Finally, create main.cpp with the following initial test. By this point, your code should pass the following test:

```
Player p;

Rock r2(3);

p.choose('s', 15).fight(r2); // p should win in this round against rock r2 because of its scissor strength
```

Define a class Game (create game.h and game.cpp).

This class is associated with two Player objects.

The Game class also has the following fields:

- name (a string)
- time (a string containing date and time details)
- winner (a pointer to the winning player, initialized as NULL)

Additionally, it has a public method show() to display the above game details.

Player registration

- The class provides a public method:

void registerPlayer(Player&)

This method registers a player into the game.

- The game can register at most two players. If a third player attempts to register, it is ignored.

Verification of registration

The Game class must provide a public method:

bool verifyRegistration()

This method checks whether both players have **different IDs**.

- If the IDs are different, the game can proceed.
- If the IDs are the same, both players are invalidated, and the game should not continue.

Gameplay

Game should have a method choose:

Player& choose()

The first call to choose should allow the **first registered player** to make a move, and then the subsequent calls should allow the alternate player to make move.

The Game class has the following public method:

```
void playGame(char type1, int strength1, char type2, int strength2) {
    choose().choose(type1, strength1).fight(choose().choose(type2, strength2));
    registerWinner();
}
```

Important notes:

- **Do not modify** the above implementation of playGame.
- **Do not create any alternate implementation** of this method.

The Game class also has a protected method:

void registerWinner()

This method assigns the winning player as the game's winner.

Winner mechanism

After a fight:

- The participating **tools** must know which tool won and which lost.
- To achieve this:
    - Introduce a winner field in both the Tool and Player classes (in addition to the Game).
    - When a fight occurs:
        - The winning tool should set its winner flag to true.
        - The losing tool should set its winner flag to false.
        - The corresponding player should set its winner flag to true if its **last used tool** won.
        - Once the players know who won and who lost, the Game must set its own winner.

Design guideline: Each class/object should manage its own responsibilities.

Double-checking the result

The Game class must have a public method:

bool doubleCheck()

This method verifies that:

1. The winning player has a winning tool.
2. The losing player has a losing tool.

The method must return the result of a **bitwise XOR** between the two conditions, which returns true ***only if there is a winner***:

```
bool doubleCheck() {
    bool f = // a winner player has a winner tool – check for first registered player
    bool s = // a loser player has a loser tool – check for second registered player
    return f ^ s; // logical XOR – this will be false if it is a tie
}
```

Example scenario: Auction House Selection

Refer to t**he `Auction House Selection' section in the Wikipedia** Rock paper scissors - Wikipedia:

Two players Christie international and Sotheby holdings fight for Cézanne's *Large Trees Under the Jas de Bouffan*.

main.cpp:

```
Player ci("Christie_international"), sh("Sotheby_holdings");
Game g("auction_for_Large_Trees");

g.registerPlayer(ci);

g.registerPlayer(sh);

g.playGame('r', 23, 'p', 13); // Sotheby_holdings wins
cout << ci.isWinner() << endl; // 0
cout << sh.isWinner() << endl; // 1
cout << ci.getLastUsedTool().isWinner() << endl; // 0
cout << sh. getLastUsedTool ().isWinner() << endl; // 1
```

Driver Program

Create the final main.cpp as a **command-driven game**.
Remove all previously written main() functions.

The input/output flow in main() should follow this structure (**blue text** = internal actions):

```
Enter game: <give game name> (create game)
Register Player1: <player1 name> (create player1 and register)
Register Player2: <player2 name> (create player2 and register)
... verifying registration (call to verifyRegistration) ... OK (if verifyRegistration is
true, NOK otherwise)
(if NOK in the previous step) <program ends>
(if OK in the previous step)
Play with: <type1> <strength1> <type2> <strength2> (call to playGame)
... <player name> own! (cout the winner name) OR ... its a tie! (in case no one wins)
... double checking (call to doubleCheck) ... OK (if doubleCheck is true, NOK otherwise).
(if OK in the previous step)
Congratulations! <program ends>
(if NOK in the previous step)
<repeat from Play with:  >
```

Result:

If your code is correct, Congratulations will be printed only **once**.

Rock-Paper-Scissors-Spock-Lizard *Minus One*:

We now want to implement a **famous Korean variant** of the earlier game:
Rock-Paper-Scissors-Spock-Lizard Minus One
(see *Variation* section in Rock paper scissors - Wikipedia).

Rules

- Both players throw with **both hands simultaneously**.
- Each player then chooses **one hand to remove**.
- The winner is determined by the **remaining hands** in play.
- If it's a tie, the game is replayed.

Implementation

- Create a class **`KoreanGame`** derived from **`Game`**.
  - It should preserve the player registration mechanism as before.
  - However, in this version, the players are **`KoreanPlayer`** objects, not `Player`.
- Each **`KoreanPlayer`** uses **two hands** during the game.
- You are free to design the `KoreanPlayer` class either by **inheriting from `Player`** or by **using composition**.

Here are the relevant methods of the classes:

```
// class KoreanPlayer:
Tool& choose(char leftOrRight, char type, int strength); // addition param indicating left
hand or right
```

```
// class KoreanGame:
void registerPlayer(Player&); // no change
bool verifyRegistration(); // no change
Player& choose(); // no change, still it should give alternate player on each call

void playGame(char type1_1, int strength1_1, char type1_2, int strength1_2,/* 1st four param
for the two hands of first registered player*/
    char type2_1, int strength2_1, char type2_2, int strength2_2/* 2nd four param for the two
hands of second registered player*/,
    char player1_hand /*player1 remove which hand */, char player2_hand/*player2 remove which
hand */) {
    choose().choose('l', type1_1, strength1_1).fight(choose().choose('l', type2_1,
strength2_1));
    choose().choose('l', type1_1, strength1_1).fight(choose().choose('l', type2_2,
strength2_2));
    choose().choose('r', type1_2, strength1_2).fight(choose().choose('l', type2_1,
strength2_1));
    choose().choose('r', type1_2, strength1_2).fight(choose().choose('l', type2_2,
strength2_2));
    remove_hand(player1_hand, player2_hand);
    registerWinner();
}

void registerWinner(); // no change
void remove_hand(player1_hand, player2_hand); // remove the hands and decide the winner based
on the remaining hands
```

Driver program: Now convert the main() to a command line program that gives the user a choice which game to play.

Here is a sample trace:

```
=======================================
              WELCOME
=======================================
Enter which game to play:
0. Vanilla (it is the earlier version, without Korean specialization)
1. Minus One
Enter Choice: 1
Enter game: <give game name> (create game)
Register Player1: <player1 name> (create player1 and register)
Register Player2: <player2 name> (create player2 and register)
... verifying registration (call to verifyRegistration) ... OK (if verifyRegistration is
true, NOK otherwise)
(if NOK in the previous step) <program ends>
(if OK in the previous step)
Play with: <type1_1> <strength1_1> <type1_2> <strength1_2> <type2_1> <strength2_1> <type2_2>
```

```
<strength2_2> <player1_hand> <player2_hand> (call to playGame)
... <player name> own! (cout the winner name) OR ... its a tie! (in case no one wins)
... double checking (call to doubleCheck) ... OK (if doubleCheck is true, NOK otherwise).
(if OK in the previous step)
Congratulations! <program ends>
(if NOK in the previous step)
<repeat from Play with:  >
```

File submissions:

rpskl.h, rpskl.cpp

player.h, player.cpp

KoreanPlayer.h, koreanPlayer.cpp

game.h, game.cpp

KoreanGame.h, koreanGame.cpp

main.cpp

Makefile

Reading Assignment:

Learn makefile usage and create it for your submission: GNU Make - An Introduction to Makefiles

More test cases will be uploaded on LMS.