CS246 Fall 2020 Project – Straights

Due Date 1: Wednesday, December 2, 2020, 5:00 pm
Due Date 2: Wednesday, December 16, 2020, 11:59 pm

- DO NOT EVER SUBMIT TO MARMOSET WITHOUT COMPILING AND TESTING FIRST. If your final submission doesn't compile, or otherwise doesn't work, you will have nothing to show during your demo. Resist the temptation to make last-minute changes. They probably aren't worth it.
- This project is intended to be doable by one person in two weeks. Because the breadth of students' abilities in this course is quite wide, exactly what constitutes two weeks' worth of work is difficult to nail down. Some students may finish quickly; others won't finish at all. We will attempt to grade this assignment in a way that addresses both ends of the spectrum. You should be able to pass the assignment with only a modest portion of your program working. Then, if you want a higher mark, it will take more than a proportionally higher effort to achieve, the higher you go. A perfect score will require a complete implementation. If you finish the entire program early, you can add extra features for a few extra marks.
- Above all, MAKE SURE YOUR SUBMITTED PROGRAM RUNS. The markers do not have time to examine source code and give partial correctness marks for non-working programs. So, no matter what, even if your program doesn't work properly, make sure it at least does something.

In this project, you will implement a card game, Straights. This section will describe the card game as it is played with cards, and the next section will describe how it is to be computerized.

1 Objectives

Straights is a four-player game. The objective is to get the fewest number of points among the players. The game ends when one player accumulates 80 points or more, and the player with the lowest score is declared the winner. If the lowest score is a tie, then all players with that score win.

2 The Deck

Straights uses a standard 52-card deck, without the jokers. Each player is dealt 13 cards at the beginning of each round. In this game, the Ace is considered to be the lowest card in each suit (rank of 1), while the King is the highest (rank of 13).[1]

3 Gameplay

3.1 Legal Plays

Immediately following the deal, the player with the 7 of spades goes first. This player must play the 7 of spades in the centre of the table. After the 7 of spades, the players take turns to play cards on the table. At this point, the players must play cards that constitute *legal plays*. The following cards are legal:

- A 7 of any suit. This card starts a new pile on the table.
- A card with the same suit and *adjacent rank*[2] as another card that has already been played. It must be played on the pile of the appropriate suit. (Note that the "pile" is spread across the table, so that play can proceed at either end.)

[1] A Jack has a rank of 11, while the Queen has a rank of 12. The *rank* of all other cards is their numeric value e.g. 2 has a rank of 2.
[2] A card has *adjacent rank* if its face value is one more or one less than the rank of card under consideration. The King and Ace of a suit are not considered to be adjacent ranks to each other.

For example, if the 7 of spades is the only card on the table, then the legal plays are: the 7 of diamonds, the 7 of hearts, the 7 of clubs, the 8 of spades, and the 6 of spades. Once the 8 of spades is played, the next legal plays are: the 9 of spades, the 6 of spades, the 7 of diamonds, the 7 of hearts, and the 7 of clubs. In this way, you can add cards to either end of the suit "pile" so long as there are no gaps.

3.2 Discards

When a player has no legal plays, they must then choose a card in their hand, and place it face down in front of them. This is a *discard*. Note that if a player has at least one legal play in their hand, then they must make a legal play; they may not discard in this case.

4 Scoring

The round ends when all the cards have either been played or discarded. For each player, their score for the round is the sum of all the ranks of the players discards. Jacks, Queens, and Kings count as 11, 12, and 13 points, respectively. For example, if a player discarded an Ace, a Six, and a King, the number of points would be $1 + 6 + 13 = 20$.

Each player's game score is of the sum of their scores in each round. If no player has accumulated 80 or more points at the end of a round, then the deck is reshuffled and another round begins.

5 Computer Straights

Question: What sort of class design or design pattern should you use to structure your game classes so that changing the interface or changing the game rules would have as little impact on the code as possible? Explain how your classes fit this framework.

5.1 Command-Line Parameters

The user can provide an optional integer argument *seed*, used to initialize the random number generator when starting the program:

```
./straights 44
```

This allows for pseudo-random shuffling. See the provided `shuffle.cc` for an example of declaring a random number generator that is used to randomly shuffle cards in the deck. The random number generator is seeded *once* with the (possibly global variable) `seed`, and then used in `std::shuffle`. Games that are started with the same seed value have the same sequences of deals, so long as the deck of cards is restored between rounds to its previous state before being shuffled again.

5.2 Invite Players

At the beginning of the program, prompt the user with the following message:

```
Is Player<x> a human (h) or a computer (c)?
>
```

where `<x>` is the id number representing the player being initialized. The user then types either `h` or `c` to set the desired player type. Repeat this step for each of the four players.

5.3 Shuffling and Dealing

Initially, the cards in the deck must be in the following order :

```
AC 2C 3C ... TC JC QC KC AD 2D ... QD KD AH 2H ... QH KH AS 2S ... QS KS
```

At the beginning of every round, shuffle the deck once. After the shuffle, deal out the cards such that the first 13 cards belong to Player 1, the next 13 cards belong to Player 2, the next 13 belong to Player 3, and the last 13 cards belong to Player 4.

When the round is over, the deck is restored to its previous state before shuffling it. (If it was reset to the original state, then the players would end up with the exact same hands of cards again.) For example, if after shuffling the deck has the order 6S 2S 4C 2H 6C AD JD 8D 2D AS JH 7S AH 2C 3H 3S KD 5C 5H 9C 3D JC 6H AC TH 4D 5D TD 7D 4S 7H 4H TC 9D JS KC 8S KS TS QC 9H 7C 9S QS 5S 8C 8H KH 6D QD QH 3C, then before it is shuffled for the next round, the cards are returned to their previous order of 6S 2S 4C 2H 6C AD JD 8D 2D AS JH 7S AH 2C 3H 3S KD 5C 5H 9C 3D JC 6H AC TH 4D 5D TD 7D 4S 7H 4H TC 9D JS KC 8S KS TS QC 9H 7C 9S QS 5S 8C 8H KH 6D QD QH 3C. **This way, the results will be repeatable, which will make your testing and demonstrations easier.**

5.4 Gameplay

5.4.1 Start

The game starts after the shuffle and the deal. The four players take turns to play their cards. First, print the following line (regardless of whether the first player is a human):

```
A new round begins. It's Player<x>'s turn to play.
```

where <x> depends on who has the 7 of spades.

5.5 Players

Question: If you want to allow computer players, in addition to human players, how might you structure your classes? Consider that different types of computer players might also have differing play strategies, and that strategies might change as the game progresses i.e. dynamically during the play of the game. How would that affect your structure?

Question: If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?

5.5.1 Human Player

Whenever it is a human player's turn, print the following 8 lines:

```
Cards on the table:
Clubs:<list of clubs>
Diamonds:<list of diamonds>
Hearts:<list of hearts>
Spades:<list of spades>
Your hand:<cards in your hand>
Legal plays:<legal plays in your hand>
>
```

Each of list of cards in a specific suit is an ordered sequence of all the ranks in that suit (e.g., 6 7 8 9 T J Q) that have already been played.

`<cards in your hand>` and `<legal plays in your hand>` are lists of cards in the player's hand, where each card is in the form `<rank><suit>` (e.g. `7S`). Print the cards in the same order that they appear in the deck. Do not rearrange the cards. Every list of cards, except for that produced by the `deck` command, starts with a space character, and a single space separates each card from the next. There is no space after the final card in the list. If there are no legal plays, then the list of cards consists of an empty string i.e. print `Legal plays:\n`.

The program then waits for the user to enter a command.

5.5.2 Commands

There are 5 valid commands for the human player in this game:

Command	Description

`play <card>`	Play the specified card. You may assume that the `<card>` has valid syntax (i.e., `<rank><suit>`, such as 7C), and that the specified card is in the players hand. However, it might not be a legal play.
	
	If the play is legal, print:
	
	`Player<x> plays <card>.`
	
	and proceed to the next player. Otherwise, print:
	
	`This is not a legal play.`
	`>`
	
	and do not proceed to the next player until a legal play is made.

`discard <card>`	If the player has no legal plays, discard the specified card from the player's hand into the player's discard pile. Again, assume that the `<card>` has valid syntax and that the `<card>` is in the players hand. For test purposes, the value of the card is printed even though that deviates from the normal straights game play.
	
	`Player <x> discards <card>.`
	
	Otherwise, print the following error message:
	
	`You have a legal play. You may not discard.`
	`>`

`deck`	Print the contents of the deck in order, 13 cards per line. For example:
	
	`TS 2D 3S KH 3H 2C 5D TC 8S TD AC KC QH`
	`4D JH 6H JC KD 8C 7D TH 4H 9S 6S 4S KS`
	`7S 7C QD 6C 2H 6D 3C 9C 5H 3D AD 5S 8H`
	`QC 2S 8D JS QS AS JD 4C 7H 9D 5C AH 9H`
	`>`
	
	(This command is not part of the straights game. It is provided to help both you and us with the testing and debugging of your program.)

`quit`	Terminate the program immediately.
`ragequit`	Filled with anger, a human player decides to leave! Print the following message: `Player <x> ragequits. A computer will now take over.` Replace the current human player with a computer player, and resume the game.

While your handling of the commands is not required to be robust, you will find it in your best interest to make it at least somewhat robust so that a game doesn't have to be started over from the beginning if a player enters an invalid command or card value.

5.5.3 Computer Player

If it is a computer players turn to play, print either one of the two lines, as appropriate:

```
Player<x> plays <card>.
```

or

```
Player<x> discards <card>.
```

Proceed to the next player.

The computer player that you will implement is very simple. It always makes the first legal play in its hand. If there are no legal plays, the first card in its hand is discarded.

5.6 Scoring

When all of the cards have been played, the round ends. At this point, print the following lines for each of the players:

```
Player<x>'s discards:<list of discards>
Player<x>'s score: <old score> + <score gained> = <new score>
```

`<list of discards>` is a list of the cards the player discarded in the current round. Print the cards in the same order that they were discarded.

If one player has accumulated 80 points or more, the game ends. (Note that there is no prompt to replay again. Instead, you would just run the executable again.) The player with the lowest score wins. Print the following victory message for the winning player(s):

```
Player<x> wins!
```

If multiple players tie for the win, print the above message for each winner.

If no one exceeded the score limit, then reshuffle the deck and begin another round.

6 Example

This is an example game of Straights, including only a few interesting snippets. The *[...]* indicates where the rest of the game has been excluded.

```
Is Player1 a human (h) or a computer (c)?
>h
Is Player2 a human (h) or a computer (c)?
>c
Is Player3 a human (h) or a computer (c)?
>c
Is Player4 a human (h) or a computer (c)?
>c
A new round begins. It's Player4's turn to play.
Cards on the table:
Clubs:
Diamonds:
Hearts:
Spades:
Your hand: 7C QS KH 5S 6S JD AD 7S 8D TD 6D TC KD
Legal plays: 7S
>Player4 plays 7S.
Cards on the table:
Clubs:
Diamonds:
Hearts:
Spades: 7
Your hand: 4H 5D 2D JC 8H QH 2H 6C 9H 9D 9C AS 2C
Legal plays:
>deck
4H 5D 2D JC 8H QH 2H 6C 9H 9D 9C AS 2C
JS QD 5C 8C 4D 5H KC 7D KS AC JH 6H 7H
3H 4C 3S 3D AH QC 3C TH 4S 9S 2S 8S TS
7C QS KH 5S 6S JD AD 7S 8D TD 6D TC KD
>discard AS
Player1 discards AS.
Cards on the table:
Clubs:
Diamonds:
```

```
Hearts:
Spades: 7
Your hand: JS QD 5C 8C 4D 5H KC 7D KS AC JH 6H 7H
Legal plays: 7D 7H
...
Cards on the table:
Clubs: 7
Diamonds: 7
Hearts:
Spades: 7 8
Your hand: 4H 5D 2D JC 8H QH 2H 6C 9H 9D 9C 2C
Legal plays: 6C
>play 6C
Player1 plays 6C.
...
>ragequit
Player1 ragequits. A computer will now take over.
Player1 discards 4H.
...
Cards on the table:
Clubs: 6 7 8 9
Diamonds: 5 6 7 8
Hearts: A 2 3 4 5 6 7 8 9 T J
Spades: A 2 3 4 5 6 7 8 9 T J Q
Your hand: 3C
Legal plays:
>Player3 discards 3C.
Player1's discards: 9D QD KH KD 2D
Player1's score: 16 + 49 = 65
Player2's discards: 5C JD QH AD 3D KC
Player2's score: 40 + 45 = 85
Player3's discards: AC 4C 3C
Player3's score: 15 + 8 = 23
Player4's discards: KS QC 2C 4D TD TC JC
Player4's score: 53 + 62 = 115
Player3 wins!
```

7 Testing Mode

You may find it useful to implement another optional command-line argument that represents the name of a file containing the saved state of a game so that you can demonstrate specific scenarios more easily to the grader and for your own testing purposes.

Grading

Your project will be graded as follows:

Correctness and Completeness	60%	Does it work? Does it implement all of the requirements?
Documentation	20%	Plan of attack; Final design document.
Design	20%	UML; good use of separate compilation, good object-oriented practice; is it well-structured, or is it one giant function?

Even if your program doesn't work, you can still earn marks associated with the Documentation and Design components of the project.

If Things Go Well

If you complete the entire project, you can earn up to 10% extra credit for implementing extra features. You should provide some way of playing the game with and without enhancements, as this will allow project assessors to accurately determine that you met all of the base requirements. Recompiling is **not** permitted.

The following is a list of possible enhancements that you could develop. **However, don't attempt to implement this until you've got the base game working.**

- House rules. You could invent your own house rules.
- Smarter computer players. Right now, the computer player makes the first choice of legal moves, and the first choice of discards. There is, however, quite a bit of strategy possible in terms of what is played and what is discarded.
- Better display. Even with just ASCII, the "table" could be drawn more nicely than a list of cards and moves. Or if you're feeling bold, you could implement a GUI.

To earn significant credit, enhancements must be algorithmically difficult, or solve an interesting problem in object-oriented design. Trivial enhancements will not earn a significant fraction of the available marks.

Due Dates

Due Date 1: Due on due date 1 is your plan of attack, `plan.pdf`, and initial UML diagram, `uml.pdf`, for your implementation of Straights on Marmoset, `CS246_PROJECT`.
Due Date 2: Due on due date 2 is your actual implementation of Straights. All `.h`, `.cc` and any other files needed for your project to compile and run should be included in `straights.zip`. The ZIP file must contain a suitable `Makefile` such that typing `make` will compile your code and produce an executable named `straights`. In addition, your `straights.zip` must also contain `demo.pdf` and all necessary files to run your demo (such as saved input, either with a fixed seed or loaded game files, but not the program executable, which will be compiled from the files submitted in `straights.zip`), `uml-final.pdf` and `design.pdf` to the appropriate place on Marmoset, `CS246_PROJECT`.

If you have implemented any bonus features, make sure that your submitted demo plan includes a list of all of the described bonus features, and how to run the game to see the effect of the bonus features.

See **project_guidelines-online.pdf** for instructions about what should be included in your plan of attack and final design document.