Introduction
This CS246 final project creates a text-based straights card game that you can play from your command line. A deck of 52 standard playing cards is distributed amongst 4 players who can be human or computer players. The game ends when a player has accumulated 80 points or more.

Overview (describe the overall structure of your project)

Main creates a new game instance and new observer. Main handles optional integer seed input, also has error handling for invalid input. Main handles logic for while the game is not done, continue playing a new round. If done, print results.

The entire Straights game is controlled by the Game class. The functions startGame(), setPlayers(), and dealCards() initialize the game settings. The functions newRound(), clearRound(), findFirstPlayer(), firstTurm(), playTurn(), facilitate each round and determine whether a round is over. The functions isRoundDone() and isGameOver() handle the logic for controlling game flow. printRoundResults() and printWinners() handle printing output of the game and formatting the results. The rest of the functions are accessor and mutator methods. This class also uses several functions from the Player class to handle logic.

The Deck class manages the shuffling function and printing function.

The Card class is an object created to hold each card. It has functions to retrieve the suit and value of a card, as well as overloaded functions to read in a card and print out a card in the correct format. There is also an overloaded comparison operator to compare whether 2 cards are equal.

The Table class is a concrete subject. It's a map with 4 elements. Each element's key is one of the 4 suits, and each element's value is a vector of played cards so far in the game. The addCard function handles when a player plays a card, and adds it to the map object. It also rearranges the card in ascending rank order as specified. This class also has additional functions to clear the table after each round, check whether the table is empty, and return the vector of cards per element.

The TableObserver class is part of the observer design pattern (more detailed discussion in next question). This prints the table before each round

The Player class is my abstract base class, since there are no players of type Player. The subclasses are types HumanPlayer and ComputerPlayer. The constructor for each player type passes in their id as well as a pointer to the table. Most of the player logic and actions that need to be implemented are consistent between a human and computer player, therefore most functions are located in the Player abstract base class. There are many getter functions to retrieve information about each player. There are also logic-handling functions, such as has7S(), that are used in other classes to determine starting players. There are functions to handle player actions such as playCard, discardCard, updateLegalMoves().

Changes to UML Diagram

Table + TableObserver Class

- replaced my concrete observer class GameDisplay with TableObserver
- replaced my concrete subject classRound with Table.

Game Class

- Added more private attributes such as a pointer to Table
- Added accessor and mutator functions for my private attributes
- Removed handleRageQuit() and handled it in my playTurn() method instead

I decided to remove my Round class and rename it as Table because I realised that the Round was storing the current cards on the table as well as the current player, which is essentially the game state. But since my observer doesn't need to know the current player when printing the current cards, I decided to name it Table to better reflect the purpose it serves, which is to print to the command line which cards are currently on the table.

Design

Observer Pattern

I used the observer design pattern to print the list of played cards before each computer or human turn. The concrete subject is the Table class, which contains information about which cards have been played so far, and it's state will change after a player completes an action each turn. The concrete observer is the TableObserver class which prints the current table before each turn. In my playTurn function, I call the notify() function prior to prompting the user to make an action. I chose to use the observer pattern because it is an efficient method to implement this feature of displaying the current game state to the player before each turn.. Additionally, this provides higher resilience to change if I were to implement a graphical view of my game because I could code another graphics observer class and make minimal changes to my existing code This is discussed more in the resilience to change section.

Strategy Pattern

Another design choice I made was to use the strategy pattern to design my player classes. Player is my abstract base class while ComputerPlayer and HumanPlayer are subclasses of Player. The player types differ in how they choose which card to play, therefore the only virtual function is choosePlay(), which differs in logic depending on whether it's a human or computer. This function returns a "command", which is a pair of string and Card, to the Game Class to carry out the action. The choosePlay() function computes a player's legal moves based on the current table; then it plays the first valid card or discards the first card in its hand. For a human, the function receives user input to choose a play. These functions also verify that the play inputted by a human is a valid move by searching for the card in the legalMoves vector of Cards.

This design choice also simplifies how I would deal with any human players who ragequit. In my program, I copy the human player's information into a new ComputerPlayer object and replace that player in my vector of Players in my Game class. Since choosePlay() is an overloaded function, I don't need to change any code for how a Player plays their turn.

Coupling and Cohesion

In order to maintain low coupling, I used forward declarations as much as possible to reduce compilation dependencies and prevent include cycles. This means I only included the dependent header files in a class's implementation file and only forward declared it in the header file.

I also designed my program to have high cohesion, as every class achieves a specific purpose and all the functions are highly related to the class. For example, my Game class contains all the functions needed to control the game flow of starting new rounds, calculating and printing results, and determining when to end the game. My Player class contains functions related to the logic of game rules and management of a player's personal hand. Most functions relate to updating the player's hand, discard pile, and legal moves, as well as communicating to the game class which move the player has chosen each turn. My Table, Deck, and Card classes are pretty short and self-explanatory as to why the functions are directly related to the class.

Encapsulation

In order to follow good object-oriented design, I maximized encapsulation by making the attributes of my classes private and using accessor and mutator methods to modify or retrieve the values. Additionally, I chose to make my Player class functions protected instead of public because the only other class that uses Player functions is my Game class. So to maximize encapsulation, I just declared Game as a friend class so that my Player attributes can be accessed by Game but are still protected from other classes.

Memory Management

I implemented the RAII method by using shared pointers everywhere to simplify memory cleanup and prevent leaks. I tested and debugged my program using valgrind to ensure that all the memory was safely freed by my shared pointers and that there were no cyclic references with my shared pointers.

Error Handling

I use a try-catch block in my main function to prevent invalid input for the optional command line argument for seed. Additionally, in my choosePlay() function for HumanPlayer, I ensure that the player enters a valid play. If the play isn't valid, the command is ignored and the player is prompted to re-enter another command. This allows for a better game experience since the program won't exit immediately upon an invalid command, but instead allows for errors such as typos which are very likely.

<u>Resilience to Change</u>

<u>Add graphics</u>
I would add a Graphics Observer class that displays the current cards laid on the table as well as an individual player's hand, possible plays, and discard pile so far, only when it's their turn. My notify function would now have an integer parameter to indicate which player's turn it is currently so that the observer will display the correct set of cards. The current cards on the table will be consistent regardless of which player's turn it is.

<u>Change Game Rules</u>
This would require minimal changes to my program as I only need to update the updateLegalMoves() function in my Player class to handle new logic according to the new rules. I could also change the number of players, which would change how I deal cards and how many players are created in my Game class. I would replace all instances of 4 with the new number. Or in the future, I would create a header file with all global constants such as NUM_PLAYERS for better design. I would also have a global constant for MAX_SCORE so that it can be changed easily in case the program specifications change for what constitutes a done game.

<u>Answers to Questions</u>

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

For the design of my program, I used various techniques and strategies presented throughout this course to accommodate for potential changes to my game display and rules. I decided to use an Observer design pattern to display the current game state after each turn because this allows minimal changes to be made in case I want to implement a graphical user interface in the future. With this design pattern, I can retain the structure of the Table Observer but augment it by adding code for graphics rendering. Or, similar to Assignment 4, I can create a new class for a graphics observer and in my main, I can choose which type of observer I want to use to render my game.

Additionally, I designed my classes such that changing game rules would result in minimal modification of the program code. In the class called Player, there is one function called fupdateLegalMoves() that handles most of the logic of the game rules, as it finds all valid plays. My class called "Round" handles the game flow, such as receiving a human player's input, iterating through each player's turn and playing/discarding cards as needed. Lastly, my class

“Game” handles determining when the game ends, initiates multiple rounds if needed, and prints the final results depending on the rules. If there were changes in game rules, it would only need minimal changes to some or all of these 3 classes. For example, if we increase the total score needed before the game ends, the isDone() function in Game class is the only change we need to make. If there are new cards introduced with special functionalities, the functions updateLegalMoves() would change to accommodate the new logic. These 3 main classes for the game all have specific purposes and thus there is high cohesion.

Question: 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 class structures?

A simple way to implement this feature would be to add a function in the Computer class to handle the logic for how a computer acts depending on the game state. This would not result in any changes to my class structure. However, a more object-oriented and scalable approach to this additional feature would be to implement the Strategy Design Pattern. There would be a parent Strategy abstract base class with several concrete strategy subclasses depending on the computer’s strategy rules. The current Game class contains the current game state information, so this would provide context for which strategy to use. There would be an aggregate relationship between Game and Strategy, where the Game class “Has-A” Strategy.

Question: How would your design change, if at all, if the two Jokers in a deck were added to the game as wildcards i.e. the player in possession of a Joker could choose it to take the place of any card in the game except the 7S?

I would add a subclass called WildCard as a child of the Card class. Since the Joker can take the place of any other card, the WildCard class will have additional attributes to indicate what the ‘new’ rank and suit that the Joker card represents. This class would also override the getRank() and getSuit() of the Card class by returning the ‘new’ values instead of the Joker value. I would add additional logic handling in my functions to account for wildcards. I would change the function findLegalPlays() in the Player class to include Joker as a valid card to play.

Additional considerations that result from this addition would be:

a) The total number of cards is no longer evenly divisible by 4 players. Who gets the hands with extra cards?
b) If a Joker is played, how can the program display which value the Joker undertakes?

<u>**Extra Credit Features**</u>

For extra credit, I coded my entire pointers without explicit memory management. I used shared pointers and STL containers such as vectors and maps to manage memory automatically. This was challenging at first because using smart pointers is an advanced topic and I spent a lot of time trying to understand which smart pointer was better to use in my case. I ended up choosing to use shared pointers over unique pointers because my objects required multiple pointers throughout several classes.

Final Questions (the last two questions in this document).

Q1: What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

From this project, I learned the importance of spending a good amount of time on designing our program before coding everything out. Previously in the course, when working on smaller programs such as assignment questions, my strategy would be to get a general solution strategy and just start coding and change things as I go. I was very flexible because smaller programs don't require as much of a sophisticated structure as large programs do. I would restart if necessary and learn new strategies as I tried more approaches for the problem. However, for a large project, after a lot of trial and error, I realised that it's best to explore different strategies in detail when designing the project rather than while you're coding. The design of large projects require much more thought and consideration for your code. I realised the importance of creating a uml diagram to help you visualise your code, which makes the coding process easier as a result.

At the start of this project I felt very overwhelmed because I had no idea about how to organise or structure my classes. However I learned that it's useful to think of the problem in bite-sized pieces. By adopting an object-oriented programming mindset, I tried to group similar features/attributes together and that's how I tried to organise my classes at first. It also took much more trial and error to finalize my class structures. Once I determined my classes, I worked on them little by little and that made the project much less overwhelming. This experience of coding a larger program taught me the importance of doing things one step at a time and not to feel pressured or overwhelmed by the entire project at once.

Q2: What would you have done differently if you had the chance to start over?

First, I would create a header file with global constants such as NUM_PLAYERS, NUM_CARDS, MAX_SCORE, etc, to store these key values, which would make my program more resilient to change in case the rules changed. For example, if the rules changed so that the game ends when one player has more than 100 points, I could change the MAX_SCORE from 80 to 100 and that would be the only change I need to code.

Secondly, I would implement more robust error handling using try catch blocks as well as implementing strong exception safety everywhere. I would also document my exception handling more clearly so that users will know what type of exception safety I've provided.

Thirdly, I would have implemented a special feature that provides suggestions for human players as to which card to discard if they have no legal plays. A strategy one could use to minimize the number of points they incur would be to discard the card with the lowest rank whenever they have no legal plays. This feature could be implemented with a suggestion() method in my Player class, where it searches the hand and returns the lowest ranked card to discard. I would also print this out for the player to see.

Fourth, I would have used the advanced design pattern Model View Controller (MVC) to further decrease coupling in my program. MVC suits the needs of my program because the game state, game display, and control logic would all be separated. My program loosely follows the MVC pattern where my Table represents the game state and TableObserver represents the view. These classes are related via the observer pattern. Then my Player classes use the strategy pattern. The game state and game display would follow the observer pattern, which I did. However, my program doesn't fully follow this model because I also have a Game class which also controls the game logic. I would improve my implementation by reorganizing my classes to follow the MVC pattern more closely.

On a personal organizational note, I would have used a version control platform such as github to organise my code and easily access previous working versions. This time, I tried to reduce coupling by changing my header files and broke my code as a result. It took a lot of time for me to locate and restore my last working version, I think the process would have been more effective using github.

<u>**Conclusion**</u>

Overall, I learned a lot when creating this final project because it is the largest coding project I've worked on so far. I was able to apply a lot of the new object-oriented programming techniques that I've learned during this course and that was very rewarding for me. I've gained a new appreciation for good program design as well as creating UML diagrams to help communicate your code structures to others as well as for my own understanding of my program structure.