Alex Boostrom
CSC 10200

Final Project Report

I selected this project to challenge myself to build a complex game. I have had some experience with coding in the past, and I wanted to build on it. Having built card classes before, I liked the idea of working with betting statistics to build a game that would generate data, and I selected Texas Hold ‘Em.

I worked alone on this project. The only aid I had was to test the game on others who made decisions I had not anticipated, which showed the need to make adjustments.

The IDE I used was Microsoft Visual Studio Code. Given my previous history with coding, I had found VSCode to be very helpful for catching simple errors I might otherwise have missed. I ran it on an Ubuntu Linux system as I have found Ubuntu to be very user friendly for my needs and much more stripped down than either the Mac or PC operating systems.

This program allows a group of human players to play the poker card game Texas Hold ‘Em as if it were in person with real cards. The program begins with asking the user for the number of players who will be playing, and then requesting names from the user for each of the players. The program then builds a vector of Player objects to represent all the players. Each Player is initialized with the user-provided name. After receiving the player list, the game begins with each player receiving 50 chips to start, tracked in the Player class initialization. The last player added to the vector is marked in the player class as the dealer. This is so a dealer can be found later at the start of the first round. Before initializing the first Round, the program primarily consists of a series of text strings printed to the console for the user to respond to, and getline functions that receive what the user has typed. Once the first round begins, the newly created Round class is initialized with a new Deck of Cards. Both Deck and Card are classes. Card consists of a pair of integers consisting of a cardNumber and suit. For example, the Ace of Spades is saved as cardNumber == 1 and suit == 3. The Deck class builds a vector of 52 Cards. Through a series of for loops, the deck ensures that each suit has 13 different numbers, and that there are 4 suits. The Deck then calls the ShuffleDeck method to shuffle the deck before dealing any cards to the players to ensure that the cards are randomized. Also initialized in the new Round class is a highestBet integer, which tracks the highest bet currently on the table, and a currentDealer index to track where the dealer is in the players vector at any point in time. Finally, the new Round initialization copies the referenced player vector from the main function to edit as necessary. Once the round begins, a new dealer is assigned with the AssignDealer method. This method looks for the previously marked dealer and unmarks the player. The method then marks the next player in the vector (or the first, if the previous player was at the end) and a new dealer is assigned. Then the players ante using the Round method CollectAnte. This process looks for the currentDealer (having just been reassigned and marked with the member variable) and then has the next player in the player vector (or first, if at the end) be the small blind. What this means is the player “to the left” has a chip removed from their chips variable, and the totalBet for that player is incremented by the same amount. Then, the CollectAnte function points to the next player and assigns that player as the big blind. In this case, the player has 2 chips removed from their chips variable, and the totalBet for that player is

incremented by 2. Now the cards are dealt. This consists of deleting the first card in the Deck vector and placing a copy of that card in the hand vector of the current player. This process repeats until every player has received 2 cards. Then the Round method, PlaceBet, is called. Within a betting period, each player will have at least 1 opportunity to either raise, call, or fold. If a player raises, then every other player will have the chance to call or raise that bet. This continues until every player has either called the current highest bet, gone all-in (bet all the chips that player holds), folded, or some combination of those. The logic behind this consists of using helper functions like the player's GetFoldedStat and GetAllInStat to check if a player has folded or gone all-in. If the player has, then they do not get the opportunity to bet. Otherwise, the player gets their card stats printed via the PrintHandText method (see print_screen_for_player_turn.png for an image) and the player gets the three options to choose from. If raise is selected, the user gets asked how much they would like to raise. The user gets the option to raise anywhere from 1 chip to the maximum number of chips they hold. Whatever bet is input by the user, the player's chips decrease by the total amount bet, and the totalBet increases by the same amount. This is done through the BetChips helper function. If the player calls, the process is similar, but the player matches the current highestBet. If the player folds, their member folded variable gets set to true, but their totalBet remains. After a betting round happens, the community hand is dealt cards in a similar way to how cards were dealt to players, except that before each dealing of the sets of cards to the community hand, a card is deleted, or "burned". Then another round of betting happens after every deal except if all the players but 1 have folded. If all but 1 have folded, then the loop of dealing cards and betting is broken and the winning player gets the appropriate amount of chips from the pot. In all other circumstances, even if betting ceases because everyone is all-in, the cards are all dealt to the community hand. Once the final betting has completed and all the cards have been dealt, the hands get assessed. Within the player object, there is a vector of six integers representing the handStats. The first integer in those numbers is a value from 1-10 based on the quality of the determined hand. For example, a Royal flush receives a 10 and an Ace high receives a 1. The second number is the highest relevant card to the 5 cards that make up the determined best possible hand. So, with a straight flush, the second number would be the highest card in that straight. If it were a full house, the second number would be the 3-of-a-kind from the full house. The third number in the handStats vector would be the next highest card that is not the same card. So for the full house, the third number would be the value of the pair that completes the full house. In the flush, the third number is the second highest card of the suit that makes up the flush. This handStats vector is important because it is used to compare the quality of each player's hand. It checks each number in the vector, one by one, until it finds a number that is bigger. The player with the bigger number has the better hand and that means that player with the better hand wins the round (or at least beats the other player being compared). After all the hands of players who have not folded have been evaluated and the handStats vectors properly populated, using the EvaluateCards helper function and its associated helper functions that search for each type of winning hand in the current player's hand, then the Round's DivvyPots method gets called. This method orders the players based on their handStats vectors, using a vector of pairs, with the first number of the pair being the player's index in the player vector and the second number being the tied value. The tied value starts at 0. If two players tie, then the tied value gets incremented, and any tied players with a lower hand after the current tied group get

incremented as well to ensure that no two tied groups have the same tied value. All the ordering ensures that if there are multiple winners who must share the pot, the winning players will all get the portion of the pot that is theirs, but it does not require that they all be paid simultaneously. Because of the ordering logic, and knowing how many tied players there are, the pot can be totaled among all players, divided based on how many winners there are, and then the first player in the tied group can be paid. Then that player gets removed from the tied group and the next player's winnings get calculated in the same way until all payments have been made. The payment process, completed by the AddBetsToWinner method, consists of taking the winner's totalBet and comparing it with the loser's totalBet and adding the smaller of those two values to the winner's chips variable. Then AddBetsToWinner subtracts that same amount from the losing player's totalBet. Once every player's totalBet has been subtracted to zero, all payments to winners have been made. Then the Round method DivvyPots checks if any players have zero chips left after payment. If any players meet that criterion, the player is deleted from the players vector, and thus the game. All other remaining players have their hand vectors erased, folded variables all get set to false, and handStats vectors are all cleared back to zero. Then DivvyPots returns the newly updated players vector to the main function. Following that, assuming there is more than one player in the players vector, a new round begins using the same logic as the previous round with the remaining players and their respective number of chips maintained. Once the players vector reaches a size of 1, then the game is over and the winner is declared. After declaring the winner, the program ends.

The modules include the header for the Round class, a header for the Player class, a header for the Deck class, and a header for the Card class. Each of these headers implements their respective classes and the associated helper methods as mentioned above.

There are multiple algorithms involved. For example, the algorithm for searching for straight flushes consisted of first only looking at the cards in the player hand vector that were of a specific suit, and then determining if, within that group, there were 5 cards in order. But the edge case of the ace as a high card had to be handled, so a separate case had to be checked if there were 4 cards in order, with the king as the highest, and an ace of the same suit.

I implemented a game that plays Texas Hold 'Em. With more time, I would build test suites to test my hand determination methods, with input groups of 5 cards, of specific predetermined values, and then add 2 cards to each to see what the hand determination would return. If the hand determination returned anything less than the value of what was input, then an error would be caught. I would construct an AI that would participate. I would also re-write my hand ordering so the ace would be seen as automatically a high card, and deal with the edge cases of the ace as low in straights.