CSC 17A - 48983 - 12/15/2015 - Amul Bham

Project 2
Casino Bham's BlackJack

Introduction

BlackJack is a simple card game in which the player competes against the dealer to get the higher hand but not over 21. Whoever goes over 21 loses. The player receives their first two cards and the dealer receives one. Based on the player's card total, they must decide to "hit" or "stay", trying not to go over 21 but still attempting to have a higher hand than the dealer. The dealer is set up in a way where they never "hit" if they are at 17 or higher, therefore it is the player's decision to hit or stay depending on if they think the dealer will bust or have a low card total.

For example: If the player has a hand of 14, they risk the chance of going over 21 on their next card draw. So they might decide to stay, despite having a low card total, in hopes that the dealer will ultimately bust. If they dealer has a 9 for their first card, they will keep drawing cards till they are at 17 or higher.

Basically the dealer is always playing risky, and it is up to the player to outsmart the dealer based on odds and knowing when to be aggressive or passive. Also note that if the player goes over 21, they lose no matter what because they draw their cards first. If the hand ends in a tie, no one wins. If a user has a blackjack (21), the dealer still has a chance of drawing a tie if they also have a 21.

For this version of the game I added multiplayer functionality up to four players, meaning the user can now play with other people which is more representative of the atmosphere of a real casino. It is also important to note that the players are only playing against the dealer and have no impact on each other. Meaning if you bust, and the dealer busts for their hand, then all the players win except the player who busted.

Summary:

Project Size : About 1300 lines

Number of Variables : 20-30

Number of Classes: 4

Number of Methods : 30-40

Repositories and Progress of Development:

Project 1 (Original Source code from project 1 that I modified):

https://github.com/amulbham/CSC-17A/tree/master/proj/Project_1/BlackJack_Project_

Project 2 (After Modifications)

https://github.com/amulbham/CSC-17A/tree/master/Blackjack_2

For the second project, I incorporated most concepts learned from the chapters 9-16 and felt I truly understood the process of object orientated programming. For this version of BlackJack I abstracted the game flow and logic from being a linear top down design, to a object orientated design utilizing four classes in addition to the main. The project took a total of 30 - 35 hours over the course of two weeks but most of my time can be pinned down to several challenges I faced in converting to a object orientated design. These challenges also prevented me from adding a few extra features; however the structure of the program, and the reusability of the classes leaves room for tons of improvement. In addition I was able to properly incorporate with ease all of the new concepts learned over the course of the semester.

Developing the abstraction of the players and overall game design and functionality was one of the more challenging parts of this project. The abstraction goes as follows. I created player, dealer, and deck classes that are completely stand alone classes. The dealer and player class simply consist of mutator and accessor members to store and return information about the player such as the player name, balance, binary position etc. For blackjack, they both consist of a card total variable that tracks their card values for a given game. The deck class is also developed as a stand alone class; the entire class is wrapped up into one simple function named deck.drawCard(); which displays the card drawn from the deck and returns a card value which can be applied to a variable by the caller. The game class controls all the logic of the game of blackjack and each function within the class is abstracted to handle a certain section of the game. The main function is used to organize and determine which parts of the game to call.

The player and deck objects are aggregated in the game class as blackjack always has players, and always has a deck and thus the player and deck class do not have to be determined outside a game class. Thus if I were to add a another game, I could use the main again to control the abstract functions and develop a menu to handle both games.

Another major hurdle was designing every function and section of the game to handle multiple players, meaning each function should be able to act on each player object the same. This completely changed the way I approached every part of the blackjack class and the entire game overall, as anything that occurred in the game had to be looped to occur to each player. To account for a dynamic amount of players, I used a vector x as a container to store player objects, which were dynamically created in the game class depending on how many players are playing. Then, for each part of the game, the function is looped through for each object in the x container. This ensured that each player was equal and goes through the same set of functions as the other.

Another major challenge was incorporating reading and writing from a binary file with classes. I still wanted to save each player's name and balance in a file so future games; in order to do so, I created a temporary structure used only to hold a given player's name and balance, so it can be written to the file as binary data. If the player is new, the file is appended with the new player information, if the player is returning, I used random access memory concepts such as seekp to seek to the player's position in the file and overwrite the older information. This took me several days to figure out and properly debug as I could not figure out why it would not work with a class, and soon realized that structures can only be written properly to a binary file.

UML Design:

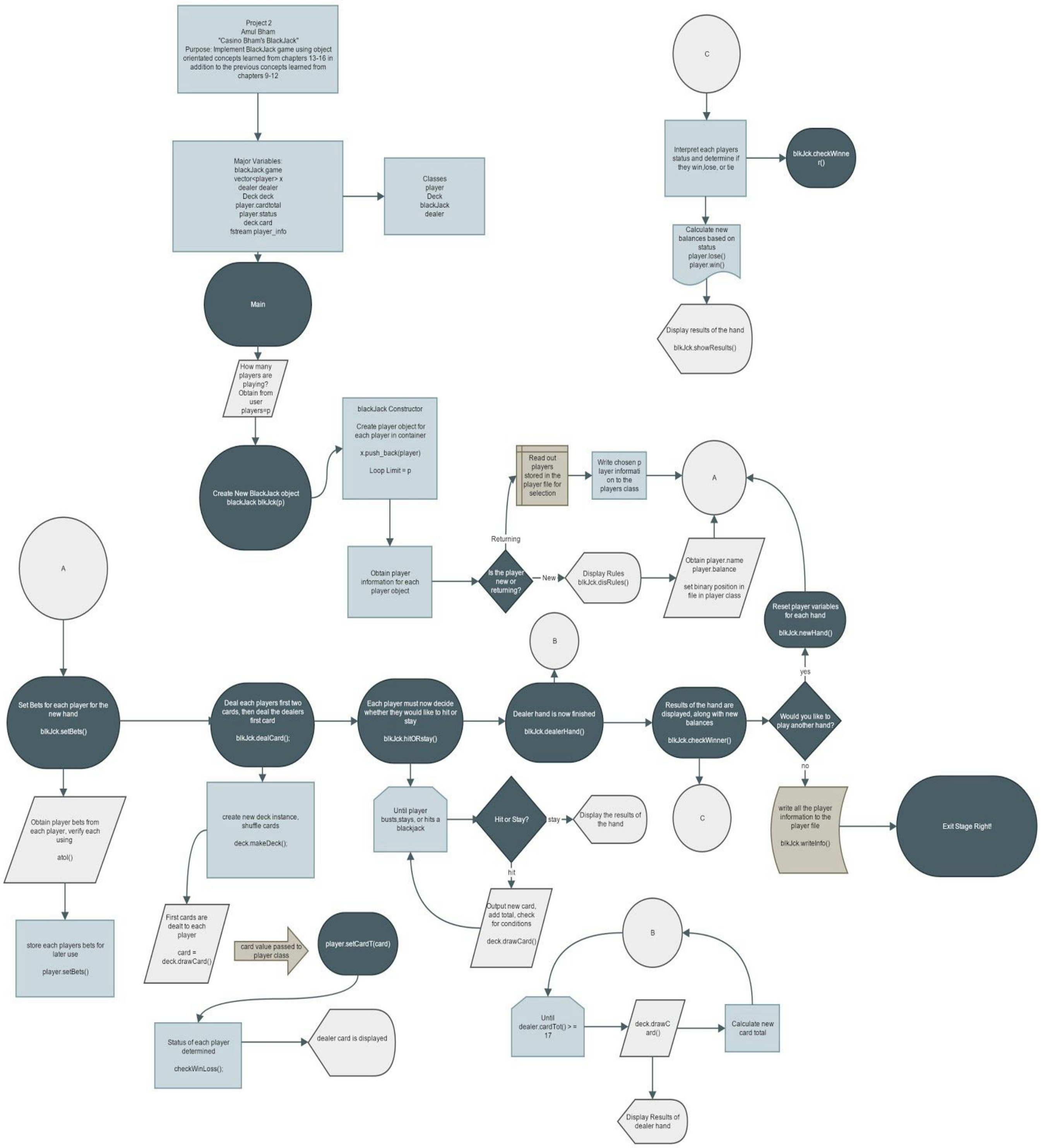
```
blackJack
--------------------------------
-    dealer : dealer
-    x: vector<players>
-    deck : deck
-    numP:int
-    cot: int
-    temp : play_info
-    player_info : fstream
--------------------------------
+    blackJack():
+    blackJack(int)
+    disBord(): void
+    disRules(): void
+    checkWinLoss();:void
+    setBets();:void
+    hitORstay();:void
+    checkWinner();:void
+    showResults();:void
+    dealCards();:void
+    dealerHand();:void
+    newHand();:void
+    getInfo(player &);:void
+    writeInfo();:void
+    virtual ~blackJack();
```

```
Deck
--------------------------------
-    deck: vector<int>
-    card: int
-    total: int
-    y: int
-    face: string
-    value: string
-    tCard: const int
--------------------------------
+       Deck()
+       shuffleCards():void
+       makeDeck():void
+       getFace():void
+       getValue():void
+       drawCard():int
+       dispCards():void
+       virtual ~Deck();
```

```
player
----------------------------------
-    bal : long int
-    bet : long int
-    binN: int
-    newP: bool
----------------------------------
+       setBin(int n) :void
+       setBal(long int b) :void
+       setBet(long int c) :void
+       setStat(); :void
+       setStat(int num) :void
+       setNew(int num) :void
+       win() :void
+       loss() :void
+       reset(); :void
+       giveNew():bool
+       giveBal(): long int
+       givePos():int
+       player(const player& orig);
+       virtual ~player();
```

```
dealer
----------------------------------
-    cardTot: int
-    status: int
-    ace: bool
----------------------------------
+    setAce(int num): void
+    dealer()
+    setStat():void
+    setCardT(int):void
+    addTotal(int):void
+    virtual void setStat(int num){status =
     num;}
+       giveAce():bool
+       giveTotal():int
+       giveStat():int
+       reset(): void
+       dealer(const dealer& orig);
+       virtual ~dealer();
```

Flowchart:


Project 2
Amul Bham
"Casino Bham's BlackJack"
Purpose: Implement BlackJack game using object orientated concepts learned from chapters 13-16 in addition to the previous concepts learned from chapters 9-12
Major Variables:
blackJack game
vector<player> x
dealer dealer
Deck deck
player.cardtotal
player.status
deck.card
fstream player_info
Classes
player
Deck
blackJack
dealer
Main
How many players are playing? Obtain from user players=p
Create New BlackJack object blackJack blkJck(p)
blackJack Constructor
Create player object for each player in container
x.push_back(player)
Loop Limit = p
Obtain player information for each player object
Is the player new or returning?
Returning
New
Read out players stored in the player file for selection
Write chosen player information on to the players class
A
Display Rules blkJck.disRules()
Obtain player.name player.balance
set binary position in file in player class
C
Interpret each players status and determine if they win,lose, or tie
blkJck.checkWinner()
Calculate new balances based on status player.lose() player.win()
Display results of the hand
blkJck.showResults()
A
Set Bets for each player for the new hand
blkJck.setBets()
Obtain player bets from each player, verify each using
atol()
store each players bets for later use
player.setBets()
Deal each players first two cards, then deal the dealers first card
blkJck.dealCard();
create new deck instance, shuffle cards
deck.makeDeck();
First cards are dealt to each player
card = deck.drawCard()
card value passed to player class
player.setCardT(card)
Status of each player determined
checkWinLoss();
dealer card is displayed
Each player must now decide whether they would like to hit or stay
blkJck.hitORstay()
Until player busts,stays, or hits a blackjack
Hit or Stay?
stay
Display the results of the hand
hit
Output new card, add total, check for conditions
deck.drawCard()
B
Dealer hand is now finished
blkJck.dealerHand()
B
Until dealer.cardTot() > = 17
deck.drawCard()
Calculate new card total
Display Results of dealer hand
Results of the hand are displayed, along with new balances
blkJck.checkWinner()
C
Would you like to play another hand?
yes
Reset player variables for each hand
blkJck.newHand()
no
write all the player information to the player file
blkJck.writeInfo()
Exit Stage Right!

Major Variables:

Type	Variable Name	Description	Location
Objects:			
dealer	dealer	used to store dealer information and status for the game	Defined within the game.h file
x<vector>	player	Used to dynamically create and store instances of player class for each game	game.h
player_info	fstream	Used to stream data in and out of a binary file	game.h
Deck	deck	deck object used to create a deck and obtain a value from it	game.h
Structures:			
play_info	temp	used to temporarily store player information for reading and writing from a file consisted of a char string for the name, an int value for the balance, and a regular int for the position.	game.h
Vector:			
int	deck	used to store the 53 values of a deck of card, randomly shuffled and ready to use	Deck.h
Int :			
	dealer.cardTotal	Used to track the card total of each player from section to section	player.h
	player.status	used to track the status of each player, determines how the program handles each instance of player class	player.h

	player.binN	used to track each player’s location in the binary file, so program knows where to overwrite the file with new data or append	player.h
	card	Tracked the card number that was drawn, incremented for the next card after each card draw	Deck.h
	dealer.ace	used to track if each player for aces, also the dealer as well	dealer.h
String:			
	Deck.face Deck.value	used to store the string value of each card drawn for display	Deck.h
Long int:			
	bal	Used to track each player’s balance throughout the game, written and saved to a file after the game is finished, saved from game to game	player.h
	bet	Used to store the value of the player bet for each hand, then applied to the balance depending on the outcome of the hand for each player	
Bool:			
	newP	true or false value to track if a player is new or returning	player.h

Checklist of Concepts Chapters 9-16:

Instead of making a giant list showing how I used every single concept used in the book, I formatted my list to simply explain why I believe my project demonstrates full understanding on that chapter. I use a section in my project that demonstrates my knowledge and from this I assure that a fair assessment can be made without me having to explain every single concept and its location.

Chapter	New Concept	Implementation
9	This chapter mostly covered dynamic memory and how to use pointers to allocate memory. My entire multiplayer addition was all based around dynamic memory allocation. Based on how many players are playing, a vector of players is created and allocated memory on the fly.	Look at the many vectors I use to address dynamic memory allocation. My main vector<x> player container is my game.h file demonstrates my knowledge and skill in handling dynamic memory. It creates instances of player class on the fly

10	this chapter mostly covered c string and string objects and how to use them for input validation and other useful techniques. Overall I use the techniques in this class so casually now that explaining how I used everything in this chapter would be redundant. Glance over any main function in my game class and you will see the evidence	I used the getline object for all my string objects to give them values, in addition, whenever I obtained user input I used the cin.get() function. I obtained player bets in a char string to check for validation, then I used atol to convert the bets into a long int which could then be added to the balance. In addition I used the a char string to store each player's name in order to be able to store the player

		name in the player file as binary data

11	Chapter 11 converted structured data, and even though classes mostly handle the duties of structures, I did find one very useful way to use them. Since structures can be written to files as binary information, I copied each player's information from their player object to a temp structure, from here I was able to properly copy the player information to the file. The structure was used to create a record in a file of all the past players	Check the getInfo() or writeInfo() functions to see the use of the structure for reading and writing from my player file.

12	Advanced file operations were one of the hardest parts of my program to get right and I know for a FACT I know this chapter very well. Look no further than my getInfo() or writeInfo() functions. I increment through the file, use random memory access to position the file, the list goes on. I definitely used all the possibilities of chapter 12.	getinfo() or writeInfo() functions in my game.cpp file. Has reading/writing/appending from a binary file as well as random memory access.

13	Classes: well what else is there to say? I abstracted the game into 4 different classes, used 1 for game	Look at any class in my program and you will see all the concepts and then some from chapter 13

	logic, the others to store information, I use constructors, accessor functions, mutators, private members, objects of arrays etc etc.	

14	More about classes: Aggregation was a crucial part to how my game worked. New player and deck objects are only defined within the blackjack class itself as those objects existing outside of that class is pointless. This allowed me to have one class which controlled all the game flow with a deck object and player objects defined within it, giving the game complete access to all the player and deck member function and variables.	Take a look at my game.cpp file to see my use of aggregation and all the cases where I define a class inside another class, fit naturally into my project

15	I made the dealer class a template function for my player class which in turn inherited all of its variables and member functions. This was possible because the dealer shared some basic variables with the players and functions with the player such as the status of their ace. I also pass my player objects around by reference in my getInfo() function	Look at how my dealer and player class interact with each other and how one inherits traits from another

16	STL is a main tool used in my program and has been for a while. I use vector library as a dynamic container to store objects such as player objects. In addition I use the random_shuffle function in the algorithm library to shuffle my deck of cards which are stored in a vector of ints.	Check out my deck class, or my random shuffle functions in my deck class to see the use of random_shuffle, All throughout my program I use vectors as containers, look at my blackjack constructor to see its use of push_back to dynamically create player objects

Program:

```
/*

 * File:   main.cpp

 * Author: Amul Bham (Project 1)

 * Purpose: To play multi player Blackjack against

 * A.I. implementing methods learned from

 * chapters 9-16, Main Function used generally just to control

 * abstract game flow and logic, as well as obtaining beginning user decisions

 * such as the number of players

 * Created on October 17, 2015, 2:42 PM

 */

//System Libraries

#include <iostream>

#include <algorithm>

#include <ctime>
```

```
#include <cstdlib>

#include <fstream>

#include <cctype>

#include <string>

#include <vector>

#include <cstring>


using namespace std;


//Global Constants


//User Libraries

#include "Deck.h"

#include "player.h"

#include "game.h"

#include "dealer.h"


//Function Prototypes

//Execution begins here

int main(int argc, char** argv) {

    //Declare Variables


    /*Both of these variables were used as outside switch variables
```

```
 to trigger what happens to the other classes and functions,
 if the player is playing with others, a vector is created,
 agn is used to control the game loop, which starts at the beginning of
 each hand and ends when agn turns to 'no'*/
 int p;
 char agn;


 //Display a greeting to the user
cout<<"                Casino Bham's BlackJack"<<endl<<endl;


/*Determine how many players are going to play, from here
 a vector is created with p players of the player class
 ex: p = 3; 3 instances of player class are created and place in a vector*/
do{
 cout<<"How many players would like to play? (Up to 4 total)"<<endl;
 cin>>p;
 //Ensure the amount of players is not over 4 as it can get hectic on one screen
 if (p<1 || p>4) {cout<<"Invalid players!"<<endl;}
}while(p<1 || p>4);
 /*Create a new instance of the blackJack class, this class control
  all of the game flow of blackJack game for the user, meaning the player
  and deck classes are totally separate and stand alone, in the future more
  games can be added here by simply creating a new class for the game and
 adding a menu here for each game*/
blackJack blkJck(p);



/*Game Loop starts here -> from here is where each new hand starts,
 the players have the choice to leave after each hand, if they do want to
 play again, their card totals are reset, however their names and
 binary file positions are maintained as that was done in the
 creation of the blackJack object*/
do{
blkJck.disBord();
/*Get the bets for the hand from each player, keep track of the bet amount
, starting balance, and if the player has overdrawn and needs to buy in again*/
blkJck.setBets();
blkJck.disBord();
/*The first cards are dealt to each player,the drawnCard function of the
 * card class is called, a card is printed out, and its value is
 * passed back and added to the players overall card total for
 * the hand, if the player has any significant hands (blackjacks, busts, etc)
 * they are displayed after they are encountered.
 each players hand has been displayed*/
blkJck.dealCards();
```

```
 blkJck.disBord(); //<- this function just prints out a standard border



 /*At this point, the players must decide if they want to hit or stay
  depending on their current card value, if a player decides to hit or stay,
  each player is given the turn and can continue till they decide to stay,
  go over 21 or hit a 21, at which point it begins the next players turn
  Status is tracked after each players hand is finished, meaning
  if they lose they are marked as a loss, if they hit a 21 they are marked
  * as a possible win,if they decide to stay under 21 they are marked for compare */
blkJck.hitORstay();
blkJck.disBord();

/*After the hit stay portion of the hand, assuming that at least 1 of the players
 has not 'busted', the rest of the dealer hand is dealt till the dealer
 hit a value of 17 of greater All the cards were displayed and valued
 using the drawn card function in the deck class*/
blkJck.dealerHand();
blkJck.disBord();

/*Using the status of each player, and the dealer, which was tracked
 after each card draw, a final comparison is held between each player
 and the dealer card value, So for example if the dealer had a 21,
 any player who is not marked with a status of '21' automatically
 * loses, and the players who do have a 21 are marked as a tie. After this
 * the balance of each player is recalculated based on the status of
 * their hand, tie = no money loss, loss = loss of bet etc.  */
blkJck.checkWinner();


/*The results of the hand are displayed in a overall report at the end
 of each hand and a recalculated balance is printed out for each
 player, from here they can decide to play another hand or end the game*/
blkJck.showResults();
blkJck.disBord();
/*The players can now decide, after seeing their results, if they want to
 play again*/
cout<<"Would you like to play another hand? Y/N"<<endl;
cin.get(agn);
/*If the players do want to play a new hand, then the game loops back to the start
 of the doWhile loop, all the card values and totals are reset for each player*/
 if (tolower(agn) == 'y') blkJck.newHand();// <- Resets each players hand
 }while(tolower(agn) == 'y');


 /*If the players decide they are finished playing, each players information
 including their name and balance are stored in a file, this allows each player
 to have their own separate account in the game and can maintain
```

```
    their own balance in the game. If they already have a spot on the binary
    file, their spot is overwritten using random access file entry, and overriding
    the spot on file that represents their binary number, if they are a new player
    their information is appended to the file in a new spot and a binary number
    representing their spot on the file is assigned to the new player*/
    blkJck.writeInfo();


    //Exit Stage Right!
 return 0;
}
/*
 * File:   game.h
 * Author: Amul.bham
 *Purpose: The Main Purpose of this class was to control the game flow
 * and logic of the game of BlackJack. This allowed my main function to focus on
 * the more abstract dealings of the program such as guiding which
 * part of the game is to be executed first, while the game class
 * handled all the minor details and logic points involved in each
 * part of the game.
 * - In the game class, their lies a vector of player classes and a
 * instance of a deck class which I decided to aggregate into the game class
 * as any instance of a blackJack game will have to have players and a deck
 * of cards, therefore it was more logical to define the players within
* the class based on how many players are playing for that given session
*
* Aggregation Logic :
* - A game of blackjack has -> players
* has a -> deck of cards
* and a -> dealer
*
* - By allowing my main to control the higher abstract game flow, the game
* class controlled how each class interacted with each other and passed
* values of each player to the different parts of the game. The main
* was used to control what parts of the black jack game were to be executed
* in what order.
*
* - This kept the main as sort of a make file in that it simply controlled
* what parts of the game class to execute, this made it so in the future,
* I could add other game classes and use the main again to control that game
* as well, therefore allowing players to select which game they would want to play
* from a menu in main.
*
* Created on December 9, 2015, 12:14 AM
*/
```

```
#ifndef GAME_H

#define GAME_H


#include <vector>

#include <iostream>

#include <ctime>

#include <algorithm>

#include <string>

#include <fstream>

#include <cstring>

#include "Deck.h"

#include "player.h"

#include "dealer.h"


using namespace std;


/*I created a temp structure as a means to temporarily store each players

 crucial information, taken from each players instance of player

 * in the x container, and then write it into the player.txt as binary data

 * (Classes cannot be written into file as binary)

 * And Also to store the information from the player.txt for reading out

 * to the player at the start of the game

 */
```

```
  struct play_info{
      //Crucial information includes player name, their balance, and their
      //binary poistion in the file
       char name[25];
       long int bal;
       int bN;
   };
class blackJack {
public:
   blackJack();
   blackJack(int n);
   //Descriptions for all functions can be found in the .cpp file
   void disBord();
   void disRules();
   void checkWinLoss();
   void setBets();
   void hitORstay();
   void checkWinner();
   void showResults();
   void dealCards();
   void dealerHand();
   void newHand();
   blackJack(const blackJack& orig);
```

```
    virtual ~blackJack();

    //create a player_info fstream to store for reading and writing our data

    fstream player_info;

    //Instance of the structure defined above

    play_info temp;

    void getInfo(player &);

    void writeInfo();


private:

    dealer dealer;  //instance of 1 dealer object for the game

    vector<player> x; //Container of player objects, each representing a player

    Deck deck;      //1 deck is object is required for the game

    int numP;       //holds the number of players

    int cot;

};


#endif   /* GAME_H */

/*

 * File:   Deck.h

 * Author: Amul.bham

 * Purpose: The deck classes was responsible for all the dealings of

 * the deck of cards used in the game. The idea was to wrap

 * all the complexity of creating a deck of cards, I really tried to make
 * the class a individual meaning it can be applied to really any card game,
 * the drawCard function does it all, wraps up all the dealings of the deck in
 * one function, prints out the cards, and returns a card value
 * The Class creates a vector of values 2-53 and then shuffles that vector
 *  - The card is drawn starting from value zero and then the variable is incremented
 * after each card draw
 * - Once the card number hits a certain limit, the deck is re shuffled automatically,
 * displaying a message to the players so they know.
 * - The name of each card is printed out as a string, and the value of the card
 * is returned back to the the caller for what ever purpose.
 *  -Ex: for blackjack, the card value is returned and added to the players
 * overall card total.
 *
 * Created on December 8, 2015, 7:37 PM
 */


#ifndef DECK_H
#define DECK_H
using namespace std;


#include <vector>
#include <iostream>
#include <ctime>
```

```
#include <algorithm>

#include <string>

#include <fstream>

#include <cstring>


class Deck {

public:

    Deck();

    void shuffleCards();

    void makeDeck();

    void getFace();

    void getValue();

    int drawCard();

    void dispCards();

    Deck(const Deck& orig);

    virtual ~Deck();

private:

    /*Vector used to store values 2-53 to represent cards in a deck

     allowed for use of random_shuffle function and easier manipulation*/

    vector<int> deck;

    /*Card int used track the current card # of the card being drawn, it is

     incremented after each draw for the next card

     total stored the card value of card that was drawn, as the card # had no
     relevance to the actual value of the card*/

    int card,total,y;

    /*Face and Value used to store the string representations of the card that was drawn,

     they are then sent to the display cards function for output, kept separate from

     the value of the card itself as face cards were not represented as numbers*/

    string face,value;

    //Static const used to represent the number of cards in a deck, ensured that this value did not
change

    static const int tCards = 53;




};




#endif   /* DECK_H */

/*

 * File:   player.h

 * Author: Amul.bham

 * Purpose: I needed a class to track all the important player information

 * for each player, throughout the game, keeping everything organized.

 *

 * As this class mainly stores information, most functions have no purpose other

 * than to return the called upon player variable or set a player variable to a
 * new value
 *
 * The player class is derived from the dealer class as they share certain
 * common characteristics such as having to have a card total variable and
 * other statuses that needed to be tracked such as receiving an ace, so
 * it made sense to use the dealer class at a template for my player class,
 * adding specific variables for the player class, such as balance.
 *
 * I also made it so the player class ACT for ONE player and represents the
 * dealings of ONE player, thus a container must be used such as a vector
 * for multiple players, I found this to be must easier and to simply loop
 * through each player instance than to create specific functions for multiple
 * players
 *
 * From this class, the only player information stored in the file for
 * later games are the name, balance, and binary number position, everything else
 * is created new from game to game.
 *
 * Used mostly in line functions as the functions were short and to aid in speed
 * Created on December 8, 2015, 9:12 PM
 */


#ifndef PLAYER_H
```

```
#define PLAYER_H


#include <vector>

#include <iostream>

#include <ctime>

#include <algorithm>

#include <string>

#include <fstream>

#include <cstring>

#include "dealer.h"


using namespace std;


//Inherits all the functions and protected members of dealer class

class player : public dealer {

public:

    //Sets binary number position

    void setBin(int n){binN = n;}

    //Sets all values to 0, and new player by default when a player class

    //is created

    player():dealer(){

        bal = 0;

        bet = 0;
        newP = true;

        setBin(50);

    }

    //Sets the balance of the player

    void setBal(long int b){bal = b;}

    //Sets the bet of the player

    void setBet(long int c){bet = c;}

    //Sets the status, overrides the dealer setStat function

    void setStat();

    //Used to manually set the status (used as a check)

    void setStat(int num){status = num;}

    //new or returning? Determines how the program will write data to file at end

    void setNew(int num){if (num == 0) newP = true; else newP = false;}

    //give the status of the player

    bool giveNew(){return newP;}

    //win? add the bet, loss means you lose bet

    void win(){bal +=bet;}

    void loss(){bal -=bet;}

    void reset();

    //return the balance to the calling object

    long int giveBal(){return bal;}

    //return the binary position of the player, for reading and writing

    int givePos(){return binN;}
```

```
    player(const player& orig);

    virtual ~player();

    char name[25];

private:

    //Stores the balance of the player

    long int bal;

    //stores the bet of the player

    long int bet;

    //stores the binary position in file of the player

    int binN;

    //TRUE = new player, FALSE = returning player

    bool newP;

};


#endif  /* PLAYER_H */

/*

 * File:   dealer.h

 * Author: Amul.bham

 * Purpose: To store and provide access to all the important dealer

 * information needed for a game

 * Created on December 12, 2015, 7:51 PM

 */

#include <vector>
```

```
#include <iostream>

#include <ctime>

#include <algorithm>

#include <string>

#include <fstream>

#include <cstring>

using namespace std;


#ifndef DEALER_H

#define DEALER_H


/*Used to represent the dealer of a card game, only needed mutator and

accessor functions, and variables to store dealer information

in regards to the game being played (ex :card total for blackjack)

 */

class dealer {

public:

    dealer();

    //Set the ace value to true or false

    void setAce(int num){

        if (num == 0) ace = false;

        else ace = true;

    }
    virtual void setStat();

    void setCardT(int card);

    void addTotal(int card);

    virtual void setStat(int num){status = num;}

    bool giveAce() {return ace;}

    int giveTotal(){return cardTot;}

    int giveStat(){return status;}

    void reset();

    dealer(const dealer& orig);

    virtual ~dealer();

protected:

    //used to track the card total of the player throughout the game

    int cardTot;



    /*variable status KEY: Determines how the player is dealt with at

     * each point in the game

     status = 1 => player is under 21 and has the option of continuing to hit

     * , if they are finished their status will change to 2

     status = 2 => player is finished with the hand or hit a 21

     status = 3 - player has lost the hand and has no chance of winning

     status = 4 -> player has tied the dealer, bet given back to player

     status = 5 -> Player has beat the dealer and won!*/

    int status;
    //TRUE = ACE, FALSE = NO ACE

    //Crucial as ace in blackjack as multiple meaning depending on the card total

    //of the player, thus must be tracked for each player, but reset for each hand

    bool ace;


};


#endif   /* DEALER_H */


/*

 * File:   game.cpp

 * Author: Amul.bham

 * Purpose : can be found in .h file

 * Created on December 9, 2015, 12:14 AM

 */


//Libraries

#include <vector>

#include <iostream>

#include <ctime>

#include <algorithm>

#include <string>
```

```
#include <fstream>

#include <cstring>

#include "game.h"

#include "Deck.h"

#include "player.h"

#include "dealer.h"


using namespace std;




/*Default Constructor - > must be defined with the amount of players

 playing as a game cannot be started without players


 -The number of players is set, and the x vector has an instance of the player

 class put inside it equal to the number of players, each class holds each players

 information such as balance, name, game status, etc.


 - After, each instance of player, or each player, is taken through a series

 of questions to determines if they are new or returning in the getInfo function


 -This is all done when the game instance is created, therefore setting the game

 up with the proper amount of player instances.*/

blackJack::blackJack(int n) {
    numP = n;

    x.clear(); //Clear the array incase of any faulty memory



    //Vector allows for dynamic creation of player instances

    while(x.size()<numP ){



        //push back a new player object for each player

        x.push_back(player());

    }

    //Loop through each player and obtain their information

    for(int i =0; i<x.size();i++){

        getInfo(x[i]);

    }

}

/*After the player info is obtained and assigned to each player instance,

 each player must place a bet for the upcoming hand

 - Used a char string to get the bet to account of commas and other user input errors

 - The bets are set for each player bet variable as they will be needed as the end

-Used atol to convert the char string after it has been approved*/

void blackJack::setBets(){

   char b[7]; //Used to temp hold the bet place by each player

   cin.ignore();

   //Loop through each player and get their bets for the hand
   for(int i =0; i<x.size();i++){

       cout<<x[i].name<<": "<<endl;

       bet:

       cout<<"Please enter a bet for the current hand ($50.00 min): $";

       cin.getline(b,7);

       //If their bet is invalid, loop back so they can place a valid bet

       if(atol(b)<50 || atol(b)>x[i].giveBal()){

       cout<<"Invalid amount! Bet must be at least $50.00 and no greater than your total
balance!"<<endl;

       cout<<"Your balance is : $"<<x[i].giveBal()<<endl;

       goto bet;

       }else{

       //Set the player bet calling the setBet player function, using atol for conversion

       x[i].setBet(atol(b));

       cout<<"Thank you for your bet"<<endl;

       }

       //Display a border after each player for clarity

       disBord();

   }

   cout<<"Good Luck to all Players!"<<endl;

}


/*The Deal Card Function:
```

```
Used to handle the logic and execution of the first part of the game
which included dealing the cards for each player and finally dealing 1
card for the dealer.

-First each player is looped through twice with a card being drawn from the
deck each time, after the DrawCard() function prints out each card value,
the total is returned and added to the players overall card total

- The checkWinLoss function of the game is called to check for any special
cases, for the first hand it would only be a 21 for any player, set the status
for each player , and then give them back their total for clarity

-After each player is dealt, the dealers first card is revealed, and
total printed, from here the next part of the game is ready to execute*/
void blackJack::dealCards(){
   /*Create a deck of cards at the start of the game, makeDeck() function
    * of card class is setup to handle all the logic for creating the deck */
   deck.makeDeck();
   cout<<"Now Dealing first two cards for each player..."<<endl<<endl;
   int card=0;
   /*Loop through each player, printing their total after each turn,
    used a double for loop, first one to loop through each instance of
    players, and second to loop through 2 cards for each player*/
   for(int i = 0; i<x.size();i++){

      cout<<x[i].name<<": "<<endl;

      for(int j =0; j<2; j++) {

      card = deck.drawCard();

      x[i].setCardT(card);

      }

      cout<<"Total: "<<x[i].giveTotal()<<endl<<endl;

   }

   /*Check for any special conditions after cards are drawn*/

   checkWinLoss();

   //Proceed to dealer first card - display total

   cout<<endl;

   disBord();

   //Set the dealer card total in the instance of the dealer class

   cout<<"Proceeding to dealer hand...."<<endl;

    card = deck.drawCard();

    dealer.setCardT(card);

    cout<<"Dealer Total: "<<dealer.giveTotal()<<endl;

    if (dealer.giveTotal() > 21) dealer.setStat(3);

    else if (dealer.giveTotal() == 21 || dealer.giveTotal() >= 17) dealer.setStat(2);

    else dealer.setStat(1);

}
```

```
/*The hisORstay function was used to handle all the logic of each player
 deciding to hit or stay. Allowed my main to simply execute this part of the program
 and kept the complicated logic bundled away from the abstract logic in my main

 - Based on the result of each player hand, their status must be updated accordingly
 to track if they lost, hit a 21, below 21 , etc. Allowed me to track
 how each player should be dealt with in the final part of the program that determines
 the winners
 */
void blackJack::hitORstay(){
    //Loop through each player
    for(int i = 0; i <x.size();i++){
       char d;
       int card; //Used to temp store the card value returned from the deck
       //Display the current player name
       cout<<x[i].name<<endl;
       /*Only ask for hit or stay if their status is = 1 (continue)
        if they had a 21 from the first cards, then they are skipped*/
       if (x[i].giveStat() == 1){
         /*Continue looping until player status changes to 2 (finished)
          which means they chose to stay, busted, or hit a 21*/
           do{
            cout<<"Card Total: "<<x[i].giveTotal()<<endl;
         cout<<"Would you like to "
         "hit or stay? H/S"<<endl;
         //Determine if user wants to hit or stay
          cin.get(d); cin.ignore();
          //If they hit, draw a new card, add card value
         if (tolower(d) == 'h'){
        card = deck.drawCard();
        x[i].setCardT(card);
        //if not then set their status to 2 for finished and go to the next player
         }else x[i].setStat(2);
    }while(x[i].giveStat() == 1);
    }
    //If the player hits a 21 or busts, let them know
    if (x[i].giveTotal() == 21){
        cout<<"BlackJack!"<<endl;
    }else if (x[i].giveTotal() > 21){
        cout<<"You Busted!"<<endl;
    }
    //Display the final card total for each player
     cout<<"Final Card Total: "<<x[i].giveTotal()<<endl;
     disBord();
 }
 //Change status again based on outcome of the hand
```

```
    checkWinLoss();
    cout<<endl;


}


/*dealerHand -> The 3rd part of the blackJack gameflow
 After each player hand is finished in the previous functions, the rest
 of the dealer hand needed to finish


 I decided to create a new function for it as it had a lot of new logic in terms
 of how to deal with the value and when to stop the hand.


 The dealer has to keep drawing until they are at 17 or higher,
 so I just used a do while loop to keep drawing and adding cards from
 the deck instance until it reached 17 or higher


 After, the status of the dealer must be checked for special conditions such as 21*/
void blackJack::dealerHand(){
    int card;
    cout<<"Current Total: "<<dealer.giveTotal()<<endl;
    cout<<"Now finishing the dealer hand..."<<endl<<endl;
    if (dealer.giveStat() == 1){
        do{
```

```
        card = deck.drawCard();

        dealer.setCardT(card);

        }while(dealer.giveTotal()<17);

    }

        cout<<"New Total: "<<dealer.giveTotal()<<endl;

    //If dealer is busted, set the status to 3 (loss)

    if (dealer.giveTotal() > 21){

        dealer.setStat(3);//set status to loss

        cout<<"Dealer has Busted!"<<endl;

    //If dealer has blackjack, set status to 2, let player know

    }else if (dealer.giveTotal() == 21){

        dealer.setStat(2);

        cout<<"Dealer has a BlackJack!"<<endl<<endl;

    //No special conditions, set status to 2 for check with other players

    }else dealer.setStat(2);

}


/*checkWinLoss function -> Used to check the player hand for any special

 conditions such as a bust or a 21, which in turn allows the program

 to know how to deal with each player in the upcoming functions


 -> Made it separate as it was crucial to the logic of my game

 and had to be executed after each player turn
 -> Having a function to check statuses always ensured that each player

 was in the proper 'state' for the next part of the program

 for Ex: if a player has a status of 3 (which mean they busted) the program

 will know to not ask them if they would like to hit or stay, or to even

 check if they won at the end as all players with status 3 have automatically lost*/

void blackJack::checkWinLoss(){

    disBord();

    //Display results after each hand, any special conditions

    cout<<"Results: "<<endl;

    for (int i = 0; i<x.size();i++){

        cout<<endl<<x[i].name<<":"<<endl;

        if (x[i].giveTotal()== 21){

            cout<<"you hit a blackjack, congratulations!"<<endl;

            x[i].setStat(2); //set status to finished with hand

        }else if (x[i].giveTotal() > 21){

            x[i].setStat(3); //set status to loss if over 21

            cout<<"you busted! Sorry!"<<endl;

            cout<<"Card Total: "<<x[i].giveTotal()<<endl;

        }else { //if not a bust or 21, set to 1 for can continue

            x[i].setStat(1);

            cout<<"Card Total: "<<x[i].giveTotal()<<endl;

    }
```

```
        cout<<endl;

}

}


/*The checkWinLoss was used to set the status of each player after each part

 of the game. However, I still needed a separate function to actually make

 * meaning of those statuses.


 checkWinner -> used to interpret the meaning of each players status

 after the hand has finished, everything checked against dealer status, and a final

 * win, loss, or tie status are initialized -> after, final results can be displayed


 If dealer has a blackjack, all players who wont -> lose

 If dealer is over 21, all players who don't already have a status of 3 ->win

 If dealer and player are set to status of 2, then the values are compared,

 whoever has a higher card value wins


 -Since each players status of winning or losing is separate from the other

 players, it must be determined independently with each players

 circumstances taken into account

    -Ex: If a player busts, and the dealer busts after, the player still loses,

 because the player busted first, however any other players who did not bust will win*/

void blackJack::checkWinner(){
```

```
for(int i = 0; i< x.size(); i++){
   //First, if any players are over 21, set them to loss prior to checking anything
  if (x[i].giveTotal() >21) x[i].setStat(3);


   //If they have a valid hand, check against each possible dealer hand
  //set their status for win,loss,or tie, the final status of the hand
  if(x[i].giveStat() != 3){
       //If dealer has 21, any player who does not, is set to loss
    if (dealer.giveTotal() == 21){
        if (x[i].giveTotal() != 21 ){
            x[i].setStat(3); //set for loss if !21
        }else x[i].setStat(4); //If they do, then set to 4 for draw


    //If the dealer has not busted, or 21, then the dealer card total
    //must be checked against each player who has not busted
    }else if (dealer.giveTotal() < 21 ){
        if (x[i].giveTotal()> dealer.giveTotal()){
            x[i].setStat(5); //player total is higher = win, if = then tie
        }else if(x[i].giveTotal() == dealer.giveTotal()){x[i].setStat(4);}
         else x[i].setStat(3); //If lower, then the player is set to a loss
    //If dealer has busted, then every player who has NOT busted, wins
    }else if (dealer.giveTotal()> 21) x[i].setStat(5);
}
```

```
    }

}

/*After the status of each player is interpreted and checked against the dealer,

 a final status is set. Finally the results are ready to be displayed for the hand

 in text format, updated balances are displayed after bets are taken into account,

 and the hand ends


 showResults -> I needed a function specifically for 'wrapping up' the hand

 * and to give a overall report to the players of the implications

 *

 * -I tried to make only for displaying results, tried not to calculate anything

 * new in this functions

 *

 * - Based on the final status of each player(1,2,3), they are told of the results

 * and their new balances are calculated accordingly */


void blackJack::showResults(){

    cout<<"Results of Hand and Updated Balances: "<<endl;

    disBord();

    //Loop through each players status

    for (int i = 0; i<x.size(); i++){

        //5 = win; bet is added to player balance

        if (x[i].giveStat() == 5){
          x[i].win();

        cout<<x[i].name<<" you have won!"<<endl;

        cout<<"Balance: $"<<x[i].giveBal()<<".00"<<endl;

        //3 = loss, bet subtracted from balance

        }else if (x[i].giveStat() == 3){

          x[i].loss();

        cout<<x[i].name<<" you have lost!"<<endl;

        cout<<"Balance: $"<<x[i].giveBal()<<".00"<<endl;

        //4 = tie; player receives bet back

        }else{

        cout<<x[i].name<<" you have tied the dealer!"<<endl;

        cout<<"Balance: $"<<x[i].giveBal()<<".00"<<endl;

        }

        //output a border after each player so the results are clear

        disBord();

    }

}


/*After each hand, the player is given the option to play a new hand

 or exit the game, if they wanted to play a new hand, I needed to make sure

 their player instance was set up again for a new hand


 newHand -> Created to reset the dealer and each player
 instance card total for the next hand, also checks if any of the players

 have run out of chips, if so, they must buy in more



 Basically I needed to ensure after each hand, the players and dealer

 variables were reset to prevent overlap between hands



 -Only certain variable needed to be reset, others such as the player.name and balance

 were maintained, therefore you could not just create a new instance of

 the blackJack game as then you would lose player information from previous hand*/

void blackJack::newHand(){

    long int y; //Temp hold buy in value

    //Call the dealer reset function first, this resets all the crucial

    //dealer variables

    dealer.reset();

    /*Then loop through each player, reset their variables, and determines

     if they ran out of money and need to buy in, set their new balance*/

    for (int i = 0; i< x.size(); i++){

        x[i].reset(); //reset each players card values

        if (x[i].giveBal() < 0){

            do{ //Loop while invalid amount

             cout<<x[i].name<<endl;

             cout<<"You must purchase more chips to continue as you are below 00.00!"<<endl;

             cout<<"Amount(100.00 min::10,000.00 max): "; cin>>y; cin.ignore();
```

```
          }while(y < 100 || y> 10000 );

          x[i].setBal(y); //Set players new balance

       }

   }

}


/*disBord() -> used to separate off information and parts of the game for the player

 to ease with clarity and readability, made it easier to insert border versus

 typing out a border each time I needed one, also kept the look

 of the program more of a similar feel*/

void blackJack::disBord(){

cout<<"**************************************************************"<<endl;

}


blackJack::blackJack(const blackJack& orig) {

}


/*After a new instance of a player class is created, the player information

 for that instance must be obtained, based on if they are new or returning,

 the program must make a wealth of choices to determine how to obtain the

 * player information for that given instance


 getInfo -> Used to handle getting all player information, each player instance
```

```
* if passed through it, determined if the player is new or returning
*
* -> From here, the program either reads out the player information on file
* and asks the user to select a account - this account number is = to the binary
* location number of each player in file, set at the end of each game
* OR -> obtains the new players information, assigns it to their instance,
* and then set their binary number position to the total size of the file + 1;
*
* -Also, if a player is already loaded into the game by someone else, the function
* must check that no 2 players are on the same account
*
* -I used a fstream object to read in the player information, I used a temp
* structure to store each player information, display it, and then increment to the
* next section (all done as a binary file, as I had to store the name, and int balance)
* - If player returning, I read in their information from structure to
* their player class
*
* - At the end of game, their names and balances are stored again in the temp
* structure, 1 by 1, and then written back into the file "player.txt" in spot
* = to their binary number position
*/


void blackJack::getInfo(player &curr){
```

```
//Variables used to store player decisions

bool t = true;

char c;

//Determine if the user is returning or if it is their first time

_rORn:

cout<<"Player "<<cot+1<<":"<<endl;

cout<<"Are you returning or a new player? (r/n)"<<endl;

cin>>c; cin.ignore();

//Needs to be valid as this part is crucial in determining how to deal with that player

if (tolower(c) != 'r' && tolower(c) != 'n') {

    cout<<"Invalid option!"<<endl; goto _rORn;

}

cot++;

int count = 1;

/*if continuing, read all the players currently in the file and ask

 the user to select which account they would like to play on

 * status is set to NOT NEW, program will rewrite this players new

 * information in the same spot in file AFTER game finishes */

if (tolower(c) == 'r'){

    //Open the player file for input, and as a binary file(has structures)

    player_info.open("player.txt", ios::in | ios::binary);

    if (!player_info) {cout<<"Error opening file!"<<endl;}
```

```
//Open the player file for binary input

cout<<"The current players on record are ..."<<endl;

disBord();

//Static cast the binary data into the temp structure for the size of the

//entire temp structure

player_info.read(reinterpret_cast<char *>(&temp), sizeof(temp));


/*Read out all the players, incremented by byte size, each player

information being stored in a structure, until the end of the file is reached*/

while(!player_info.eof()){

    cout<<"Player "<<count<<endl;

    cout<<"Name: "<<temp.name<<endl;

    cout<<"Balance: $"<<temp.bal<<endl;

    player_info.read(reinterpret_cast<char *>(&temp), sizeof(temp));

    count++;

    disBord(); //display a border between each player information

}

//Close the file to prevent memory leaks

 player_info.close(); int ply; //temp value to store player # input

 //Allow the user to choose an account to play on

 do{ //Loop until value is valid, no repeats are allowed

 cout<<"Enter the player number of the account you would"

        " like to play on: ";
 cin>>ply; ply -=1;

 //Check against other players, make sure 2 people not on same account

 for(int i =0; i<x.size();i++){

     if(ply == x[i].givePos()){

         t = false;//set loop condition

         cout<<"This account is already in use!"<<endl;

         break; //If same, break here, start loop again

     }else {t = true;}

 }

 //If ply is valid, t set to true, loop ends

 }while(!t);


 //Read in the player information based on the account choosen

 player_info.open("player.txt", ios::in|ios::out|ios::binary);

 //Seek to the memory location of the user account, based on the num

 player_info.seekg(sizeof(temp)*(ply), ios::beg);

 //Read in the player information, including their name and balance

 player_info.read(reinterpret_cast<char *>(&temp),sizeof(temp));

 /*After appropriate binary information for given player is determined

  read it into the temp structure for temp storage*/


 /*Then copy each temp structure variable to the required variables

  in the given players player instance - read in name, balance, and binary num*/
```

```
    strcpy(curr.name, temp.name); curr.setBal(temp.bal);

    curr.setBin(ply); curr.setNew(1); //Set player status at NOT NEW


    cout<<"Welcome Back "<<curr.name<<"!"<<endl;

    //Determine if the user would like to purchase more coins

    cout<<"Your current balance is $"<<curr.giveBal()<<endl;

    cout<<"would you like to buy in more? (Y/N)"<<endl;

    cin.ignore(); cin.get(c);

    if (tolower(c) == 'y'){

        cout<<"Enter how much more you like to buy in($10,000.00 Limit) or type 0 for none"<<endl;

    //Get the buy in as a cstring to prevent run time errors, convert to int using atoi.

      cin.ignore(); long int c; cin>>c; c += curr.giveBal();

    curr.setBal(c);} //Add the buy in to the total player balance

    //Remind user of updated balance, close player file

    cout<<"Your current balance is $"<<curr.giveBal()<<endl;

    player_info.close();


/*if the user is new, obtain new player information, set player status

 to NEW, this will tell the program to APPEND the player.txt file

 to create a new section for the new player versus overriding

 another players file*/


}else{
    //Open the file just to get the size of it in bytes

player_info.open("player.txt", ios::in | ios::binary);

curr.setNew(0);  curr.setBin((sizeof(player_info)/sizeof(temp))+1);

player_info.close();

char r;

//Name stored as cstring as strings cannot be written to file

cout<<"A new player! Terrific! First I'll need your name: "<<endl;

cin.getline(curr.name,25); //get the new players name



//If they want to hear rules, call disRules function

cout<<"Would you like to hear the rules? (Y/N)"<<endl;

cin.get(r);cin.ignore();

if(tolower(r)== 'y'){

    disRules();

}

disBord();

//Determine how much the player wants to buy into game, loop till valid amount

_buyin:

cout<<"How much would you like to buy in?($100.00 minimum::$10,000.00 limit)"<<endl;

long int b;

cin>>b; if (b<100 || b>10000) {cout<<"Invalid amount!"<<endl;goto _buyin;}

curr.setBal(b); //set the new players balance in their instance

}
```

```
   disBord();

}

/*After players are finished playing, I needed a function to handle the logic

 * for writing the updated players information back into player.txt

 *

 * writeInfo -> Loops through each player and does the followings

 *

 * 1. copy vital player information(balance,name,binary position) into a temp

 * player_info structure for storage

 *

 * 2. Write player data(stored in a structure) into the file for the size of

 * structure and in binary so we can store separate instances of different data types

 *

 * 3. if player is new -> append the file -> create new instance in the file

 *  if player is returning -> overwrite the position in the file = the players

 *  binary number position * size of the structure(the size of the temp structure

 * is basically the measurement for incrementing between players in the

 * binary file, thus their position can always be determined given that a value is tracked

 * for each player of their position), this will seek the file

 *  at the spot that the player previously had their information stored

 *  (Random Access Memory), and only overwrite their previous data

 */

void blackJack::writeInfo(){
```

```
for(int i =0; i<x.size(); i++){

   //Copy the name, position, and updated balance to the temp structure for storage

   strcpy(temp.name, x[i].name); //use string copy for char string instead of looping

   temp.bal = x[i].giveBal();

   temp.bN = x[i].givePos();


   //Open up player.txt file for output, and in binary

   player_info.open("player.txt", ios::in | ios::out | ios::binary);


  //If a players giveNew() functions is true, their information is appended

   //to the file

   if(x[i].giveNew()== true) {

     player_info.close();//close and reopen the file for append mode

     player_info.open("player.txt",ios::binary| ios::app);

     player_info.write(reinterpret_cast<char *>(&temp),sizeof(temp));

   //For returning players, seek to the position in the file previously

     //occupied by the player, and overwrite the previous information

   }else{

     player_info.seekp((x[i].givePos()) * sizeof(temp), ios::beg);

     player_info.write(reinterpret_cast<char *>(&temp),sizeof(temp));

   }

}   //close the player.txt file

    player_info.close();
```

```
}

blackJack::~blackJack() {

}


/*disRule -> Used to display all the rules of the game given a player

 does not no how to player and prompts to hear them, simply a wall of text,

 does not consist of anything significant, but made it cleaner to separate it

 and only call it if a player prompts the program to do so.*/

void blackJack::disRules(){

    cout<<"Basically the point of the game is to get to 21 or as close as\n"

                "possible without going over, otherwise you lose"

                "\nIn addition, if my card total is higher than yours"

                " you also lose"<<endl;

    disBord();

        cout<<"I will deal 2 cards from a regular deck of cards \n"

                "depending on the total value of these cards, you must\n"

                "decide if you want another card or stay where you are (hit) \n"

                "Also, note that my job as a dealer is to always hit if my card\n"

                "value is below 17,if i bust (go over 21) then you automatically\n"

                "win. So its your choice whether you want to pursue a higher card "

                "value(at the risk of busting)\nor stay and wait for the dealer to bust\n"

                "If we score a tie, then the hand ends in a draw \n";

        disBord();
        cout<<"Also, Jack, Queens, King all have a value of 10\n"
                "while the Ace has a value of either 1 or 11\n"
                "All other cards represent their face value\n";
        cout<<"You will buy in an amount of your choosing which will be tracked throughout the
game"<<endl;
        cout<<"At the start of each hand, you will place a bet: "<<endl;
        cout<<"If you win, you win 2x your original bet"<<endl;
        cout<<"A loss means you lose the amount of the bet"<<endl;
        cout<<"And finally a draw or PUSH results in no loss of money"<<endl;
        cout<<"Minimum bets: $ 50.00   "<<"Maximum bets: $ ACC BAL  "<<endl<<endl;
        disBord();
        cout<<"Also note, you are playing against the dealer, so the card values"
                "\nof other players do not affect if you win or lose, you are"
                "\nonly competing against the dealer!"<<endl;


}
/*
 * File:   Deck.cpp
 * Author: Amul.bham
 *
 * Created on December 8, 2015, 7:37 PM
 */
using namespace std;
```

```
#include <vector>

#include <iostream>

#include <ctime>

#include <cstring>

#include <algorithm>

#include <string>

#include <fstream>

#include "Deck.h"



Deck::Deck() {

    card = 0;

    face = "";

    value = "";

    total = 0;

    //Create the deck of card as soon as a deck object is created

    makeDeck();

}


void Deck::makeDeck(){

    //Initialize a vector with values 2-53 to represent each card in a 52 card deck
    for(y=2;deck.size() < tCards ;y++){

        deck.push_back(y);

        }

    //After creating a array of 2-53, send to function to random shuffle the cards

        shuffleCards();

}


int Deck::drawCard(){

    /*Re shuffle the cards only if the card count goes over 30, this

      allows the game to not be predictive, also let the players know

      so that its fair and the players are aware a new deck is being used*/

     if (card>30) {shuffleCards();}


    //Reset the total prior to each new card drawn

    total = 0;


    //Draw a card from deck and first determine the face and the numerical value

       getFace();


    //Then the numerical value of the card must be translated to a string

       getValue();


    //Display the drawn card for the user
```

```
      dispCards();



       /*Return the player card back to the player,increment to the next card

        this makes it so the next card is ready to be drawn, the total is returned

        so that an overall card total can be tracked for each player*/

       card++;

       return total;


}


void Deck::shuffleCards(){

    /*Let the players know any time the deck is re shuffled,

     reset the card count to 0*/

    cout<<"Shuffling Cards..."<<endl<<endl;

    card= 0; //Reset the card number when the deck re shuffles

    srand (time(0)); //Random number seed for random_shuffle function


    /*Deck is shuffled using random_shuffle temp inside the algorithm

     standard library, this function only works on vectors, so it gave me

     all the more reason to put the deck inside in vector*/

    random_shuffle(deck.begin(),deck.end());

}
```

```
void Deck::getFace(){
        /*Set the total as equal to the card, then the value of the card
         is determined from the total, this allowed the NUMBER value of the card
         to remain separate from the CARD NUMBER, I made all calculations
         on the total number and used the deck to represent where the card
         was in the deck*/
        total = deck[card];


        /*Determine suit and card total based on value of drawn card 2-53
          then I subtracted off a certain value depending on its range
         to make the values all a number between 2-13, this made it possible
         to use a switch to display the card name, so first the suit is determined
         based on its range in the deck, then the value is made a value between
         2-13, then a value is printed out alongside the suit*/
          if (deck[card]<=14){
              face = "Hearts";
          }else if (deck[card]>14 &&deck[card]<=27){
              total -= 13;
              face = "Diamonds";
          }else if (deck[card]>27 && deck[card]<=40){
              total -= 26;
              face = "Cloves";
          }else if (deck[card]>40){
            total -=39;

            face = "Spades";

          }


}


void Deck::dispCards(){

    /*Function used purely to display the name and suit of each

     card that is drawn, since they are not suppose to be bound to any thing

     in real life, that was how I made my function, display the card

     return only what the player needs to know, which is the card value*/

    cout<<value<<" of "<<face<<endl;

}




/*getValue -> Used to determine the value of the card in terms of its string value

 name, and stores it in the value string object for display along with the face


 By normalizing all cards to a value between 2-13, I was able to use a switch

 statement to determine the string value


 The card total is not changed and only used as a condition to determine its string value*/

void Deck::getValue(){
```

```
/*Use a switch to determine the string value of the card

 based on the numerical value*/

switch(total){

        case 2:

            value="Two";

            break;

        case 3:

            value = "Three";

            break;

        case 4:

            value = "Four";

            break;

        case 5:

            value = "Five";

            break;

        case 6:

            value = "Six";

            break;

        case 7:

             value = "Seven";

            break;

        case 8:

            value = "Eight";
            break;

        case 9:

            value = "Nine";

            break;

        case 10:

            value = "Ten";

            break;

        case 11:

            value = "Jack";

            break;

        case 12:

            value = "Queen";

            break;

        case 13:

            value = "King";

            break;

        case 14:

            value = "Ace";

            break;

    }


}

Deck::Deck(const Deck& orig) {
}


Deck::~Deck() {

}

/*

 * File:   player.cpp

 * Author: Amul.bham

 *

 * Created on December 8, 2015, 9:12 PM

 */


#include <vector>

#include <iostream>

#include <ctime>

#include <algorithm>

#include <string>

#include <fstream>

#include <cstring>


#include "player.h"

using namespace std;


/*Used to check the card value of the player after each card draw,
 and then determine the status of the player for the next card draw


 Then the status is used by the calling object to determine if the player

 has lost, skip the next card draw, hit a black jack, etc


 -> this made it easier to determine which players needed to be skipped in the

 game flow*/

void player::setStat(){

    if (cardTot < 21) status = 1; //continue compare

    else if (cardTot == 21) status = 2; //finished

    else if (cardTot > 21) status = 3; // lost

}


/*reset() -> overrides the dealer reset function as it has to reset

 additional player values after each hand such as the bet,

 called to reset player variables that are required to be a certain value

 at the start of each hand

 * for Ex: card total MUST be reset for each player

 * or the card total would be carried over into the next hand! */

void player::reset(){

    cardTot = 0;

    status = 1;

    ace = false;
    bet = 0;

}



player::player(const player& orig) {

}



player::~player() {

}



/*

 * File:   dealer.cpp

 * Author: Amul.bham

 *

 * Created on December 12, 2015, 7:51 PM

 */



#include <vector>

#include <iostream>

#include <ctime>

#include <algorithm>

#include <string>

#include <fstream>

#include <cstring>
```

```
#include "dealer.h"


using namespace std;

//If a dealer is created, reset all values, make ace false

dealer::dealer() {

    cardTot = 0;

    status = 1;

    ace = false;

}

/*Used by the player class as well.

 ->Took the card value returned by the deck object,

 ->normalizes the value in the case of a face card

 ->Checks for ace, if so, sets player ace variable to true

 ->Checks card value and assigns a value of 1 or 11 to the card

 ->After doing so, value is added to the overall player total*/

void dealer::setCardT(int card){

    //If ace, check value, set value for ace -> set status of ace

    //if 1 ace:false, if 11 ace::true

    if (card >= 14){

        if (giveTotal()<= 10){ card = 11; setAce(1);}

        else{ card = 1; setAce(0);}

    }else if (card>10 && card < 14){

        card = 10;
    }

    //After card is normalized, it is sent to addTotal to be added to the overall player total

    addTotal(card);

}



//Function created to resets important dealer variables whenever called

void dealer::reset(){

    cardTot = 0;

    status = 1;

    ace = false;

}

//Dealer has own logic for determining its status, as its goals are different

//than the players in some instances, so therefore this status function is

//overridden by the player class

void dealer::setStat(){

    if (cardTot < 17) {status = 1;}

    else if (cardTot >=17 && cardTot <= 21){status = 2;}

    else if (cardTot > 21) {status = 3; }



}

//After the total has finally be determined, it is added to the player total

void dealer::addTotal(int card){

    cardTot += card; //add the card value
    //Check for Ace! If ace is true, and value is over 21
    //Then the ace value is switched to a 1 instead of 11!
    if (giveAce()==true && cardTot > 21) {
        cardTot -=10;
        ace = false;
    }
    //Set the status after adding total as the total is what dictates the status
    setStat();
}
dealer::dealer(const dealer& orig) {
}


dealer::~dealer() {
}
```