Overview:

Classes:

Main:

Main file basically handles creating necessary classes to run an MVC method program. It creats the controller(action), view, model(hydra). Gets the amount of players, Then leaves controller to handle all users inputs from there on.

Action:

Action is the controller in this program. It takes the amount of players, and runs a game with game(). Game() runs each players turn with playerRound(), and playerRound() uses placeCard() to place card inside head when needed.

Hydra:

Hydra is the model of this program. It contains player, head, card classes and data type to store information. It provides many methods to help other class access and manipulate data of player, head, card without breaking encapsulation. Encapsulation is only broken when View, Model needs access to card to display information. Whenever sufficient changes is given, the player class will notify view to update player's screen.

View:

View handles the information display of Heads, and player cards. So, whenever there is changes to Hydra, and the user needs to be known of the information, the View will update and provide new information to users.

Card:

Card is a class that works as a data type. The value and Suit of a card can be accessed with encapsulation.

Player:

Player is a class that works as a data type, it stores draw, discard, reserve, playing card information of the player, the cards in player can be manipulated via public methods provided by players. The cards can also be taken out for other purposes. Encapsulation is strong and never broken.

Head:

Head contains the attributes designed for a head. It has a top card and a list of cards that a head should contain. Cards or important information about a head can be accessed via public method. Encapsulation is strong.

Design:

In order to increase cohesion and reduce coupling, the MVC method using observer have been used to implement the program. View handles display to users, Hydras stores information and is responsible for protected data management. Then, action does the rules for the game. Using MVC, I was able to create rules without making any accommodation in Hydra, everything regarding rules and the actions that player took can be written in playerRound(), placeCard() without making changes to Hydra. Like for cutting of an head, I just need to call PlayToNewHead(), cutHead(), cut() function from Hydra in action. So I don't have to go over the details of implementation, which facilitates rules implementation on a greater scale. View handles display so I just need to notify view using hydra infoRefresh(). For Hydra, it takes care of actions that may be generalized so action can have an easier life, basically, it takes care of things such as draw(), swap(), nextplayer() that are considered an actual action.

To facilitate data managed. Card classes are made so it can act like a data type. This way, I can easily use card and place it in players, head or display it in View as I like. I no longer need to be aware of Suit, or value as Card handles it for me.

Similarly, Player class is also created so I can manage a list of players efficiently. I don't have to manage every data of each player individually anymore. Player was there so I can manipulate player data and handle player information swiftly. This way, hydra don't need to care about how to draw a specific card from a pile, or how to manage piles, all it needs to do is convert basic player card manipulation into actions that Action can use.

Head class is also built for the same reason. To simplify data management, head takes care of all the characteristics of a head, so Hydra can use it like a data type, and take whatever it need from head using get functions.

All three classes built helped make my program implementation simpler and debugging easier as it avoids a lot of unnecessary data management by encapsulating them, and it also make each class target a specific tasks and make unrelated functions separated in different class to reduce cohesion.

Some design helps that my OOP did is avoided possible basic mistakes such as managing a vector or a list of pointers wrongly, since putting into class will automatically reduce such bug risks. Cut head was simplified with Action being responsible for exactly how to cut head, Hydra doing the steps of cut head through function called by action and player managing the card placement. So if I have a player level rules, I modify player, such has reserveToDraw(), and actions like placeCard() will be implemented on head to show how exactly card is placed, and for player drawing card etc, Player will be responsible to implement it such as addDraw().

Resilience To Change:

By using MVC, and ensuring encapsulation, new features and rules for actions can be build by only modifying minimal amount of 1 or 2 classes. For change in big rules, such as valid cut head, cutting head for head < player when there is another head > player, Actions only need to use public methods of Hydra to build the rules. If the rules get specific and the hydra base don't have functions that make up the new rule, we can simply modify hydra a little bit to provide a new rule.

For the bonus where we want red up, black down, we have that since it changes the validity of placing a card, we can simply only modify the valid(), cut() function in Player that checks for validity of the move to implement this feature in our structure, and add an addition option to enable this rule in main. This demonstrates minimum modification and compiling.

If we want to add a GUI interface or change display, changes will only need to be made for view and action, we will redirect the inputs received in Action to interaction from mouse and keypad, and we can modify the View to display information on screen instead. For example. If I want to display more information on linux, I can simply modify View to display more information of player, in my bonus feature -cheat, I only modified View and Hydra to make each player's top 5 cards from draw pile and discard pile visible. Another example would be -list bonus feature, which displays user's valid input in order at the end of the program, I only needed to change action by building a vector to store all strings inputted to be able to handle this case.

If we want to make changes to things like how the cards are drawn, how Cards are pulled, and how many cards are given to each player and what the player has. Basically, the card management related rules, All we have to do is modify Player to manage the rule. For example, if we want to change the amount of cards a player can have, we can simply just modify how cards are given in create Player() and give this option in main so player knows that it should only take certain amount of cards from a deck to each player. This feature is implemented as -amount.

Answer to Questions:

Question 1: I used MVC so changes in game rule or interface would have little impact, change in interface can be easily implemented by changing view and main file, to change game rules, all is needed is to change the rules in action using public methods of hydra.

Question 2: I structed my joker as a card with unique characters to make joker the same data type as other cards in Card class, so it would simply be treated as another card in Card class, and only be treated whenever rules demand. This minimizes the special treatment needed and made my function easier.

Question 3: I still used MVC, since the rules is separated from View and Hydra, computer player is no different from normal player, their change in strategies would not affect how the game is run. Basically, the program runs independent of the change in NPC.

Question 4: Since player is only marked with a name to differentiate each from other player, to change to a computer player, the only information that might need to transfer is name, so I can just simply change the name of the player to indicate that it has now became a new player.

Extra Credit Feature:

The program uses RAII principle. I build many command in line options for optimization and testing. All of the commands except list can be placed in command line at once in any order but with no repetition of any command.

-amount: gives fixed set of amount of card to each player at the start of game from the amount given by input in the beginning of the game, 3 <= amount <= 54.

-randomSeed: prints out the seed used for the RNG in the beginning of the game.

-seed: uses manual seed input at the beginning of the game for the RNG

-list: lists all valid user input at the end of the program in order and the random variable in between initialization of game and starting game. This command line must be put at the end.

-cheat: gives additional display for top five draw and discard of the player.

-valtesting: testing but only asking for value.

-validcut: cuts head < player card even if there is head > player card

-blackup: When card = black, only card < head will cut. Card = red stays the same as default.

Details explained in demo

Quality of life feature:
Grammar an for A, 8.
You can put in 1,11,12,13 as input for joker or testing.

Change in UML:

Initially, I thought of using current player as a management for player, so playing card and reserve would only be in player. But then I found that I can't really downcast Player to current player and other methods would not help OOP, so I removed current player. I changed the name of Info Listing to View, Controller to Action, and redistributed some of the tasks for each class. I didn't include card initially because I haven't thought of that in the beginning. I wasn't sure about how MVC would actually be implemented in my case so I didn't have an observer design pattern set up. It was a later addition

Final Questions:

1. The lessons I have learn about writing large program is that OOP is good and using unique pointer saves a lot of time with memory management. Having a good initial UML and idea for class makes the work process clear and straight forward. Building each class one by one makes the relationship between functions and data easier to comprehend. Moreover, I avoided being stuck on how to implement a specific function or class as I broke them into smaller task so it's easier to build it. It was important to test each class before using them in the whole project as it reduces the time for debug. Reduce coupling and increase cohesion made my program and workflow easier to manage. I can simply add whatever I needed for a specific class by just adding a function on top of it. It felt amazing.
2. If I could have done differently, I would spend even more times think about the validity of my initial UML. Sometimes it didn't consider enough, so when I went with the UML, there were part in classes that didn't integrate well, so I had to revise and delete many function multiple times during my project to make the OOP more viable. But again, it's hard to think up with a near realistic UML when just starting, I know better now so I can clearly come up with a better UML.

Conclusion:

This project has been very fun and, in my opinion, one of the best programs I have made thanks to OOP. Implementing classes had made my knowledge of OOP dramatically increased. I loved this course very much and now I am more ready with C++ to tackle some GUI interface and build projects on my own.