CSE 232

Programming Project 03

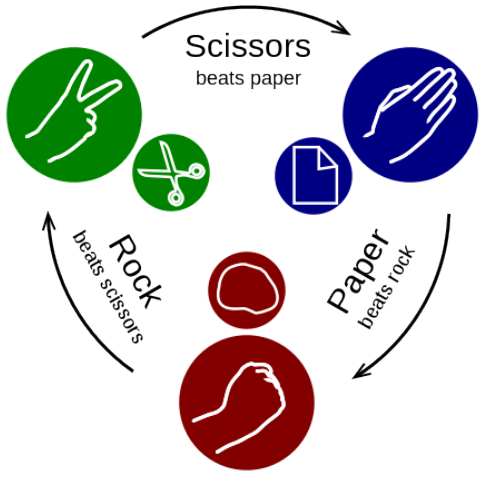
Assignment Overview
This assignment will give you more experience on the use of loops and conditionals, and introduce the use of functions.

Background
Games have been interesting to computer scientists for, well since there was computer science. Not shoot-em-up, blood-and-gore games, those are just silly. We are talking about games that can be analyzed and their properties understood. In so doing we can design algorithms that do a better job solving a game, and perhaps apply those algorithms to other problems.

Anyway, let's play the game rock-papers-scissors.

Rock-Paper-Scissors
Let's call it RPS, I am a CS person afterall. RPS is a children's game, a two player game that involves the two players choosing one of three hand gestures to see who wins a round of play, see https://en.wikipedia.org/wiki/Rock-paper-scissors . If you read the link to associated Ken games, https://en.wikipedia.org/wiki/Sansukumi-ken, you would see that it's origins are less than child-like. It seems to have started as a game from Asia (in China, in Japan), one of a series of such games, that had to do with brothel's and drinking games. How quaint!

The Rules
The rules are pretty simple, as illustrated by the Wikipedia page image shown below.



In this two-player game, each player can make one of three hand gestures. The players simultaneously (usually after the count of three) show their hand gesture and determine a winner. One of two outcomes is possible:

- both players select the same hand gesture, resulting in a tie.
- one player wins the round. The rules as shown are:
    - rock wins over scissors
    - scissors wins over paper
    - paper wins over rock

Project Description / Specification

Input and Output
We are going to use integers here to mean different things in different contexts so pay attention:

- The legal player numbers are 1 or 2 (player1 or player2)
- For game moves: 1 is rock, 2 is paper, 3 is scissors.
- For scoring purposes: 0 is a tie, 1 means player1 won, 2 means player2 won.

main
We are going to write some strategies to play RPS, and then run those strategies to see how they fare. We provide a skeleton file to start with that contains a predefined main. You can begin by editing this file and filling in the functions in the provided space. This updated file, with the unchanged pre-defined main, is what you will turn into Mimir**. Do not change the main program**! It will mess with the tests and you won't get the credit you deserve! The TAs will check for this during grading. All you do is provide the required functions.

The main program uses a switch statement to determine which functions to run. The first argument to main is the switch input value. Subsequent input values depend on the input needs for the function selected.

We will provide you with **some starter code** that has the main function written without the functions that you can fill in.

Functions
The 3 functions take the same arguments, even if that particular strategy doesn't need to use the information in all the arguments. Makes calling the functions more standard.

function: `strategy1`:

- return is `int`, the move rock(1), paper(2) or scissors(3) you select for the next round.
- Arguments are four integers in the following order:
    - `player` : which player are you (1 or 2).
    - `previous_result` : 0 if a tie otherwise 1 or 2 indicating player1 or player2 won the last round.
    - `previous_play`: 1, 2 or 3 if this player (the player of the first argument) played rock(1), paper(2) or scissors(3) in the last round.
    - `opponent_previous_play`: 1, 2 or 3 if the **other** player played rock(1), paper(2) or scissors(3) in the last round.

This function represents a simple strategy. It cycles through rock(1), paper(2) or scissors(3) in that exact order. Thus if your `previous_play` was: 1 you return 2; if 2 returns 3; if 3 returns 1;

function: `strategy2`:

- return is `int`, the move rock(1), paper(2) or scissors(3) you select for the next round.
- Arguments are four integers in the following order:
    - `player` : which player are you (1 or 2).
    - `previous_result` : 0 if a tie otherwise 1 or 2 indicating player1 or player2 won the last round.
    - `previous_play`: 1,2 or 3 if this player played rock(1), paper(2) or scissors(3) in the last round.

    - `opponent_previous_play`: 1,2 or 3 if the other player played rock(1), paper(2) or scissors(3) in the last round.

This is the stick-or-switch strategy. If you **won or tied** the last round return your previous play. Otherwise, return your opponent's previous play.

function: `strategy3`:

- return is `int`, the move rock(1), paper(2) or scissors(3) you select for the next round.
- Arguments are four integers in the following order:
    - `player` : which player are you (1 or 2).
    - `previous_result` : 0 if a tie otherwise 1 or 2 indicating player1 or player2 won the last round.
    - `previous_play`: 1,2 or 3 if this player played rock(1), paper(2) or scissors(3) in the last round.
    - `opponent_previous_play`: 1,2 or 3 if the other player played rock(1), paper(2) or scissors(3) in the last round.

This is the stick-or-win strategy. If you **won or tied** the last round, return your previous play. Otherwise, return a move that would **beat** your opponent's previous play.

function: `score_round`:

- return is `int`. 0 for tie, otherwise 1 or 2 (which player won player1(1) or player2(2)).
- Arguments are two integers:
    - player1 move and player2 move (in that order). Each has a value of rock(1), paper(2) or scissors(3), the play made by each player.

Which player (1 or 2) won this round, or a 0 if it was a tie.

Deliverables

`proj03/proj03.cpp` -- the skeleton.cpp code with your functions added in. (*remember to include your section, the date, project number and comments in this file*).

1) In mimir this is Project 03 - RPS
2) Mimir will start with the skeleton code in the IDE

Hints:

First, you can ***write as many functions as you like*** over and above the ones I have specified.

- I in fact did write other, supporting functions.
- make sure you write the requested functions exactly as specified. They will be tested individually according to that specification.
- Use the starter code to get going. Using that code you can test each function by providing, as input, which test case you are testing.

Attack/identify the different subproblems in the description. For example:

- write the function for `strategy1`. Check it using the provided to confirm that it works
- write the function for `strategy2`. Confirm it
- etc.

Some questions to ask yourself

1. Which strategy appears to be the best?
2. Do the starting conditions matter?
3. Are there any simple changes I could make to improve the strategy?
    a. does sticking on a tie matter?