Chris Fitzgerald

**Blackjack Win Probability of Various Strategies**

A common game at the casino played by many is Blackjack, where the goal is for players to “beat” the dealer. This can be done in two ways: players hand total more than the dealer’s hand without going over 21 or the dealer goes “bust” which is when the dealer’s score goes over 21.

The goal of this object-oriented programming in C++ is to simulate as many plays as you want with various methods to see which gives the greatest win probability.

Starting with main, the program asks you to input the number of decks, and how many hands to be played, then choosing which methods to calculate the win probability. After inputting your choices, the main program will create an object Decks and Method to start the stimulation.

Inside the Decks.cpp, I have two constructors depending on the parameters being passed through to the object and then call the initializer. The default in which there is no input to create a deck object of with just 1 deck. The second constructor will use the number of decks to create the deck. For example, if I wanted to create 6 decks, then it will create a deck vector of length 6\*52 (the number of cards per deck) with values assigned by the initializer by using the modulus, %, from 1 to 10 using the equation deck.at(i) = i % 10 + 1;

Then it will shuffle the deck vector. There is also a function *NewDeck()* to clear the deck and reinitialize the deck object when running out of cards in the deck vector when called. Lastly the *printDeck()* function to print the value of all the cards in the current deck.

In the method class, a constructor that can reference to the deck object to be incorporated in the implementation file, method.cpp. This constructor can start calling the *deck.dealcard()* to be added to attributes playerhand and dealerhand to simulate the beginning of the game where each player and dealer take turns for two cards.

I can keep track of what each has to be able to score the hand by the *PlayerScore()* and *DealerScore()* functions by using a complicated method to determine if the player has an “Ace” with a value of 1 that will let it figure out if it can be a 1 or 11 with:

if (playerhand.at(i) == 1 && (sum + 11 <= 21))

playerhand.at(i) = 11;

The other functions in the method class are clearhands() to remove the cards from the players and dealers for the next play, clearwin() to clear the number of wins in between calling methods and most importantly to print out the win percentage based on the method chosen and plays. This percentage is calculated as *perc = wins\*1.0/(1.0\*totplays);* where wins are the sum of all the win of the hands being played per method called. One such method is trying to hit at least a score of 17 when a dealer shows 7 or higher. As shown

int METHOD::dealerplay(int plays)

{

wins = 0;

totplays = plays;

for (int i = 0; i < plays; i++)

{

if (!((DealerScore() == 21) && (PlayerScore() < 21)))

{

while(PlayerScore() < 12 || (dealerhand.back() >= 7 && (PlayerScore() < 17)))

{

playerhand.push\_back(deck.dealcard());

PlayerScore();

}

while(DealerScore() < 17)

{

dealerhand.push\_back(deck.dealcard());

DealerScore();

}

if (DealerScore() > 21)

wins += 1;

else if (PlayerScore() > DealerScore() && (PlayerScore() <= 21))

wins += 1;

}

clearhands();

}

return wins;

}

The result of this simulation is eye opening because I usually always stand (don’t receive another card) when the player is over the score of 11. With 6 decks and 500,000 plays the win percentage is around 34.0662% with this method. Using the same parameters, when choosing to always hit to at least 17 the win percentage is 47.3532%, however when dealers shows a 7 or higher then the player will try to hit to 17, the win percentage is at 34.0232%. And lastly the method to never hit and see if the dealer goes bust, that win percentage is a paltry 24.6722%.

With this new knowledge that always trying to hit to at least 17 will yield the greatest win probability of the methods tested, I will have a better return on “investment” should I hit up the casino again.

To improve on this program, we can add more players to see if others can affect your win percentage. Adding functions of using Double Down and Splits as well as implementing betting system with positive or negative progressions to increase your chance of earnings. Lastly, the pinnacle of a game simulator is to incorporate running strategy such as card counting with betting system.

I would love to keep working and refining on this program for my own pleasure. Hope you enjoy!