C++ Project: Texas Hold'em Poker

Parth Chawla

December 6, 2016

1 Introduction

In this project, the C++ programming language is used to create an interactive Texas Hold'em Poker game. The user running the program plays against the computer who mirrors the moves made by the user. Three classes interact with each other in this game, namely Card, Deck and Game. A list of what the methods in each class do in short are given below. A second program is also made for statistical analysis where the user is asked to input two cards for which 1000 games are simulated and the number of wins, loses and ties are counted.

Card class:

- Set values for rank and suit of a card.
- Get values for rank and suit of a card.
- Print a card.

Deck class:

- Set deck of 52 cards.
- Shuffle deck.
- Get top card from deck.
- Deal top card.

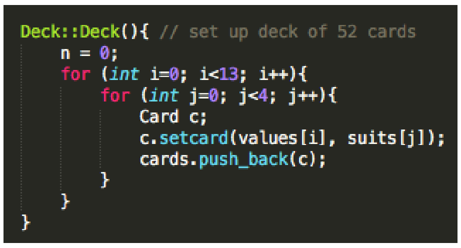
Game class:

- Start game.
- Update money values for players.
- Get 9 cards from top of deck.
- Deal 4 cards to players, set remaining as community cards, get suit and ranks for cards, sort the cards rank wise and find number of each card.
- Print your hand.
- Play round and call, raise, check and fold.
- Print the flop.

- Print the turn.
- Print the river.
- Print the computer's hand.
- Find your high card.
- Find computer's high card.
- Find your hand score.
- Find computer's score.
- Check hand strength.
- Check who has higher score.
- Announce winner.

2 Card Class and Deck Class

At the beginning of the program, two constant arrays are defined, one with ints holding ranks 2 to 14, and one with strings holding the suits spades, diamonds, clubs and hearts. The values in these arrays are passed to the setcard() method in the Deck constructor to set up a deck of 52 cards. This is done using two nested for loops, one which gets the ranks by looping 13 times and one which gets the suits by looping 4 times. The object from the Card class in the Deck constructor is what setcard() is called on and then this object is passed to a vector of cards.



```
Deck::Deck(){ // set up deck of 52 cards
    n = 0;
    for (int i=0; i<13; i++){
        for (int j=0; j<4; j++){
            Card c;
            c.setcard(values[i], suits[j]);
            cards.push_back(c);
        }
    }
}
```

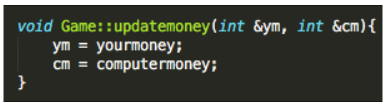
Figure 1: Creating a deck of 52 cards.

The cards are saved in a vector because then in the shuffle() method, std::random_shuffle is used to shuffle the cards easily. Also, it is quicker and easier to get the top card from the vector using the dealcard() method. The cards are printed using the printcard() method in Card class. A switch statement is used in this method to make sure J, Q, K and A are printed for ranks instead of 11, 12, 13 and 14.

3 Game Class

The program starts with the Game constructor that initialises the money values for the player and the computer. The values are passed from the main source file since they need to updated after every game. The start() method is what is called from the main source file and contains the structure of the entire game, all other methods are called from it. The pot is always set to £20 since the game requires a compulsory bet of £10 from the player and the computer.

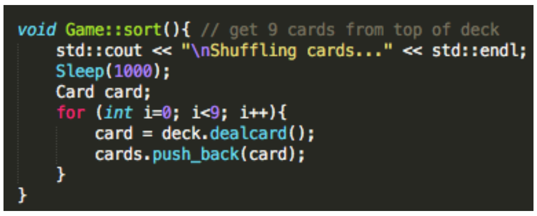
The deck is then shuffled, the cards are dealt, the bets are placed and the winner is found and announced. An option of just terminating the function is also included, in case the player decides to fold. The updatemoney() method takes in two ints from the main source file which need to be passed as reference. This is because the original money values for the player and computer need to be updated after each game.



```
void Game::updatemoney(int &ym, int &cm){
    ym = yourmoney;
    cm = computermoney;
}
```

Figure 2: Updating money by passing as reference from main.

Cards are then sorted using the sort() method which means 9 cards from the top of the deck are taken. These cards are put into a cards vector in the same order so that they can dealt properly.



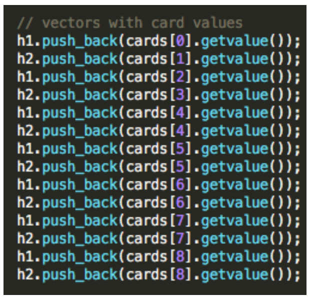
```
void Game::sort(){ // get 9 cards from top of deck
    std::cout << "\nShuffling cards..." << std::endl;
    Sleep(1000);
    Card card;
    for (int i=0; i<9; i++){
        card = deck.dealcard();
        cards.push_back(card);
    }
}
```

Figure 3: Getting 9 top cards from deck

The deal() method deals the cards from the cards vector created in sort(). Four additional vectors are created, h1 and h2 for storing the ranks as ints, and s1 and s2 for storing the suits as strings. The first four cards are dealt alternatively like in a real poker game and then the rest are dealt to both, the player and computer. This makes the player and the computer have a hand of 7 cards each, with ranks and suits for those cards saved in four vectors.

The benefit of splitting them is that then the std::sort is used on the vector with ranks (ints) to easily sort the vector and check for straight/straight flush. Also, the std::count is used to count how many instances of each rank are there in h1 and h2. This doesn't take that long since the vectors only have 7 elements each and makes the process of looking for pairs, two pairs, three of a kind, four of a kind and full house extremely easy.

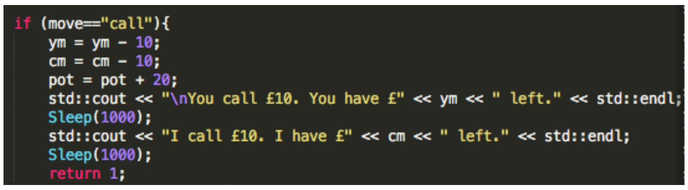
Some methods are then called to print the player's hand, computer's hand, the flop, the turn and the river. These are all done by printing specific elements from the original cards vector.



```
// vectors with card values
h1.push_back(cards[0].getvalue());
h2.push_back(cards[1].getvalue());
h1.push_back(cards[2].getvalue());
h2.push_back(cards[3].getvalue());
h1.push_back(cards[4].getvalue());
h2.push_back(cards[4].getvalue());
h1.push_back(cards[5].getvalue());
h2.push_back(cards[5].getvalue());
h1.push_back(cards[6].getvalue());
h2.push_back(cards[6].getvalue());
h1.push_back(cards[7].getvalue());
h2.push_back(cards[7].getvalue());
h1.push_back(cards[8].getvalue());
h2.push_back(cards[8].getvalue());
```

Figure 4: Dealing cards.

The round() method is where the player actually gets to play the game by raising, calling, folding or checking. Based on the user input, the money values are then updated. In case the player folds, the pot is given to the computer and the start() method is terminated. The money values are passed by reference to this method so they can be updated after every round.



```
if (move=="call"){
    ym = ym - 10;
    cm = cm - 10;
    pot = pot + 20;
    std::cout << "\nYou call £10. You have £" << ym << " left." << std::endl;
    Sleep(1000);
    std::cout << "I call £10. I have £" << cm << " left." << std::endl;
    Sleep(1000);
    return 1;
```

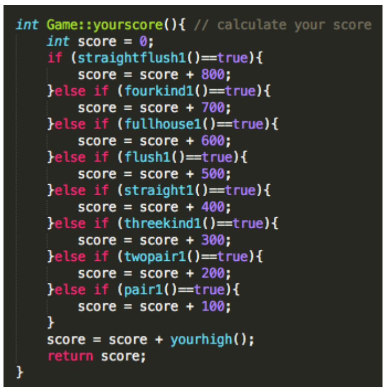
Figure 5: If the player calls.

The high cards in the player's deck and the computer's deck are found using the yourhigh() and computerhigh() methods. This is done by comparing the int values from the original cards vector.

The high card values from before are used in the yourscore() and computerscore() method to calculate the total scores for the player and the computer. First, the hand type is determined (pair, straight, flush etc.) for which the score is given in the multiples of 100 based on the strength. The high card value is then added to this. For example, if both the player and the computer have a pair but the second card in the player's deck is K, whereas in the computer's deck the second card is 9, the player's score will be 113 and the computer's score will be 109. The player will win because of the higher card.

The methods following scoring are booleans for determining the strength of the hand for both, the player and the computer. Because the count for each card is already known from before, checking for pairs, three of a kind, four of a kind and full house is very straightforward. Two pairs are found by using a for loop through the hands. The algorithms for straight and flush are discussed in the following section. Checking for straight flush is the same as checking for straight and flush together.

The evaluate() method determines who has the higher score value. The winner() method first prints what the winning and losing hands are and then announces the winner. The money values are updated by the adding the pot value to the winning player's money value. The money values are passed to the



```
int Game::yourscore(){ // calculate your score
    int score = 0;
    if (straightflush1()==true){
        score = score + 800;
    }else if (fourkind1()==true){
        score = score + 700;
    }else if (fullhouse1()==true){
        score = score + 600;
    }else if (flush1()==true){
        score = score + 500;
    }else if (straight1()==true){
        score = score + 400;
    }else if (threekind1()==true){
        score = score + 300;
    }else if (twopair1()==true){
        score = score + 200;
    }else if (pair1()==true){
        score = score + 100;
    }
    score = score + yourhigh();
    return score;
}
```

Figure 6: Calculating score based on hand type.

main source file by using pass by reference, as before.

NOTE: The reason I've split the scoring, high card calculation and the hand strength determination for the player and the computer is because I wanted to create an interactive two-player game without a Player class. Copy-pasting wasn't as tedious as creating an entire new class or new functions to pass different values for both the players.

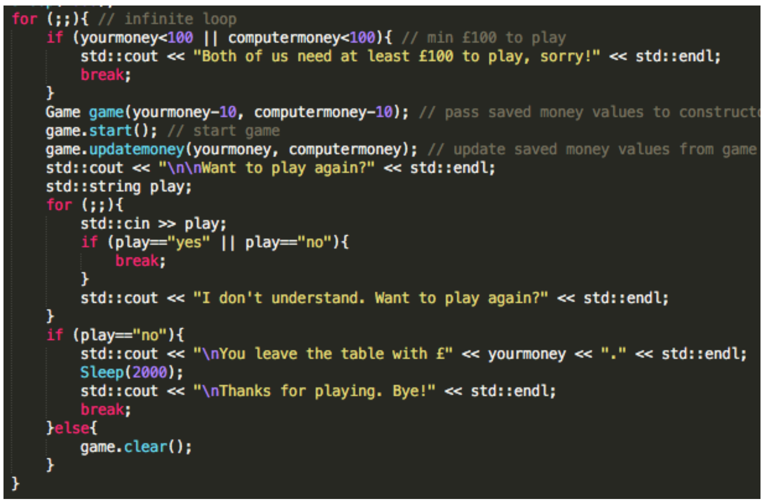
4 Main Source File

The main source file starts by printing the instructions and initialising the money values for the player and the computer. It then creates a Game class object inside an infinite loop and passes to the constructor the money values minus £10 for the first bet. The start() method from Game is executed and the new money values are saved in the existing variables by using pass by reference. The loop also checks if there is a minimum of £100 to play with both players. Finally, it asks the user if they want to play again.

5 Straight and Flush Algorithms

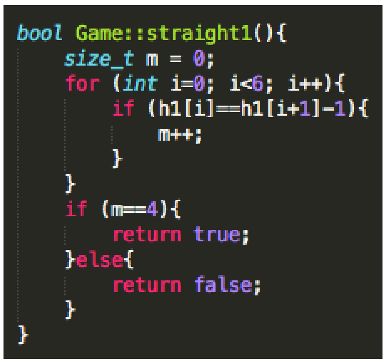
The algorithms for determining if the hand of 7 cards contains a straight or a flush are very similar. The idea is to create a count m which will be incremented every time the condition is true. At the end, if m is equal to 4, the hand contains a flush or straight. This is because we're looking for 5 consecutive cards for which the condition holds.

A for loop from 0 to 5 is used to go through the hand since consecutive pairs are being looked at. h1 and h2 are already sorted which means for straight the condition just needs to be that the difference in consecutive card values is -1. An extra case for A, 2, 3, 4 and 5 is added. For flush, the condition will hold if the suits of consecutive cards are equal.



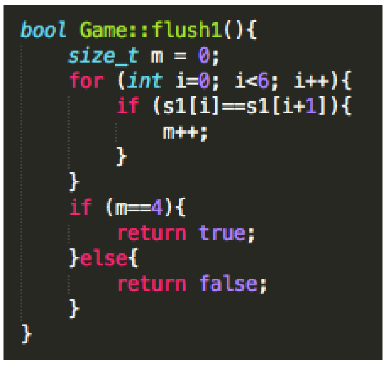
```cpp
for (;;){ // infinite loop
    if (yourmoney<100 || computermoney<100){ // min £100 to play
        std::cout << "Both of us need at least £100 to play, sorry!" << std::endl;
        break;
    }
    Game game(yourmoney-10, computermoney-10); // pass saved money values to constructo
    game.start(); // start game
    game.updatemoney(yourmoney, computermoney); // update saved money values from game
    std::cout << "\n\nWant to play again?" << std::endl;
    std::string play;
    for (;;){
        std::cin >> play;
        if (play=="yes" || play=="no"){
            break;
        }
        std::cout << "I don't understand. Want to play again?" << std::endl;
    }
    if (play=="no"){
        std::cout << "\nYou leave the table with £" << yourmoney << "." << std::endl;
        Sleep(2000);
        std::cout << "\nThanks for playing. Bye!" << std::endl;
        break;
    }else{
        game.clear();
    }
}
```

Figure 7: Infinite loop for playing again.



```cpp
bool Game::straight1(){
    size_t m = 0;
    for (int i=0; i<6; i++){
        if (h1[i]==h1[i+1]-1){
            m++;
        }
    }
    if (m==4){
        return true;
    }else{
        return false;
    }
}
```

Figure 8: Algorithm for straight.



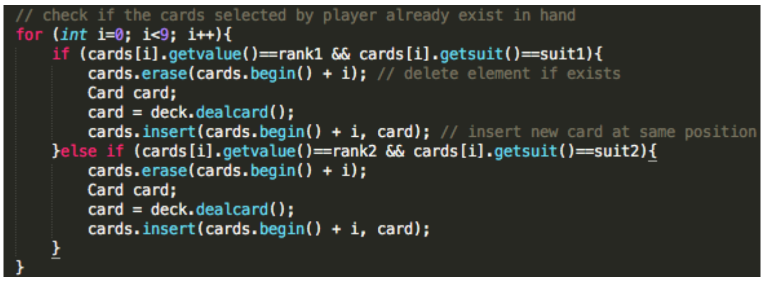
```cpp
bool Game::flush1(){
    size_t m = 0;
    for (int i=0; i<6; i++){
        if (s1[i]==s1[i+1]){
            m++;
        }
    }
    if (m==4){
        return true;
    }else{
        return false;
    }
}
```

Figure 9: Algorithm for flush.

6 Statistical Analysis

A new source file with a new main() function is created for statistical analysis. The user can enter cards they want to simulate 1000 games for and see the number of times they win, lose or tie. This is done by removing all the betting and passing the card rank and suit as parameters to the Game object. The number of wins, loses and ties are calculated and updated just as the money was in the game. The ranks and suits of the cards selected by the player are added to h1, h2, s1 and s2 vectors instead of drawing them from the top of the deck like before.

To make sure that duplicate cards don't exist in the hands, an if statement is used to delete and replace cards with the same ranks and suits with new ones from the deck.



```
// check if the cards selected by player already exist in hand
for (int i=0; i<9; i++){
    if (cards[i].getvalue()==rank1 && cards[i].getsuit()==suit1){
        cards.erase(cards.begin() + i); // delete element if exists
        Card card;
        card = deck.dealcard();
        cards.insert(cards.begin() + i, card); // insert new card at same position
    }else if (cards[i].getvalue()==rank2 && cards[i].getsuit()==suit2){
        cards.erase(cards.begin() + i);
        Card card;
        card = deck.dealcard();
        cards.insert(cards.begin() + i, card);
    }
}
```

Figure 10: Checking for duplicate cards.

The following table shows the numbers of wins, loses and ties for 11 cards. The win percentage is calculated for each card and is compared to the ideal percentage in a real poker game.

First Card	Second Card	Wins	Loses	Ties	Win %	Ideal %
A of Spades	A of Hearts	745	239	16	74.5	85
A of Diamonds	K of Diamonds	655	282	63	65.5	66
A of Diamonds	K of Spades	628	302	70	62.8	65
A of Clubs	J of Hearts	620	328	52	62	63
J of Clubs	9 of Spades	521	442	37	52.1	54
7 of Hearts	7 of Clubs	593	393	14	59.3	66
3 of Hearts	3 of Clubs	533	466	1	53.3	53
8 of Diamonds	9 of Spades	463	503	34	46.3	46
3 of Spades	K of Diamonds	594	360	46	59.4	49
2 of Hearts	6 of Hearts	346	630	24	34.6	35
2 of Hearts	6 of Clubs	304	670	26	30.4	31

Some hands are significantly more closer to the ideal result than others. The reason for this is that some algorithms are more accurate than others. The algorithms for straight and flush seem to be more accurate than the algorithms for high card, pair, two pair, three of a kind and four of a kind. The number of ties seems to be arbitrary but probably also has to do with my high card algorithm.