Ethan Hua

Rodrigo Becerril Ferreyra

California State University, Long Beach

Professor Minhthong Nguyen

Blackjack Final Project

10 December 2021

Introduction

The purpose of this project is to test our skills and knowledge acquired this semester. Specifically, our task is to implement the game of Blackjack (also known as 21) in C++. This project tests our knowledge of object-oriented programming (classes, objects, etc.), data structures, and file management. Additionally, this project tests our teamwork and communication skills.

The following is a brief summary of the project requirements: the game is to be implemented on the command line, to avoid any GUI-related issues. The game will follow a standard game of Blackjack (rules available here) played with one deck. The game is played by the player against the dealer. Along with the standard rules, the player is allowed to split his or her hand if the first cards he or she receives are the same value; this split is only allowed once, meaning that the player can have a maximum of two hands. The total amount of money the player has won or lost is stored in an "account" file, which also stores a user ID and the number of games won.

Program Analysis and Design

There are three classes required by the project specifications:

1. an `Account` class.
2. a `Card` class.
3. a `Player` class.

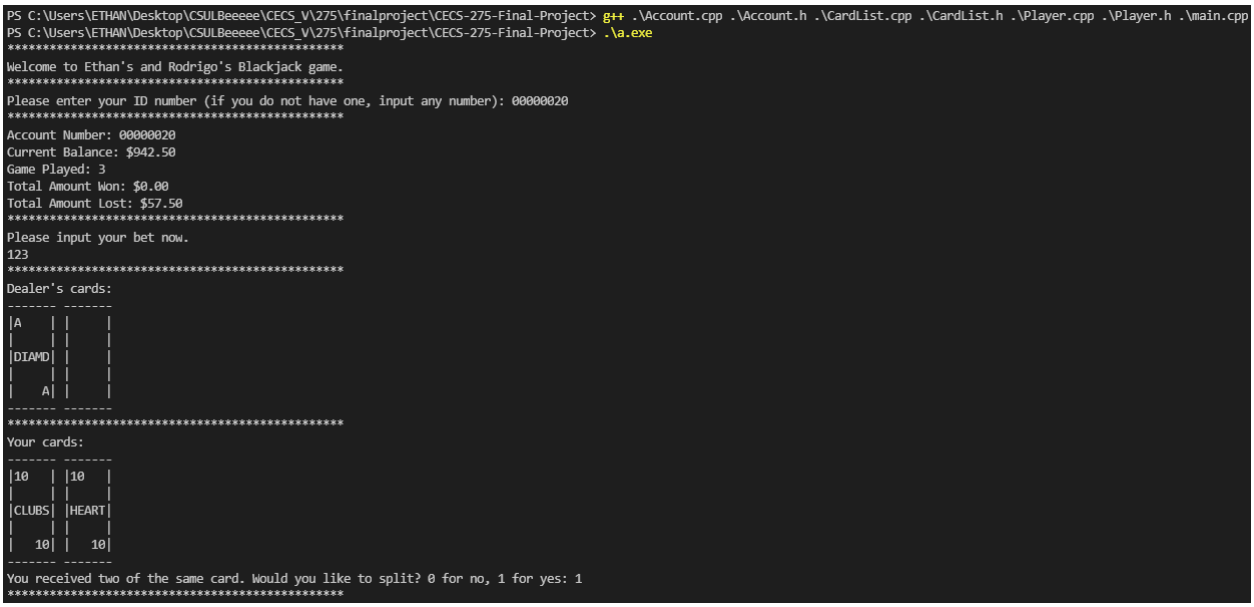
The Account class is in charge of file-handling functions. It makes sure to save the results of every game into a file so it can be retrieved later. The Card class (named CardList) is implemented as a linked list; a list of Cards can represent a player's hand or a deck of cards. The

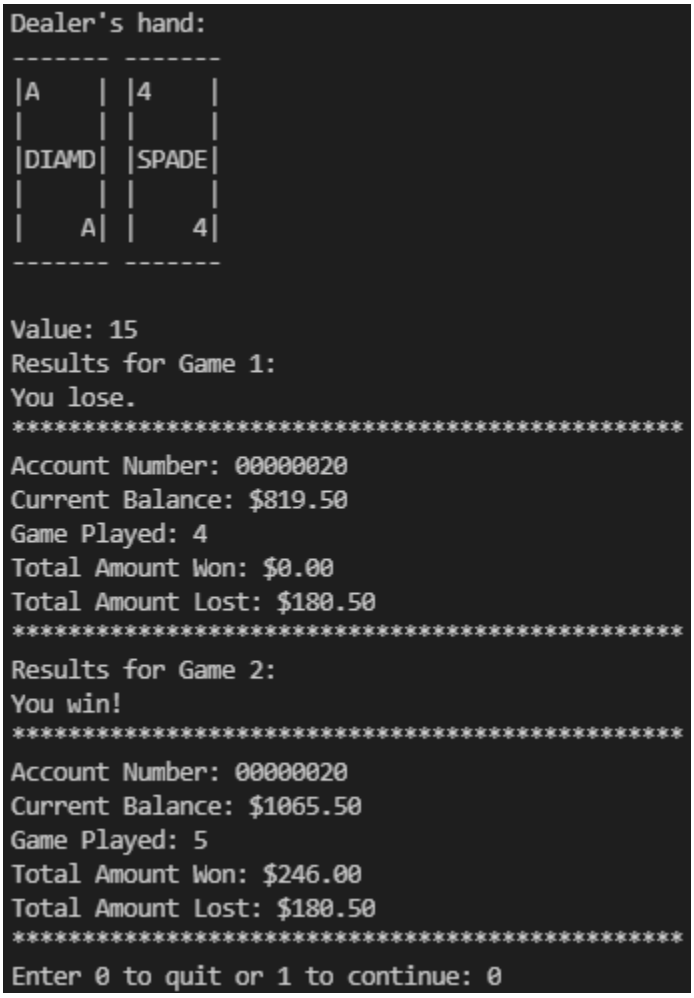
Player class implements functions that players perform, such as the action the user will take (hit, stand, split). All these classes work together to create a game.

The game itself follows a standard Blackjack format: first, a deck is created. Then, cards are dealt to both the player and the dealer. The dealer then reveals one of his cards, and the player is given a choice to perform an action: hit or stand (additionally split if the cards are the two cards dealt have the same value). If the player chooses to hit, he or she is dealt another card and asked to perform an action. This continues until the player chooses to stand or the value of the player's hand reaches or exceeds 21. Meanwhile, the dealer automatically draws cards until the value of his hand reaches or exceeds 17. After both players have played, the hands are compared, and the game is finished.

Screenshots

Game 1 (KNOWN ACCOUNT, SPLIT, HIT ON BOTH HANDS, LOST 1, WON 2)





Game 2 (GENERATED 8-DIGIT ACCOUNT FROM 7-DIGIT INPUT, 1 HAND, STAND, WON)

Game 3 (CONTINUOUS PLAY)

```
**********************************************
Now playing: hand 1.
------- -------
|4    | |J    |
|     | |     |
|CLUBS| |SPADE|
|     | |     |
|    4| |    J|
------- -------

Value: 14
Enter 0 for hit, 1 for stand: 1
Dealer's hand:
------- -------
|5    | |2    |
|     | |     |
|DIAMD| |HEART|
|     | |     |
|    5| |    2|
------- -------

Value: 7
Results for Game 1:
You win!
**********************************************
Account Number: 15342646
Current Balance: $1240.00
Game Played: 1
Total Amount Won: $240.00
Total Amount Lost: $0.00
**********************************************
Enter 0 to quit or 1 to continue: 1
Please input your bet now.
321
**********************************************
Dealer's cards:
------- -------
|J    | |     |
|     | |     |
|CLUBS| |     |
|     | |     |
|    J| |     |
------- -------
**********************************************
```

Account with many games played on it

```
**********************************************
Account Number: 53149756
Current Balance: $136387.00
Game Played: 17
Total Amount Won: $42980.00
Total Amount Lost: $6593.00
**********************************************
Enter 0 to quit or 1 to continue: 0
```

Files updating

acc_15613249.txt

```
1   Account Number: 15613249
2   Current Balance: $1064.00
3   Game Played: 2
4   Total Amount Won: $500.00
5   Total Amount Lost: $1000.00
6
```

```
Please enter your ID number (if you do not have one, input any number): 15613249
**********************************************
Account Number: 15613249
Current Balance: $1064.00
Game Played: 2
Total Amount Won: $500.00
Total Amount Lost: $1000.00
**********************************************
```

```
**********************************************
Please input your bet now.
15
**********************************************
Dealer's cards:
------- -------
|4    | |     |
|     | |     |
|CLUBS| |     |
|     | |     |
|    4| |     |
------- -------
**********************************************
Your cards:
------- -------
|7    | |A    |
|     | |     |
|HEART| |HEART|
|     | |     |
|    7| |    A|
------- -------
**********************************************
Now playing: hand 1.
------- -------
|7    | |A    |
|     | |     |
|HEART| |HEART|
|     | |     |
|    7| |    A|
------- -------

Value: 18
Enter 0 for hit, 1 for stand: 1
Dealer's hand:
------- -------
|4    | |6    |
|     | |     |
|CLUBS| |CLUBS|
|     | |     |
|    4| |    6|
------- -------

Value: 10
Results for Game 1:
You win!
**********************************************
Account Number: 15613249
Current Balance: $1094.00
Game Played: 3
Total Amount Won: $530.00
Total Amount Lost: $1000.00
**********************************************
Enter 0 to quit or 1 to continue: 0
```

acc_15613249.txt

```
1   Account Number: 15613249
2   Current Balance: $1094.00
3   Game Played: 3
4   Total Amount Won: $530.00
5   Total Amount Lost: $1000.00
6
```

9-digit ID error handling (also, a tie game)

```
**********************************************
Welcome to Ethan's and Rodrigo's Blackjack game.
**********************************************
Please enter your ID number (if you do not have one, input any number): 100000000
NumOutOfBounds; Value received: 100000000; expected value between 1 and 99999999 (inclusive).
Generating new account with truncated ID.
**********************************************
Account Number: 00000001
Current Balance: $1000.00
Game Played: 0
Total Amount Won: $0.00
Total Amount Lost: $0.00
**********************************************
Please input your bet now.
123
**********************************************
```

```
Enter 0 for hit, 1 for stand: 0
------- ------- -------
|9    | |2    | |5    |
|     | |     | |     |
|SPADE| |HEART| |HEART|
|     | |     | |     |
|    9| |    2| |    5|
------- ------- -------
Value: 16
Enter 0 for hit, 1 for stand: 1
Dealer's hand:
------- -------
|6    | |J    |
|     | |     |
|DIAMD| |DIAMD|
|     | |     |
|    6| |    J|
------- -------

Value: 16
Results for Game 1:
It's a tie!
************************************************
Account Number: 00000001
Current Balance: $938.50
Game Played: 1
Total Amount Won: $0.00
Total Amount Lost: $61.50
************************************************
Enter 0 to quit or 1 to continue: 0
```

Account with many games on it

```
************************************************
Account Number: 53149756
Current Balance: $136387.00
Game Played: 17
Total Amount Won: $42980.00
Total Amount Lost: $6593.00
************************************************
Enter 0 to quit or 1 to continue: 0
```

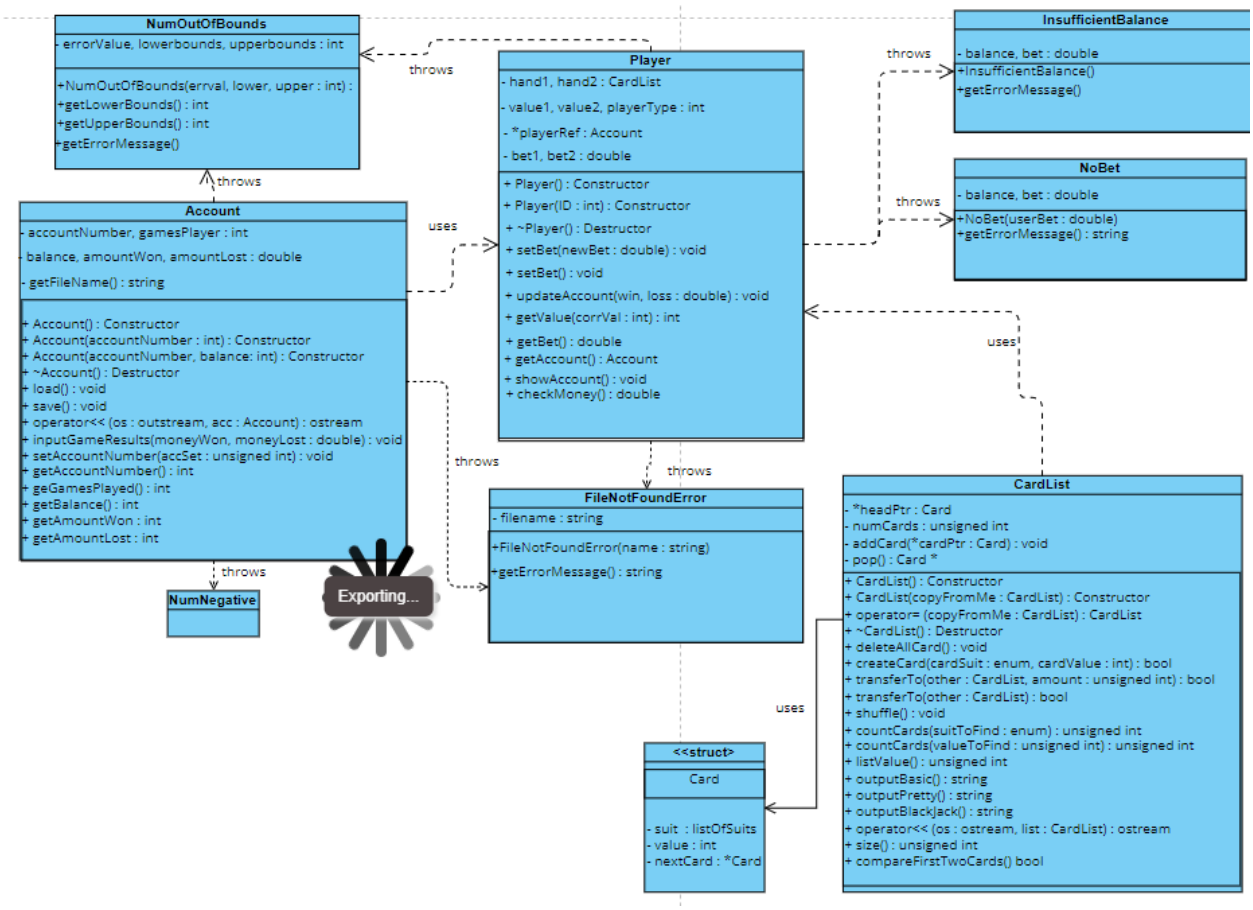
Choosing not to split

```
**********************************************
Your cards:
------- -------
|4    | |4    |
|     | |     |
|CLUBS| |SPADE|
|     | |     |
|    4| |    4|
------- -------
You received two of the same card. Would you like to split? 0 for no, 1 for yes: 0
**********************************************
Now playing: hand 1.
------- -------
|4    | |4    |
|     | |     |
|CLUBS| |SPADE|
|     | |     |
|    4| |    4|
------- -------

Value: 8
Enter 0 for hit, 1 for stand: 0
------- ------- -------
|5    | |4    | |4    |
|     | |     | |     |
|CLUBS| |CLUBS| |SPADE|
|     | |     | |     |
|    5| |    4| |    4|
------- ------- -------
Value: 13
Enter 0 for hit, 1 for stand: 0
------- ------- ------- -------
|10   | |5    | |4    | |4    |
|     | |     | |     | |     |
|DIAMD| |CLUBS| |CLUBS| |SPADE|
|     | |     | |     | |     |
|   10| |    5| |    4| |    4|
------- ------- ------- -------
Value: 23
Dealer's hand:
------- -------
|8    | |4    |
|     | |     |
|CLUBS| |HEART|
|     | |     |
|    8| |    4|
------- -------

Value: 12
Results for Game 1:
Bust! You lose.
```

Diagram


NumOutOfBounds
- errorValue, lowerbounds, upperbounds : int
+NumOutOfBounds(errval, lower, upper : int) :
+getLowerBounds() : int
+getUpperBounds() : int
+getErrorMessage()
throws
Player
- hand1, hand2 : CardList
- value1, value2, playerType : int
- *playerRef : Account
- bet1, bet2 : double
+ Player() : Constructor
+ Player(ID : int) : Constructor
+ ~Player() : Destructor
+ setBet(newBet : double) : void
+ setBet() : void
+ updateAccount(win, loss : double) : void
+ getValue(corrVal : int) : int
+ getBet() : double
+ getAccount() : Account
+ showAccount() : void
+ checkMoney() : double
throws
InsufficientBalance
- balance, bet : double
+InsufficientBalance()
+getErrorMessage()
throws
NoBet
- balance, bet : double
+NoBet(userBet : double)
+getErrorMessage() : string
throws
Account
- accountNumber, gamesPlayer : int
- balance, amountWon, amountLost : double
- getFileName() : string
+ Account() : Constructor
+ Account(accountNumber : int) : Constructor
+ Account(accountNumber, balance: int) : Constructor
+ ~Account() : Destructor
+ load() : void
+ save() : void
+ operator<< (os : outstream, acc : Account) : ostream
+ inputGameResults(moneyWon, moneyLost : double) : void
+ setAccountNumber(accSet : unsigned int) : void
+ getAccountNumber() : int
+ geGamesPlayed() : int
+ getBalance() : int
+ getAmountWon : int
+ getAmountLost : int
uses
throws
NumNegative
Exporting...
throws
FileNotFoundError
- filename : string
+FileNotFoundError(name : string)
+getErrorMessage() : string
throws
uses
CardList
- *headPtr : Card
- numCards : unsigned int
- addCard(*cardPtr : Card) : void
- pop() : Card *
+ CardList() : Constructor
+ CardList(copyFromMe : CardList) : Constructor
+ operator= (copyFromMe : CardList) : CardList
+ ~CardList() : Destructor
+ deleteAllCard() : void
+ createCard(cardSuit : enum, cardValue : int) : bool
+ transferTo(other : CardList, amount : unsigned int) : bool
+ transferTo(other : CardList) : bool
+ shuffle() : void
+ countCards(suitToFind : enum) : unsigned int
+ countCards(valueToFind : unsigned int) : unsigned int
+ listValue() : unsigned int
+ outputBasic() : string
+ outputPretty() : string
+ outputBlackJack() : string
+ operator<< (os : ostream, list : CardList) : ostream
+ size() : unsigned int
+ compareFirstTwoCards() bool
uses
<<struct>>
Card
- suit : listOfSuits
- value : int
- nextCard : *Card

```cpp
/**
 * Main function file; game logic goes here.
 * CECS 275 - Fall 2021
 * @author Rodrigo Becerril Ferreyra
 * @author Ethan Hua
 * @version 1
 */

#include <iostream>
#include "CardList.h"
#include "Player.h"

int main()
{
    int useroption; bool splitflag;
    int hand1val, hand2val, dealval;
    double userBet;
    int validBet = 0;
    // build deck
    CardList deck;
    // create 52 cards
    for(int i = 1; i <= 13; ++i)
    {
        deck.createCard(CardList::CLUBS, i);
        deck.createCard(CardList::DIAMONDS, i);
        deck.createCard(CardList::HEARTS, i);
        deck.createCard(CardList::SPADES, i);
    }

    std::cout << "************************************************\n";
    std::cout << "Welcome to Ethan's and Rodrigo's Blackjack game.\n";
    std::cout << "************************************************\n";

    std::cout << "Please enter your ID number (if you do not have one, input"
        << " any number): ";
    std::cin >> useroption;

    Player player(useroption);
    Player dealer;

    std::cout << "************************************************\n";
    player.showAccount();
    std::cout << "************************************************\n";

    // do-while loops until user decides to exit program
    do
    {
        splitflag = false;
        deck.shuffle();
        std::cout << "Please input your bet now.\n";
        std::cin >> userBet;
        try
        {
            player.setBet(userBet);
        }
        catch(Player::InsufficientBalance &e)
        {
            std::cerr << e.getErrorMessage() << "\n";
            do
            {
                std::cout << "Please input a valid bet.\n";
                std::cin >> userBet;
                try
                {
                    player.setBet(userBet);
```

```
                    validBet = 1;
                }
                catch(Player::InsufficientBalance &e)
                {
                    std::cerr << e.getErrorMessage() << "\n";
                }
                catch(Player::NoBet &e)
                {
                    std::cerr << e.getErrorMessage() << "\n";
                }
            } while(validBet == 0);
        }
        catch(Player::NoBet &e)
        {
            std::cerr << e.getErrorMessage() << "\n";
            do
            {
                std::cout << "Please input a valid bet.\n";
                std::cin >> userBet;
                try
                {
                    player.setBet(userBet);
                    validBet = 1;
                }
                catch(Player::InsufficientBalance &e)
                {
                    std::cerr << e.getErrorMessage() << "\n";
                }
                catch(Player::NoBet &e)
                {
                    std::cerr << e.getErrorMessage() << "\n";
                }
            } while(validBet == 0);
        }
        std::cout << "************************************************\n";

        // deals two cards to each
        std::cout << "Dealer's cards:\n";
        dealer.drawCard(0, deck, 2);
        std::cout << "************************************************\n";
        std::cout << "Your cards:\n";
        player.drawCard(0, deck, 2);

        if(player.checkMoney() >= player.getBet()*2)
        {
            if(player.splitCondition())
            {
                std::cout << "You received two of the same card. Would you like to"
                    << " split? 0 for no, 1 for yes: ";
                std::cin >> useroption;
                if(useroption)
                {
                    splitflag = true;
                    try
                    {
                        player.setBet();
                    }
                    catch(Player::InsufficientBalance &e)
                    {
                        std::cerr << e.getErrorMessage() << "\n";
                    }
                    player.split();
                }
            }
        }
```

```
131
132             std::cout << "************************************************\n";
133             // hand 1
134             std::cout << "Now playing: hand 1.\n";
135             std::cout << player.outputPrettyWrapper(0) << "\n";
136             std::cout << "Value: " << player.getValue(0) << "\n";
137             while(true)
138             {
139                 if(player.getValue(0) >= 21)
140                     break;
141
142                 std::cout << "Enter 0 for hit, 1 for stand: ";
143                 std::cin >> useroption;
144                 if(useroption == 0) // hit
145                 {
146                     player.drawCard(0, deck, 1);
147                     std::cout << "Value: " << player.getValue(0) << "\n";
148                 }
149                 else // stand
150                     break;
151             }
152
153             // hand 2
154             if(splitflag)
155             {
156                 std::cout << "Now playing: hand 2.\n";
157                 std::cout << player.outputPrettyWrapper(1) << "\n";
158                 std::cout << "Value: " << player.getValue(1) << "\n";
159                 while(true)
160                 {
161                     if(player.getValue(1) >= 21)
162                         break;
163
164                     std::cout << "Enter 0 for hit, 1 for stand: ";
165                     std::cin >> useroption;
166                     if(useroption == 0) // hit
167                     {
168                         player.drawCard(1, deck, 1);
169                         std::cout << "Value: " << player.getValue(1) << "\n";
170                     }
171                     else // stand
172                         break;
173                 }
174             }
175
176             hand1val = player.getValue(0);
177             hand2val = player.getValue(1);
178             dealval  = dealer.getValue(0);
179
180             std::cout << "Dealer's hand:\n";
181             std::cout << dealer.outputPrettyWrapper(0) << "\n";
182             std::cout << "Value: " << dealval << "\n";
183
184             std::cout << "Results for Game 1:\n";
185             if(hand1val > 21)
186             {
187                 std::cout << "Bust! You lose.\n";
188                 player.updateAccount(0,userBet);
189                 std::cout << "************************************************\n";
190                 player.showAccount();
191             }
192             else if(hand1val > dealval)
193             {
194                 std::cout << "You win!\n";
195                 player.updateAccount(userBet*2,0);
```

```
            std::cout << "************************************************\n";
            player.showAccount();
        }
        else if(hand1val < dealval)
        {
            std::cout << "You lose.\n";
            player.updateAccount(0,userBet);
            std::cout << "************************************************\n";
            player.showAccount();
        }
        else // tie
        {
            std::cout << "It's a tie!\n";
            player.updateAccount(0,userBet/2);
            std::cout << "************************************************\n";
            player.showAccount();
        }
        if(splitflag)
        {
            std::cout << "************************************************\n";
            std::cout << "Results for Game 2:\n";
            if(hand2val > 21)
            {
                std::cout << "Bust! You lose.\n";
                player.updateAccount(0,userBet);
                std::cout << "************************************************\n";
                player.showAccount();
            }
            else if(hand2val > dealval)
            {
                std::cout << "You win!\n";
                player.updateAccount(userBet*2,0);
                std::cout << "************************************************\n";
                player.showAccount();
            }
            else if(hand2val < dealval)
            {
                std::cout << "You lose.\n";
                player.updateAccount(0,userBet);
                std::cout << "************************************************\n";
                player.showAccount();
            }
            else // tie
            {
                std::cout << "It's a tie!\n";
                player.updateAccount(0,userBet/2);
                std::cout << "************************************************\n";
                player.showAccount();
            }
        }
        player.returnCards(deck,0);
        player.returnCards(deck,1);
        dealer.returnCards(deck,0);
        std::cout << "************************************************\n";
        std::cout << "Enter 0 to quit or 1 to continue: ";
        std::cin >> useroption;
    } while(useroption);

    return 0;
}
```

```
/**
 * CardList class header file.
 * CECS 275 - Fall 2021
 * @author Rodrigo Becerril Ferreyra
 * @author Ethan Hua
 * @version 1
 */

#ifndef CARDLIST_H
#define CARDLIST_H

#include <string>
#include <ostream>

/**
 * CardList is a singly-linked list class that can store all of the cards in a
 * standard French 52-card deck. It implements several functions including
 * shuffle, which randomizes the order of the deck.
 */
class CardList
{
    public:
        enum listOfSuits {SPADES, CLUBS, DIAMONDS, HEARTS};

    private:
        struct Card
        {
            listOfSuits suit;
            // note that 1 is A, 11 is J, 12 is Q, and 13 is K
            unsigned int value;
            Card* nextCard;
        };
        Card* headPtr;

        unsigned int numCards;

        /**
         * Adds an existing card to the front of the list. This function does
         * not create a new card. THIS FUNCTION DOES NOT CHECK FOR THE EXISTENCE
         * OF A Card OBJECT THAT IS POINTED TO BY cardPtr!!!
         * This is very important, because many things can go
         * wrong if a random pointer is put into this function.
         * However, if cardPtr is nullptr, nothing is put onto the list.
         * @param cardPtr The pointer to the Card object to add to the list.
         */
        void addCard(Card* cardPtr);

        /**
         * Removes the Card at the top of the list and returns a pointer to it.
         * Note that this function does not destroy the Card nor free the memory
         * taken by it.
         * @return The pointer to the first Card in the list. If the list is
         * empty, return nullptr.
         */
        Card* pop();

    public:
        /**
         * The constructor simply initializes an empty list.
         */
        CardList() {headPtr = nullptr; numCards = 0;}

        /**
         * The copy constructor traverses the list of the original object and
         * creates new nodes that are copies of the original object's nodes.
```

```
66          * This is to avoid having two objects pointing to the same list.
67          * @param copyFromMe The object to be copied from.
68          */
69         CardList(const CardList &copyFromMe);
70
71         /**
72          * Assignment operator overload. Deletes all Cards in the source list
73          * and creates new ones that are copies of copyFromMe.
74          * @param copyFromMe The object to copy from.
75          * @return A reference to itself.
76          */
77         CardList& operator= (const CardList &copyFromMe);
78
79         /**
80          * The destructor deletes all cards in the list and frees all memory.
81          */
82         ~CardList();
83
84         /**
85          * Deletes all cards that are inside the list.
86          */
87         void deleteAllCards();
88
89         /**
90          * Creates a card and adds it to the front of the list. Checks if an
91          * identical card exists; if so, no card is created.
92          * @param cardSuit The suit of the card (CLUBS, DIAMONDS, HEARTS, or
93          * SPADES).
94          * @param cardValue The value of the card (between 1 and 13 inclusive).
95          * @return True if a card was created, false otherwise. Note that if
96          * either of the parameters are out of range.
97          */
98         bool createCard(enum listOfSuits cardSuit, unsigned int cardValue);
99
100        /**
101         * Transfers cards in the card list to another card list. This
102         * function simply removes the desired amount of cards in the list that
103         * calls this function and transfers them to other. No new cards
104         * are created.
105         * @param other The list to transfer all cards to.
106         * @param amount The number of cards to transfer.
107         * @return True if the transfer was successful, false if it was not
108         * successful (for example, if there were no cards to transfer to
109         * begin with).
110         */
111        bool transferTo(CardList &other, unsigned int amount);
112
113        /**
114         * Transfers all the cards in the list to the CardList other.
115         * @param other The CardList to transfer all cards to.
116         * @return True if the transfer was successful, false otherwise.
117         */
118        bool transferTo(CardList &other) {return transferTo(other, numCards);}
119
120        /**
121         * Shuffles the list in place.
122         */
123        void shuffle();
124
125        /**
126         * Traverses through the list and counts the number of cards that match
127         * the parameter given.
128         * @param suitToFind The suit that the function will look for.
129         * @return The number of cards that match the description.
130         */
```

```
131         unsigned int countCards(listOfSuits suitToFind) const;
132
133         /**
134          * Traverses through the list and counts the number of cards that match
135          * the parameter given.
136          * @param valueToFind The value that the function will look for.
137          * @return The number of cards that match the description.
138          */
139         unsigned int countCards(unsigned int valueToFind) const;
140
141         /**
142          * Calculates the total value of all the cards in the list.
143          * @return The sum of the values of the cards in the list.
144          */
145         unsigned int listValue() const;
146
147         /**
148          * Creates and returns a string that has the basic information of
149          * all Cards in the List.
150          * @return A formatted std::string.
151          */
152         std::string outputBasic() const;
153
154         /**
155          * Creates and returns a string that is nicely formatted to show the
156          * cards in the list. Each card will be 7x5 characters in size. Only
157          * 10 cards will be able to be printed onto one line; any further cards
158          * will be printed on more lines. Note that if the amount parameter
159          * is not greater than 0 or the list is empty, the function will return
160          * an empty string.
161          * @param amount The amount of cards to print out.
162          * @return A formatted std::string.
163          */
164         std::string outputPretty(unsigned int amount) const;
165
166         /**
167          * This overloaded function prints out all cards in the list nicely
168          * according to the specifications of
169          * CardList::outputPretty(unsigned int).
170          * @return A formatted std::string/
171          */
172         std::string outputPretty() const {return outputPretty(numCards);}
173
174         /**
175          * Outputs the first card in the list along with a blank card. This
176          * function can be used in order to begin a game of Blackjack, which
177          * displays one of the dealer's cards and hides the second.
178          * @return A formatted std::string. If the list holds no cards,
179          * the function returns an empty string.
180          */
181         std::string outputBlackjack() const;
182
183         /**
184          * Stream extraction operator overload that calls CardList::outputBasic
185          * and displays the same information.
186          * @return The same std::ostream operator that was used to call
187          * the function.
188          */
189         friend std::ostream& operator<<(std::ostream& os, const CardList &list);
190
191         /**
192          * Getter function for numCards.
193          * @return The length of the list; the amount of cards currently
194          * held by the list.
195          */
```

```
196         unsigned int size() const {return numCards;}
197
198         /**
199          * Compares the values (not the suits) of the first two cards.
200          * @return True if the value of the first two cards are the same,
201          * false if they are different (or there are less than two cards).
202          */
203         bool compareFirstTwoCards();
204 };
205
206 #endif//CARDLIST_H
207
```

```cpp
/**
 * CardList class implementation file.
 * CECS 275 - Fall 2021
 * @author Rodrigo Becerril Ferreyra
 * @author Ethan Hua
 * @version 1
 */

#include <algorithm> // for std::shuffle
#include <random>    // for std::default_random_engine
#include <chrono>    // for std::chrono
#include <string>    // for std::string and std::to_string
#include <ostream>   // for std::ostream
#include "CardList.h"

// ***************************************************************************
// node functions
// ***************************************************************************

CardList::Card* CardList::pop()
{
    // do nothing if list is empty
    if(!headPtr)
        return nullptr;

    // else pop the head and return it
    Card* oldhead = headPtr;
    headPtr = headPtr->nextCard;
    --numCards;
    return oldhead;
}

void CardList::addCard(Card* cardPtr)
{
    cardPtr->nextCard = headPtr;
    headPtr = cardPtr;
    ++numCards;
}

bool CardList::transferTo(CardList &other, unsigned int amount)
{
    if(!headPtr)
        return false;

    Card* cardBeingTransferred = nullptr;

    // do not remove more cards than there are in the CardList
    if(amount > numCards) amount = numCards;

    // this loop transfers the number of cards requested
    for(int i = 0; i < amount; ++i)
    {
        cardBeingTransferred = pop();
        if(!cardBeingTransferred) // if the list is empty for any reason
            return false; // this prevents nullptr being added to other
        other.addCard(cardBeingTransferred);
    }

    return true;
}

bool CardList::createCard(CardList::listOfSuits cardSuit,unsigned int cardValue)
{
    // the function does not do anything if the value is out of range
    if(cardValue > 13 || cardValue < 1)
```

```
66          return false;
67
68      // next, look through the list and see if any cards match requested card
69      Card* traversePtr = headPtr;
70      while(traversePtr) // iterate until end of list
71      {
72          // if there is a match, return false
73          if(traversePtr->suit == cardSuit && traversePtr->value == cardValue)
74              return false;
75          traversePtr = traversePtr->nextCard;
76      }
77
78      // if there is no match, create a new card and add it
79      traversePtr = new Card;
80      traversePtr->suit = cardSuit;
81      traversePtr->value = cardValue;
82      addCard(traversePtr);
83
84      return true;
85  }
86
87  unsigned int CardList::countCards(CardList::listOfSuits suitToFind) const
88  {
89      unsigned int count = 0;
90      Card* traversePtr = headPtr;
91
92      // loop until traversePtr == nullptr
93      while(traversePtr)
94      {
95          if(traversePtr->suit == suitToFind)
96              ++count;
97          traversePtr = traversePtr->nextCard;
98      }
99
100     return count;
101 }
102
103 unsigned int CardList::countCards(unsigned int valueToFind) const
104 {
105     unsigned int count = 0;
106     Card* traversePtr = headPtr;
107
108     // loop until traversePtr == nullptr
109     while(traversePtr)
110     {
111         if(traversePtr->value == valueToFind)
112             ++count;
113         traversePtr = traversePtr->nextCard;
114     }
115
116     return count;
117 }
118
119 unsigned int CardList::listValue() const
120 {
121     unsigned int total = 0;
122     Card* traversePtr = headPtr;
123
124     // loop until traversePtr == nullptr
125     while(traversePtr)
126     {
127         // Truncate Jack/Queen/King
128         if (traversePtr->value > 10)
129             total += 10;
130         else
```

```
131             total += traversePtr->value;
132
133         traversePtr = traversePtr->nextCard;
134     }
135
136     return total;
137 }
138
139 bool CardList::compareFirstTwoCards()
140 {
141     if(numCards < 2) return false;
142
143     if(headPtr->value == (headPtr->nextCard)->value)
144         return true;
145     return false;
146 }
147
148 CardList::CardList(const CardList &copyFromMe)
149 {
150     headPtr = nullptr; numCards = 0;
151     *this = copyFromMe;
152 }
153
154 CardList& CardList::operator= (const CardList &copyFromMe)
155 {
156     deleteAllCards();
157     const Card* otherPtr = copyFromMe.headPtr;
158     Card* currPtr = nullptr, *prevPtr = nullptr;
159
160     // while otherPtr != nullptr
161     // Note that this is skipped if the other list is empty.
162     while(otherPtr)
163     {
164         // create new card and copy values
165         currPtr = new Card; ++numCards;
166         currPtr->value = otherPtr->value;
167         currPtr->suit = otherPtr->suit;
168         currPtr->nextCard = nullptr;
169
170         // if headPtr == nullptr then have currPtr be the start of the list
171         if(!headPtr)
172             headPtr = currPtr;
173         // if prevPtr != null then link the previous card to the new card
174         if(prevPtr)
175             prevPtr->nextCard = currPtr;
176
177         // make sure we can link this card to the next card on the
178         // next iteration
179         prevPtr = currPtr;
180
181         // move to next card in other list
182         otherPtr = otherPtr->nextCard;
183     }
184
185     return *this;
186 }
187
188 CardList::~CardList()
189 {
190     deleteAllCards();
191 }
192
193 void CardList::deleteAllCards()
194 {
195     Card* currPtr = headPtr;
```

```
196       Card* nextPtr = nullptr;
197
198       // loop until currPtr is at the end of the list
199       while(currPtr)
200       {
201           nextPtr = currPtr->nextCard;
202           delete currPtr; --numCards;
203           currPtr = nextPtr;
204       }
205       headPtr = nullptr;
206   }
207
208   // ************************************************************************
209   // display functions
210   // ************************************************************************
211
212   std::string CardList::outputBasic() const
213   {
214       std::string outstring = "VALUE | SUIT\n------|---------\n";
215
216       // traverse through the list and add data to the string
217       Card* traversePtr = headPtr;
218       while(traversePtr) // while not nullptr
219       {
220           // pick the value
221           switch(traversePtr->value)
222           {
223               case 1: // ace
224                   outstring += "A     ";
225                   break;
226               case 11: // j
227                   outstring += "J     ";
228                   break;
229               case 12: // q
230                   outstring += "Q     ";
231                   break;
232               case 13: // k
233                   outstring += "K     ";
234                   break;
235               case 10:
236                   outstring += "10    ";
237                   break;
238               default: // anything else in between
239                   outstring += std::to_string(traversePtr->value) + "     ";
240                   break;
241           }
242           outstring += " | ";
243
244           // pick the suit
245           switch(traversePtr->suit)
246           {
247               case CardList::CLUBS:
248                   outstring += "CLUBS";
249                   break;
250               case CardList::DIAMONDS:
251                   outstring += "DIAMONDS";
252                   break;
253               case CardList::HEARTS:
254                   outstring += "HEARTS";
255                   break;
256               case CardList::SPADES:
257                   outstring += "SPADES";
258           }
259
260           outstring += "\n";
```

```
261             traversePtr = traversePtr->nextCard;
262         }
263 
264         return outstring;
265     }
266 
267     std::string CardList::outputPretty(unsigned int amount) const
268     {
269         // The way this works is that the function must print one line at a time.
270         // Each piece is one line. The function makes sure to only print 10
271         // cards at a time, then moves to the next line of cards.
272 
273         // do not print more cards than there are
274         if(amount > numCards) amount = numCards;
275 
276         int oldAmount, i;
277         std::string outstring;
278         Card* traversePtr = headPtr;
279         Card* oldPtr = nullptr;
280 
281         // loop until there are no more cards to print
282         // Note that this is skipped if there are no Cards in the list.
283         while(amount > 0)
284         {
285             oldAmount = amount;
286             oldPtr = traversePtr;
287             // print the top lines
288             // run 10 times (the amount of cards that fit on one line) or
289             // until the amount of cards is reached
290             for(i = 0; i < 10 && amount > 0; ++i)
291             {
292                 outstring += "------- ";
293                 --amount;
294             }
295             outstring += "\n";
296 
297             // print the first line with the value
298             amount = oldAmount;
299             for(i = 0; i < 10 && amount > 0; ++i)
300             {
301                 outstring += "|";
302                 switch(traversePtr->value)
303                 {
304                     case 13: // K
305                         outstring += "K    ";
306                         break;
307                     case 12: // Q
308                         outstring += "Q    ";
309                         break;
310                     case 11: // J
311                         outstring += "J    ";
312                         break;
313                     case 10: // 10 needs a special case because it's 2 digits long
314                         outstring += "10   ";
315                         break;
316                     case 1: // A
317                         outstring += "A    ";
318                         break;
319                     default:
320                         outstring += std::to_string(traversePtr->value) + "    ";
321                 }
322                 outstring += "| ";
323                 --amount;
324                 traversePtr = traversePtr->nextCard;
325             }
```

```
326         outstring += "\n";
327 
328         // empty line
329         amount = oldAmount;
330         for(i = 0; i < 10 && amount > 0; ++i)
331         {
332             outstring += "|     | ";
333             --amount;
334         }
335         outstring += "\n";
336 
337         // print suit info
338         amount = oldAmount;
339         traversePtr = oldPtr;
340         for(i = 0; i < 10 && amount > 0; ++i)
341         {
342             outstring += "|";
343 
344             switch(traversePtr->suit)
345             {
346                 case CardList::CLUBS:
347                     outstring += "CLUBS";
348                     break;
349                 case CardList::DIAMONDS:
350                     outstring += "DIAMD";
351                     break;
352                 case CardList::HEARTS:
353                     outstring += "HEART";
354                     break;
355                 case CardList::SPADES:
356                     outstring += "SPADE";
357                     break;
358             }
359 
360             traversePtr = traversePtr->nextCard;
361             --amount;
362             outstring += "| ";
363         }
364         outstring += "\n";
365 
366         // empty line
367         amount = oldAmount;
368         for(i = 0; i < 10 && amount > 0; ++i)
369         {
370             outstring += "|     | ";
371             --amount;
372         }
373         outstring += "\n";
374 
375         // print value info again
376         amount = oldAmount;
377         traversePtr = oldPtr;
378         for(i = 0; i < 10 && amount > 0; ++i)
379         {
380             outstring += "|";
381 
382             switch(traversePtr->value)
383             {
384                 case 13: // K
385                     outstring += "    K";
386                     break;
387                 case 12: // Q
388                     outstring += "    Q";
389                     break;
390                 case 11: // J
```

```
391                     outstring += "    J";
392                     break;
393                 case 10: // 10 needs a special case because it's 2 digits long
394                     outstring += "   10";
395                     break;
396                 case 1: // A
397                     outstring += "    A";
398                     break;
399                 default:
400                     outstring += "    " + std::to_string(traversePtr->value);
401                     break;
402             }
403 
404             traversePtr = traversePtr->nextCard;
405             --amount;
406             outstring += "| ";
407         }
408         outstring += "\n";
409 
410         amount = oldAmount;
411         for(i = 0; i < 10 && amount > 0; ++i)
412         {
413             outstring += "------- ";
414             --amount;
415         }
416         outstring += "\n";
417     }
418 
419     return outstring;
420 }
421 
422 std::string CardList::outputBlackjack() const
423 {
424     std::string outstring;
425     // if the list is not empty
426     if(headPtr)
427     {
428         // print top of card
429         outstring += "------- -------\n";
430 
431         // print value
432         switch(headPtr->value)
433         {
434             case 13: // K
435                 outstring += "|K    | ";
436                 break;
437             case 12: // Q
438                 outstring += "|Q    | ";
439                 break;
440             case 11: // J
441                 outstring += "|J    | ";
442                 break;
443             case 10: // 10 has two digits so it needs a special case
444                 outstring += "|10   | ";
445                 break;
446             case 1: // A
447                 outstring += "|A    | ";
448                 break;
449             default:
450                 outstring += "|" + std::to_string(headPtr->value) +"    | ";
451                 break;
452         }
453         outstring += "|     |\n";
454 
455         // print empty line
```

```
456             outstring += "|     | |     |\n";
457
458             // print suit
459             switch(headPtr->suit)
460             {
461                 case CardList::CLUBS:
462                     outstring += "|CLUBS| ";
463                     break;
464                 case CardList::DIAMONDS:
465                     outstring += "|DIAMD| ";
466                     break;
467                 case CardList::HEARTS:
468                     outstring += "|HEART| ";
469                     break;
470                 case CardList::SPADES:
471                     outstring += "|SPADE| ";
472                     break;
473             }
474             outstring += "|     |\n";
475
476             // print empty line
477             outstring += "|     | |     |\n";
478
479             // print value
480             switch(headPtr->value)
481             {
482                 case 13: // K
483                     outstring += "|    K| ";
484                     break;
485                 case 12: // Q
486                     outstring += "|    Q| ";
487                     break;
488                 case 11: // J
489                     outstring += "|    J| ";
490                     break;
491                 case 10: // 10 has two digits so it needs a special case
492                     outstring += "|   10| ";
493                     break;
494                 case 1: // A
495                     outstring += "|    A| ";
496                     break;
497                 default:
498                     outstring += "|    " + std::to_string(headPtr->value) +"| ";
499                     break;
500             }
501             outstring += "|     |\n";
502
503             // print bottom of card
504             outstring += "------- -------\n";
505         }
506
507         return outstring;
508     }
509
510     std::ostream& operator<< (std::ostream& os, const CardList &list)
511     {
512         os << list.outputBasic();
513         return os;
514     }
515
516     // ************************************************************************
517     // misc functions
518     // ************************************************************************
519
520     void CardList::shuffle()
```

```
{
    // if list is empty, do nothing
    if(!headPtr)
        return;

    // create an array of all the addresses of all nodes in the list
    int i = 0;
    Card** addressArray = new Card*[numCards];

    // start at head; loop until you reach the end of the list; go to next card
    for(Card* nodePtr = headPtr; nodePtr; nodePtr = nodePtr->nextCard)
        addressArray[i++] = nodePtr;
        // note that the array avoids putting nullptr into the array

    // shuffle the array of addresses
    // shuffle implementation taken from https://stackoverflow.com/a/26682712
    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
    std::default_random_engine rng(seed);
    std::shuffle(&addressArray[0], &addressArray[numCards], rng);
    // std::shuffle takes the address of the first element and the address of
    //   the element after the last element. The last element is
    //   addressArray[numCards-1], so the element after it is
    //   addressArray[numCards].

    // re-link all nodes according to the new order of addressArray
    // start from the back and work your way up to the front
    addressArray[numCards-1]->nextCard = nullptr;
    for(i = numCards - 2; i >= 0; --i)
        addressArray[i]->nextCard = addressArray[i+1];
    headPtr = addressArray[0];

    // free array memory
    delete[] addressArray;
}

```

```cpp
/**
 * Account class header file.
 * CECS 275 - Fall 2021
 * @author Rodrigo Becerril Ferreyra
 * @author Ethan Hua
 * @version 1
 */

#ifndef ACCOUNT_H
#define ACCOUNT_H

#include <exception>
#include <fstream>
#include <ostream>

/**
 * The Account class handles subjects such as account number and records
 * such as total money and money lost. The Account class also handles saving
 * all these values in a file.
 */
class Account
{
    public:
// ***************************************************************************
// Exception handling classes
// ***************************************************************************
        /**
         * Class for exception handling of account number in constructor.
         */
        class NumOutOfBounds : public std::exception
        {
            public:
                /**
                 * Initializes the class to hold the expected lower and upper
                 * bounds of the value.
                 * @param errval The value that caused the error.
                 * @param lower The lower bound of the expected value
                 * (inclusive).
                 * @param upper The upper bound of the expected value
                 * (inclusive).
                 */
                NumOutOfBounds(int errval, int lower, int upper);

                int getLowerBounds() const {return lowerbounds;}
                int getUpperBounds() const {return upperbounds;}

                /**
                 * Builds an error message and returns it.
                 * @return A detailed error message.
                 */
                std::string getErrorMessage();
            private:
                int errorvalue;
                int lowerbounds;
                int upperbounds;
        };

        /**
         * Class for exception handling of balance in constructor.
         */
        class NumNegative : public std::exception{};

        /**
         * Class for exception handling of file not existing.
         */
```

```
66          class FileNotFoundError : public std::exception
67          {
68              public:
69                  /**
70                   * Sets the name of the file.
71                   * @param name The filename.
72                   */
73                  FileNotFoundError(std::string name);
74
75                  /**
76                   * Builds and returns an error message.
77                   * @return An error message.
78                   */
79                  std::string getErrorMessage();
80              private:
81                  std::string filename;
82          };
83
84  // ************************************************************************
85  // constructors
86  // ************************************************************************
87          /**
88           * Default constructor.
89           */
90          Account();
91
92          /**
93           * This overloaded constructor allows the user to set the value for
94           * one of the class's members. Uses a known Account Number to
95           * attempt to load value from a .txt file.
96           * @param accountNumber The account number of the new account.
97           * @throws ParameterOutOfBounds if the account number is not a
98           * positive eight-digit number.
99           */
100         Account(int accountNumber);
101
102         /**
103          * This overloaded constructor allows the user to set the value for
104          * two of the class's members.
105          * @param accountNumber The account number of the new account.
106          * @param balance The new account's balance.
107          * @throws ParameterOutOfBounds if the account number is not a
108          * positive eight-digit number.
109          * @throws ParameterOutOfBounds if the balance is
110          */
111         Account(int accountNumber, int balance);
112
113 // ************************************************************************
114 // destructors
115 // ************************************************************************
116
117         /**
118          * Automatically saves all data when object is destroyed.
119          */
120         ~Account();
121
122 // ************************************************************************
123 // load and save functions
124 // ************************************************************************
125
126         /**
127          * Reads the account data from the file acc_{accountNumber}.txt.
128          * @throws FileNotFoundError if the file does not exist.
129          */
130         void load();
```

```
131
132         /**
133          * Writes the account data to the file acc_{accountNumber}.txt.
134          * Overwrites the file if it exists, and creates a new file if it
135          * does not.
136          */
137         void save();
138
139 // ********************************************************************************
140 // other functions
141 // ********************************************************************************
142
143         /**
144          * Stream extraction operator overload. Extracts the contents of the
145          * object (in the same format as the file) and feeds it into the
146          * std::ostream operator used to call it. Can be chained.
147          */
148         friend std::ostream& operator<< (std::ostream& os, const Account& acc);
149
150 // ********************************************************************************
151 // setter and getter functions
152 // ********************************************************************************
153
154         // setter functions
155         /**
156          * Use this function to input the results of a game of blackjack.
157          * @param moneyWon The amount of money won from the game.
158          * @param moneyLost The amount of money lost from the game.
159          */
160         void inputGameResults(double moneyWon, double moneyLost);
161
162         /**
163          * Attempts to set accountNumber independently of other members.
164          * Use to update empty Account to attempt loading file.
165          * @param accSet  Attempted overwrite of accountNumber.
166          * @throws ParameterOutOfBounds if the account number is not a
167          * positive eight-digit number.
168          */
169         void setAccountNumber(unsigned int accSet);
170
171         // getter functions
172         /**
173          * Getter function for accountNumber.
174          * @return accountNumber
175          */
176         int getAccountNumber() const {return accountNumber;}
177
178         /**
179          * Getter function for gamesPlayed.
180          * @return gamesPlayed
181          */
182         int getGamesPlayed() const {return gamesPlayed;}
183
184         /**
185          * Getter function for balance.
186          * @return balance
187          */
188         double getBalance() const {return balance;}
189
190         /**
191          * Getter function for amountWon.
192          * @return amountWon
193          */
194         double getAmountWon() const {return amountWon;}
195
```

```
196         /**
197          * Getter function for amountLost.
198          * @return amountLost
199          */
200         double getAmountLost() const {return amountLost;}
201 
202     private:
203         int accountNumber;
204         int gamesPlayed;
205         double balance;
206         double amountWon;
207         double amountLost;
208 
209         /**
210          * Gets the filename of the filename according to the account number.
211          * @return The filename of the file where the account is saved.
212          */
213         std::string getFilename();
214 };
215 
216 #endif//ACCOUNT_H
217 
```

```cpp
/**
 * Account class implementation file.
 * CECS 275 - Fall 2021
 * @author Rodrigo Becerril Ferreyra
 * @author Ethan Hua
 * @version 1
 */

#include <string>
#include <iomanip>
#include <fstream>
#include <ostream>
#include "Account.h"

Account::NumOutOfBounds::NumOutOfBounds(int errval, int lower, int upper)
{
    errorvalue = errval;
    lowerbounds = lower;
    upperbounds = upper;
}

std::string Account::NumOutOfBounds::getErrorMessage()
{
    return "NumOutOfBounds; Value received: " + std::to_string(errorvalue)
      + "; expected value between " + std::to_string(lowerbounds)
      + " and " + std::to_string(upperbounds) + " (inclusive).";
}

Account::FileNotFoundError::FileNotFoundError(std::string name)
{
    filename = name;
}

std::string Account::FileNotFoundError::getErrorMessage()
{
    return "FileNotFoundError; The file " + filename + " could not be found.";
}

Account::Account()
{
    this->accountNumber = 0;
    this->balance = 0;
    this->gamesPlayed = 0;
    this->amountWon = 0;
    this->amountLost = 0;
}

Account::Account(int accountNumber)
{
    // account number must be a positive eight-digit number
    if(accountNumber < 1 || accountNumber > 99999999)
        throw NumOutOfBounds(accountNumber, 1, 99999999);
    this->accountNumber = accountNumber;
    load();
}

Account::Account(int accountNumber, int balance)
{
    // account number must be a positive eight-digit number
    if(accountNumber < 1 || accountNumber > 99999999)
        throw NumOutOfBounds(accountNumber, 1, 99999999);

    // balance must be non-negative
    if(balance < 0)
        throw NumNegative();
```

```
66
67      this->accountNumber = accountNumber;
68      this->balance = balance;
69      this->gamesPlayed = 0;
70      this->amountWon = 0;
71      this->amountLost = 0;
72      save();
73  }
74
75  void Account::load()
76  {
77      // open file
78      std::fstream infile(getFilename().c_str(), std::ios::in);
79      if(infile.fail())
80          throw FileNotFoundError(getFilename());
81
82      // load all values into variables
83      std::string lines[5] = {};
84
85      for(int i = 0; i < 5; ++i)
86          std::getline(infile, lines[i]);
87
88      // cast string into appropriate type
89      accountNumber = std::stoi(lines[0].substr(16));
90      balance       = std::stod(lines[1].substr(18));
91      gamesPlayed   = std::stoi(lines[2].substr(13));
92      amountWon     = std::stod(lines[3].substr(19));
93      amountLost    = std::stod(lines[4].substr(20));
94
95      infile.close();
96  }
97
98  void Account::save()
99  {
100     // open file
101     std::fstream outfile(getFilename().c_str(), std::ios::out);
102
103     // populate file
104     outfile << *this << "\n";
105
106     outfile.close();
107 }
108
109 void Account::inputGameResults(double moneyWon, double moneyLost)
110 {
111     ++gamesPlayed;
112     amountWon += moneyWon;
113     amountLost += moneyLost;
114     balance += (moneyWon - moneyLost);
115     save();
116 }
117
118 void Account::setAccountNumber(unsigned int accSet)
119 {
120     if(accSet < 1 || accSet > 99999999)
121         throw NumOutOfBounds(accSet, 1, 99999999);
122     accountNumber = accSet;
123 }
124
125 Account::~Account()
126 {
127     save();
128 }
129
130 std::string Account::getFilename()
```

```
{
    // 8-digit account
    int maxNum = 8;
    // Take the current accountNumber's length after casting to string.
    std::string tempAcc = std::to_string(accountNumber);
    int str_length = tempAcc.length();
    // Append leading zeroes until 8 digits
    for(int i = 0; i < maxNum - str_length; i++)
    {
        tempAcc = "0" + tempAcc;
    }
    // Generate file name
    std::string filename = "acc_";
    filename += tempAcc;
    filename += ".txt";
    return filename;
}

std::ostream& operator<< (std::ostream& os, const Account& acc)
{
    os << "Account Number: " << std::setfill('0') << std::setw(8)
       << acc.accountNumber << "\n";
    os << "Current Balance: $" << std::fixed << std::setprecision(2)
       << acc.balance << "\n";
    os << "Game Played: " << acc.gamesPlayed << "\n";
    os << "Total Amount Won: $" << std::fixed << std::setprecision(2)
       << acc.amountWon << "\n";
    os << "Total Amount Lost: $" << std::fixed << std::setprecision(2)
       << acc.amountLost;
    return os;
}

```

```cpp
/**
 * Player class header file.
 * CECS 275 - Fall 2021
 * @author Rodrigo Becerril Ferreyra
 * @author Ethan Hua
 * @version 1
 */

#ifndef PLAYER_H
#define PLAYER_H

#include <iostream>
#include <string>
#include "CardList.h"
#include "Account.h"
#include <iostream>

/**
 * The Player class is responsible for all operations having to do with a Player
 * in the game of Blackjack: holding one (or two) hands with different values,
 * having an Account, etc.
 */
class Player
{
    public:
// ***************************************************************************
// Exception handling classes
// ***************************************************************************
        /**
         * Class for exception handling of betting more than the playerRef
         * Account's balance.
         */
        class InsufficientBalance : public std::exception
        {
            public:
                InsufficientBalance(double userBet, double accBal);
                /**
                 * Builds an error message and returns it.
                 * @return A detailed error message.
                 */
                std::string getErrorMessage();
            private:
                double balance;
                double bet;
        };
        /**
         * Class for exception handling of invalid action.
         */
        class NoBet : public std::exception
        {
            public:
                NoBet(double userBet);
                /**
                 * Builds an error message and returns it.
                 * @return A detailed error message.
                 */
                std::string getErrorMessage();
            private:
                double bet;
        };
// ***************************************************************************
// constructors
// ***************************************************************************
        /**
         * Initialization constructor to create a new Player. Defaults to a
```

```
66          * "dealer"-type Player, which disables the ability to save or
67          * load an Account.
68          */
69         Player();
70 
71         /**
72          * Constructor specific to a "player"-type Player. Similar
73          * implementation to default constructor but specifically
74          * activates ability to set up Account.
75          * @param ID  Positive integer to attempt accessing account.
76          * @throws FileNotFound if no text file but valid account.
77          * @throws NumOutOfBounds if player ID not 8-digit positive integer.
78          */
79         Player(int ID);
80 
81 // ***************************************************************************
82 // destructors
83 // ***************************************************************************
84         /**
85          * Deletes dynamically allocated Account pointer.
86          */
87         ~Player();
88 // ***************************************************************************
89 // setter and getter functions
90 // ***************************************************************************
91         /**
92          * Overloaded function acepts a user's bet to begin the game.
93          * @param newBet  Desired sum to bet for new game.
94          * @throws InsufficientBalance if user bets more than account holds.
95          */
96         void setBet(double newBet);
97         /**
98          * Overloaded function copies the value of bet1 over to bet2.
99          * Used when splitting.
100         * @throws InsufficientBalance if user bets more than account holds.
101         */
102        void setBet();
103 
104        /**
105         * Account wrapper function that takes the user's earnings
106         * and updates their associated Account parameters.
107         * @param win  How much money was won in the game.
108         * @param loss How much money was lost in the game.
109         */
110        void updateAccount(double win, double loss)
111            { playerRef->inputGameResults(win, loss); }
112 
113        /**
114         * Getter function for hand/value.
115         * @param corrVal  The hand to retrieve the value for.
116         * @return         Value of the hand.
117         */
118        int getValue(int corrVal) const;
119 
120        /**
121         * Getter function for user's initial bet.
122         * Used to ensure that the user can split.
123         * @return     User's current bet.
124         */
125        double getBet() const
126            { return bet1; }
127 
128        /**
129         * Getter function for player's account.
130         * @return    Account parameters.
```

```
131          */
132         Account getAccount() const
133             { return *playerRef; }
134 
135         /**
136          * Outputs the Player object's associated Account.
137          */
138         void showAccount() const
139             { std::cout << *playerRef << "\n"; }
140 
141         /**
142          * Wrapper function to retrieve the user's current balance in
143          * their account to verify that they are making a legal bet.
144          * @return     The user's current balance of funds.
145          */
146         double checkMoney()
147             { return playerRef->getBalance(); }
148 // ***************************************************************************
149 // other functions
150 // ***************************************************************************
151         /**
152          * Gives the designated Player's hand the top cards from
153          * another CardList.
154          * @param hand  Player hand to draw cards into.
155          * @param deck  Base CardList to take cards from.
156          * @param count How many cards from the target CardList to take.
157          */
158         void drawCard(int hand, CardList &deck, int count);
159 
160         /**
161          * Returns the designated Player's hand to a larger pool.
162          * @param deck   Card pool to place Player's cards back into.
163          * @param hand   Player's hand to remove cards from.
164          */
165         void returnCards(CardList &deck, int hand);
166 
167         /**
168          * Wrapper for CardList::compareFirstTwoCards. Only works on hand1.
169          * @return True if the first two cards have the same value,
170          * false if they do not (or if there are less than two cards).
171          */
172         bool splitCondition()
173             {return hand1.compareFirstTwoCards();}
174 
175         /**
176          * Wrapper function to update the value of Player's hand.
177          * @param corrVal  Flag used to differentiate which value to update.
178          */
179         void updateVal(int corrVal);
180 
181         /**
182          * Wrapper for CardList::outputPretty()
183          * @param corrVal Flag used to differentiate between the CardLists.
184          * @return A formatted string.
185          */
186         std::string outputPrettyWrapper(int corrVal)
187         {
188             if(corrVal == 0)
189                 return hand1.outputPretty();
190             return hand2.outputPretty();
191         }
192 
193         /**
194          * Removes one card from hand1 and gives it to hand2.
195          * @return True if successful, false otherwise (for example
```

```
196          * if there are more or less than two cards).
197          */
198         bool split();
199 
200     private:
201         CardList hand1, hand2;    // the hands to hold the CardLists
202         int value1, value2;       // the values of each hand
203         int playerType;           // 0 for "Dealer" or 1 for "Player"
204         Account *playerRef;       // Defines Player's statistics
205         double bet1, bet2;        // Player's current bets
206 };
207 
208 #endif//PLAYER_H
209 
```

```cpp
/**
 * Player class implementation file.
 * CECS 275 - Fall 2021
 * @author Rodrigo Becerril Ferreyra
 * @author Ethan Hua
 * @version 1
 */

#include "CardList.h"
#include "Account.h"
#include "Player.h"
#include <iostream>
#include <string>

Player::InsufficientBalance::InsufficientBalance(double userBet, double accBal)
{
    bet = userBet;
    balance = accBal;
}

std::string Player::InsufficientBalance::getErrorMessage()
{
    return "InsufficientBalance; Player bet " + std::to_string(bet - balance)
    + " over maximum possible Account balance.\n";
}

Player::NoBet::NoBet(double userBet)
{
    bet = userBet;
}

std::string Player::NoBet::getErrorMessage()
{
    return "Cannot bet $" + std::to_string(bet) + "\n";
}

Player::Player()
{
    value1 = value2 = 0;
    playerType = 0;
    bet1 = 0;
    bet2 = 0;
    playerRef = nullptr;
}

Player::Player(int ID)
{
    playerRef = nullptr;
    value1 = value2 = 0;
    // If ID is a positive value, attempt to load/save account.
    if (ID > 0)
    {
        playerType = ID;
        // Attempt to load account using ID value.
        try
        {
            playerRef = new Account(ID);
        }
        catch(Account::FileNotFoundError &e)
        {
            // File did not already exist, so make a new one.
            std::cerr << e.getErrorMessage() << "\n";
            std::cout << "Generating new account with provided ID.\n";
            playerRef = new Account(ID,1000);
        }
```

```
66          catch(Account::NumOutOfBounds &e)
67          {
68              // File input is not valid.
69              std::cerr << e.getErrorMessage() << "\n";
70              std::cout << "Generating new account with truncated ID.\n";
71              playerRef = new Account(ID%99999999,1000);
72          }
73      }
74      else
75      {
76          // Default to playerType of dealer.
77          playerType = 0;
78      }
79      bet1 = 0;
80      bet2 = 0;
81  }
82
83  Player::~Player()
84  {
85    delete playerRef;
86    playerRef = nullptr;
87  }
88
89  void Player::setBet(double newBet)
90  {
91      double money = checkMoney();
92      if (money < newBet)
93          throw Player::InsufficientBalance(newBet,money);
94      else if (newBet == 0)
95          throw Player::NoBet(newBet);
96      else
97          bet1 = newBet;
98  }
99
100 int Player::getValue(int corrVal) const
101 {
102     if (corrVal == 0)
103         return value1;
104     else
105         return value2;
106 }
107
108 void Player::setBet()
109 {
110     // Verify once more that this is a possible bet.
111     double money = checkMoney();
112     if(money < bet1*2)
113         throw Player::InsufficientBalance(bet1,money);
114     else
115         bet2 = bet1;
116 }
117
118 void Player::drawCard(int hand, CardList &deck, int count)
119 {
120     // sets the hand to hand1 or hand2
121     CardList* chosenHand = (hand % 2 == 0) ? &hand1 : &hand2;
122
123     deck.transferTo(*chosenHand, count);
124     if (playerType)
125         std::cout << chosenHand->outputPretty();
126     else
127         std::cout << chosenHand->outputBlackjack();
128     updateVal(hand % 2);
129 }
130
```

```
131 void Player::returnCards(CardList &deck, int hand)
132 {
133     if (hand%2 == 0)
134         hand1.transferTo(deck);
135     else
136         hand2.transferTo(deck);
137 }
138
139 void Player::updateVal(int corrVal)
140 {
141     int numAces = 0;
142     // Input is 0, 2, 4, etc. Ideally the input is 0 for hand1.
143     if (corrVal % 2 == 0)
144     {
145         value1 = hand1.listValue();
146         // Special handling for Aces being both 1 and 11.
147         numAces = hand1.countCards(1);
148         if (numAces)
149         {
150             value1 += numAces*10;
151             // If having Ace = 11 exceeded 21, revert increase.
152             if (value1 > 21)
153             {
154                 do {
155                     value1 -= 10;
156                     numAces--;
157                 } while (numAces && value1 > 21);
158             }
159         }
160     } else {
161         value2 = hand2.listValue();
162         // Special handling for Aces being both 1 and 11.
163         numAces = hand2.countCards(1);
164         if (numAces)
165         {
166             value2 += numAces*10;
167             // If having Ace = 11 exceeded 21, revert increase.
168             if (value2 > 21)
169             {
170                 do {
171                     value2 -= 10;
172                     numAces--;
173                 } while (numAces && value2 > 21);
174             }
175         }
176     }
177 }
178
179 bool Player::split()
180 {
181     if(hand1.size() != 2)
182         return false;
183     hand1.transferTo(hand2, 1);
184     updateVal(0);
185     updateVal(1);
186     return true;
187 }
188
```