Hydra

Game Introduction: (In here I will introduce in detail how the card game hydra works)

Hydra is a card game for two or more players, which a player wins if he possesses no cards. In the game, each player has area for four decks of cards, namely: stockpile, discard pile, reserve, and handcard pile. Thus, meaning for a player to win, he must have no cards in stockpile, discard pile, reserve, and handcard pile.

The value of cards rank follows the following order: “A, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K”. For “Joker”, when played, the player can give the card any value he pleasures, however if the “Joker” appeared when generating heads, it should be given the value 2. For any head starting with “A”, any card can be place on top value.

To start, we take as many decks of cards as the number of players and shuffle the cards together. Then each player takes 54 cards as their stock. Then player 1 pulls out the topmost card from his stockpile and create the first head.

Players play alternately. To start a round, the player takes the same number of cards from the stockpile as the number of heads on the table. Then the player turns his cards individual and decide which head he wants to place the card on. To “***play***” a card, the value of the current card in his hand should be smaller than or equal to the value of the topmost card of the head. If the

value of the current card is smaller, after playing the card, the player continues to the next card in hand until there's no card left in hand. If the value is equal to the value of the topmost card, by playing the card, the player immediately "***end***" his turn and place all the cards in hand back to the stockpile and then if there's card in reserve, put the reserve back as well. If there's no card in the reserve player can "***reserve***" a card anytime they want during their round, by doing so, he should place the card in the reserve and continues onto turning his next card in hand. However, if theirs is a card in the reserve, to reserve the current card, he should "***swap***" the card in the reserve with the current card. If there's no cards left in his hand, and if there's anything in reserve place the reserve back on top of his stockpile and end the turn. If there's no head that's smaller than the current card, the player can choose either to reserve the current card and continue onto turning his next card in hand, or "***cut a head***", by which he should put the card to his discard pile along with the oldest head, and then "***generate two new heads***", by which he should put down the next two hand cards onto the table to become two new heads. If there's heads with smaller values, the player cannot choose to cut a head.

Overview: （In here I will describe the overall structure of my project）

My program has 5 parts, namely: Game Class, Head Class, Player Class, Deck Class, Card Class, and main(). And in the following I will discuss the structure of each of them.

Game Class is designed to control the "big picture" of the game. It has public methods for handling private variables and as well as the whole logic flow of the game, and private variables such as "testing" (a Boolean for check whether we are in testing mode), "hasWin" (a Boolean for

checking whether somebody has win the game, thus leading the game to end), “playerNum” (an integer for storing the number of players in the game), “roundNum” (an integer for storing which players turn), “gameDeck” (a Deck for generating and shuffling the initial deck of cards for all players), “players” (a vector for storing all the Players), hand “heads” (a vector for storing all the Heads)

Head Class is design to handle each implementation and storage of each head. It has public methods for handling private variables, and private variables such as “id” (an integer for storing the id of the head), and headPile (a Deck for storing all the cards placed on to the head).

Player Class is design to store values crucial for each player and to handle whatever is move is available for each player. It has public methods to handle private variables and players moves, and private variables such as “id” (an integer for storing the id of the player), “totalCards” (an integer for storing the total number of cards), “stockPile” (a Deck for stockpile), “handCard” (a Card for storing the current card in hand), “discardPile” (a Deck for discard pile), “reservePile” (a Deck for discard pile).

Card Class is design to store information for each card and handle its information. It has public methods to set, modify, and get the information of cards, and as well as private variables such as “index” (an integer to store the numeric sequence of cards, from 0 - 13), “suit” (a char for storing the suits of cards, ‘J’, ‘H’, ‘D’, ‘C’, ‘S’), and “rank” (a char for storing the rank of the cards, including: ‘J’, ‘A’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘T’, ‘J’, ‘Q’, ‘K’)

Deck Class is design to handle a vector of Cards. It has public methods for shuffling, and to get the information of the deck, and cards. It has a single private variable: “deck” (a vector of Cards).

Design: （In here I will specify the techniques I used to solve some problems I see as challenging）

Passing Cards: to prevent memory leak and segmentation faults, every time when it’s needed to pass a Card, I pass on a copy and remove the previous.

Adding and cutting Head: since adding to head require info and cards from both Head class and Player class, I created a method in game than link those two together via individual methods.

Joker Card: Every time we draw a joker, we check whether the joker under suit ‘J’ is initialized or not if it’s not we ask for a value. However, when we draw cards to create heads, I used flow control to automatically set the value of Joker to 2.

A Card: I used flow control when check a number is greater or not, and add if the value is one, any value is allowed.

Resilience to Change: （In here I will introduce how my program can adapt changes）

If you look at the structure of my program, it is strictly following OOP, which private variables and public methods to control the logic and the variables. Thus, if any changes should be imposed, it would be easy to be adapted.

Answers to questions:

1. 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.

   Answer: I decided to use the Builder Pattern. Since it provides clear separation between the construction and representation of an object, thus changing the game rules would only impact on several parts of the program.

2. Question: Jokers have a different behavior from any other card. How should you structure your card type to support this without special-casing jokers everywhere they are used?

   Answer: Since the Jokers can be any card we wanted, I would first check if the card type is “Joker” (*in my program checking Suit: ‘J’, index: 0, value: ‘J’*). If it’s a “Joker”, which is what we are discussing for this question, we should ask the player with “Joker” to input a value ***v***, then we change the value of the card (*in my program changing index and*

value) to ***v***. Therefore, leaving the Card to have a type (*suit*) “J” and a value (*index*) ***v***, and *value* ***v***

3. 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?

   Oh whoops, I complete ran out of time for this. Lol. But for allowing bots, I would write create a Boolean variable in Player Class called “autoPlayOn”, and every time I write the program to ask for a move for the play, I implement a flow control to manage the move of the bot.

4. 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?

   Simply, when every time asking for a move, I add hint for “-1” for bot, and this will turn “autoPlayOn” to true, thus start seeing the player as a bot and skips his inputting turns.

Extra Credit Features:

1. Dumb users' proof! I added extra hints and extra protection in case the players of the game accidentally input something incorrect, for instance I hint the players what to input for each input, and if the input is incorrect re-print the message.

Final Questions:

1. I worked alone. I'm kind of used to write large programs, just not ones from C++. For a large program, planning on how your logic will flow is crucial to the development of the program. And always code and look at what you have coded at the same time, by which I mean: code a part of it and let it build and run and see what you would get (this is a habit I develop from my year-long experience in JS development). In this way you can see the things you have built and solve any problems that come up along the way, it would be much easier to debug when you have a small incrementing amount of code than a full bulk.
2. I really regretted not planning carefully before starts coding. For a large project in C++ like this, one must think carefully on passing by references and copies, which is something I ignored at the very beginning, and struggled a lot in between the process of the coding. And which is also why "Segmentation fault (core dumped)" accompanied me all the way along my process to code.