Hao Huynh

Professor Mark Lehr

CSC 17A – 48983

22 November 2015

Project 2: Gambling.

1. Introduction.

This Gambling project currently contains two games: Poker (version 2) and Blackjack.

* Poker (version 2): If in the first version of [Poker](Poker(version1).docx), the player can easily win a game when the final five cards meet one of the nine winning conditions, this second version of Poker makes the scenario of this game become more complex and difficult than the previous one. Indeed, it levels up the difficulty of a game by adding a dealer and scoring all cards on both hands of the player and the dealer. In addition, the details about Poker scoring rules will be explained in the description. Basically, the score of a player or a dealer is achieved by following the below order:
  + A Royal Flush (10000000 pts)
  + A Straight Flush (9000000 pts)
  + A Four of the Same Kind (8000000 pts)
  + A Full House (7000000 pts)
  + A Flush (6000000 pts)
  + A Straight (5000000 pts)
  + A Three of the Same Kind (4000000 pts)
  + Two Pairs (3000000 pts)
  + A Pair (2000000 pts)

Certainly, the player must has a higher score than the one of the dealer in order to win a game.

* Blackjack: is a game where the goal is to hit 21 through a combination of cards or have the dealer bust (over 21). Each number has its own face value while a Jack, Queen, or King has the value of 10. An ace can be represented as a 1 (hard) or an 11 (soft). At the beginning, the player is dealt two cards and the dealer is dealt two cards. Then, the player can select to hit (get another card) or stay (not get another card). The player is required to hit at least on a soft 17. However, the dealer must have a hard 17 or higher. If the dealer has a higher number than the player, the player loses their initial bet. Otherwise, the user wins.

I decided to implement this project because I could take advantage of the project 1. Moreover, based on the available, logical 52 card database, I added Blackjack to the project smoothly. Lastly, the Poker and Blackjack scoring rules are not so complicated that they could be developed in a reasonable amount of time.

1. Summary.
   1. The program statistics:
      1. The number lines of code (LOC) are about 1993 lines which are greater than 1000 lines in the requirement.
      2. There are 8 main variables.
      3. There are about 61 constants.
      4. There is 3 constructor and 2 destructor.
      5. There are about 50 methods.
   2. Miscellaneous: This program took me approximately 36 hours for coding and fixing errors. Moreover, I had learned about the Poker and Blackjack scoring techniques.
2. Description.
   1. The program problems:
      1. Manage 52 cards for both Poker and Blackjack.
      2. Apply the Poker and Blackjack scoring rules to check whether the player wins or not.
   2. The program solutions:
      1. Create the abstract base class AbstractCardTable being inherited by the two derived classes: PokerCardTable and BlackJackCardTable.
      2. Use two vectors to store all cards on both hands of the player and the dealer as well as create the method calculateScore().
   3. Sample Input/Output:
      1. The Gambling menu options: (The player choose an option by its index)



* + 1. The player enters a bet roll before playing the Poker:



* + 1. The player gets the first five Poker cards:



* + 1. The player chooses any number of the first five ones for replacing:



* + 1. The final result of Poker:



* + 1. The player enters a bet roll before playing the Blackjack:



* + 1. Both player and dealer get their first pairs of cards:



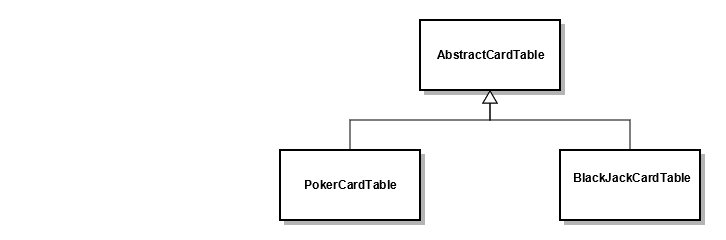
* + 1. The player chooses Stay:



* + 1. The player chooses Hit one times before Stay:



* 1. Flowchart and Pseudocode:
     1. Class design:



* + 1. BlackJackCardTable::[processDealerTurn](Flowcharts/processDealerTurn.jpg)().
    2. PokerCardTable::[replaceCards](Flowcharts/replacePokerCards.jpg)().
    3. BlackJackCardTable::calculateScore():
       1. Adding up all points of cards except for the Aces. The point of a card is the face value of that card. Especially, Jack, Queen, or King has the value of 10.
       2. Determine the value (1 or 11) of the Aces by comparing the current, accumulative sum with 21.
    4. PokerCardTable::calPokerScore():
       1. Sort cards by rank and by suit.
       2. Depends on the 9 winning conditions, calls the equivalent functions to get the score.
    5. PokerCardTable::getHandScore():

The total score of cards in a hand equals to

10^4\*the-first-highest-rank + 10^3\*the-second-highest-rank

+ 10^2\*the-third-highest-rank + 10^1-the-fourth-highest-rank

+ the-lowest-rank

* + 1. PokerCardTable::getRoyalFlushScore():

The total Royal Flush score of cards in a hand equals to the basic Royal Flush score (10000000 pts) + the hand score.

* + 1. PokerCardTable::getStraightFlushScore():

The total Straight Flush score of cards in a hand equals to the basic Straight Flush score (9000000 pts) + the hand score.

* + 1. PokerCardTable::getFlushScore():

The total Flush score of cards in a hand equals to the basic Flush score (6000000 pts) + the hand score.

* + 1. PokerCardTable::getStraightScore():

The total Flush score of cards in a hand equals to the basic Straight score (5000000 pts) + the hand score.

* + 1. PokerCardTable::getFourOfAKindScore():

The total Four of a Kind score of cards in a hand equals to its basic score (8000000 pts) + the rank of the middle card.

* + 1. PokerCardTable::getFullHouseScore():

The total Full House score of cards in a hand equals to its basic score (7000000 pts) + the rank of the middle card.

* + 1. PokerCardTable::getThreeOfAKindScore():

The total Three of a Kind score of cards in a hand equals to its basic score (4000000 pts) + the rank of the middle card.

* + 1. PokerCardTable::getTwoPairScore():

The total Two Pair score of cards in a hand equals to its basic score (3000000 pts) + 10^2\*The-First-Highest-Two-Pair + 10\*The-Second-High-Two-Pair + Low-Card

* + 1. PokerCardTable::getAPairScore():

The total A Pair score of cards in a hand equals to its basic score (2000000 pts) + 10^3\*A-Pair + 10^2\*First-High-Card + 10\*Second-High-Card + Low-Card

* 1. Variables:

|  |  |  |
| --- | --- | --- |
| Type | Name | Location |
| vector<Card\*> | crCards  (The player’s current cards) | class AbstracCardTable  class PokerCardTable  class BlackJackCardTable |
| vector<Card\*> | crDCrds  (The dealer’s current cards) |
| vector<Card\*> | crSRnks  (The player’s current cards sorted by rank) | class PokerCardTable |
| vector<Card\*> | crSSuits  (The player’s current cards sorted by suit) |
| set<int> | rIndexes  (The on-screen replacing indexes) | method PokerCardTable::replaceCards() |

* 1. Concepts:

|  |  |
| --- | --- |
| Concept | Location |
| 1. Memory allocation | * main function. * AbstractCardTable::populateCardBy() |
| 1. Memory deallocation | * AbstractCardTable::deleteCards() * PokerCardTable::deleteCardsBy() |
| 1. Data structure | * Card in AbstractCardTable |
| 1. enum data type | * class AbstractCardTable * class CardTableHelper |
| 1. Functions with structures | * class AbstractCardTable * class PokerCardTable |
| 1. Arrays with structures | * class AbstractCardTable * class PokerCardTable |
| 1. Pointer arrays | * AbstractCardTable::displayCards() |
| 1. Character arrays | * class CardTableHelper. |
| 1. String object | * main function. * class AbstractCardTable * class CardTableHelper |
| 1. Binary file IO | * class CardTableHelper |
| 1. Object Oriented Programming (OOP) | * class CardTableHelper * class AbstractCardTable * class PokerCardTable * class BlackJackCardTable |
| 1. Static members |
| 1. Operator overloading | * class CardTableHelper |
| 1. Inheritance | * class PokerCardTable * class BlackJackCardTable |
| 1. Polymorphism & Virtual functions | * class AbstractCardTable * class PokerCardTable * class BlackJackCardTable |
| 1. Abstract base classes & Pure virtual functions |
| 1. Exceptions | * class CardTableHelper |
| 1. Function templates |
| 1. Standard Templates Library (STL) | * class AbstractCardTable * class PokerCardTable * class BlackJackCardTable |

1. Program (Header Files).
   1. AbstractCardTable.h

/\*

\* File: AbstractCardTable.h

\* Author: HaoHuynh

\*

\* Created on November 20, 2015, 10:03 AM

\*/

#ifndef ABSTRACTCARDTABLE\_H

#define ABSTRACTCARDTABLE\_H

#include <cstdlib>

#include <iostream>

#include <vector>

#include <string>

#include <time.h>

#include "CardTableHelper.h"

using namespace std;

/\*

\* This class is used for managing user's cards base on the below logical mapping table

\*

\* 0 1 2 3 4 5 6 7 8 9 10 11 12

\* Suits/Ranks 2 3 4 5 6 7 8 9 10 Jack Queen King Ace

\* 0 Hearts 0 1 2 3 4 5 6 7 8 9 10 11 12

\* 1 Diamonds 13 14 15 16 17 18 19 20 21 22 23 24 25

\* 2 Clubs 26 27 28 29 30 31 32 33 34 35 36 37 38

\* 3 Spades 39 40 41 42 43 44 45 46 47 48 49 50 51

\*/

class AbstractCardTable {

protected:

/\*

\* There are thirteen ranks per suit: from Ace to King.

\*/

enum CARD\_RANKS {

TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE

};

/\*

\* There are four suits in fifty two desk cards.

\*/

enum CARD\_SUITS {

HEARTS, DIAMONDS, CLUBS, SPADES

};

/\*

\* A card is defined by a combination of rank and suit.

\* The id is the index of a card in fifty two desk cards

\*/

typedef struct CARD {

int id;

CARD\_SUITS suit;

CARD\_RANKS rank;

} Card;

/\*

\* The total of cards are 52

\*/

static const int CARDS\_TOTAL = 52;

/\*

\* There are thirteen ranks per suit.

\*/

static const int RANKS\_PER\_SUIT = 13;

/\*

\* There are 4 suits in 52 desk cards.

\*/

static const int SUIT\_MAX = 4;

/\*

\* Labels for displaying a card suit.

\*/

static const string CARD\_SUIT\_LABELS[SUIT\_MAX];

/\*

\* Labels for displaying a card rank.

\*/

static const string CARD\_RANK\_LABELS[RANKS\_PER\_SUIT];

/\*

\* Vector contains all the user's current cards

\*/

vector<Card\*> crCards;

/\*

\* Vector contains all the current dealer cards

\*/

vector<Card\*> crDCrds;

public:

/\*\*

\* The default constructor of AbstractCardTable

\*/

AbstractCardTable();

/\*\*

\* This function interacts with players to drive them through a game

\*/

virtual CardTableHelper::GAME\_BOOL populateConsole() = 0;

protected:

/\*\*

\* This function create a Card by mapping the id to a Card structure through

\* the logical Card Table

\* @param id

\* @return Card

\*/

Card\* populateCardBy(int id);

/\*\*

\* This function sends all current cards to the console

\* @param cards

\* @param subject : who has the cards

\*/

void displayCards(vector<Card\*> cards, string subject = "Player");

/\*\*

\* This function generates a new card for the replacing process.

\* The new one should not be duplicated with any existing card.

\* @return a new Card structure

\*/

Card\* dealsNewCard();

/\*\*

\* This function deallocates the memories of all elements in cards and clears the cards

\* @param cards : a vector of Card Structure pointers

\*/

void deleteCards(vector<Card\*>& cards);

/\*\*

\* This function uses the id to check if a card has been dealt

\* @param id : the Card Table index

\* @return true/false : A card is existed or not

\*/

bool isCardExistedBy(int id);

/\*\*

\* This function deals first five poker cards for the player

\* @param nOfCds : the number of cards need dealing

\*/

void dealsCards(int nOfCds);

/\*\*

\* This function checks if the current hand win or lose

\* @return -1/0/1: /drawn/lose/win/

\*/

virtual CardTableHelper::GAME\_BOOL isPlayerWin() = 0;

/\*\*

\* This function reset all the current cards for a new game

\*/

virtual void clean() = 0;

};

#endif /\* ABSTRACTCARDTABLE\_H \*/

* 1. PokerCardTable.h

/\*

\* File: PokerCardTable.h

\* Author: HaoHuynh

\*

\* Created on October 10, 2015, 10:49 AM

\*/

#ifndef POKERCARDTABLE\_H

#define POKERCARDTABLE\_H

#include <set>

#include "AbstractCardTable.h"

using namespace std;

/\*

\* This class manages the table of a Poker Game

\*/

class PokerCardTable : public AbstractCardTable {

private:

/\*

\* Maximum poker cards for each player.

\*/

static const int FIVE\_POKER\_CARDS = 5;

/\*

\* The first poker card index

\*/

static const int INDEX\_0 = 0;

/\*

\* The second poker card index

\*/

static const int INDEX\_1 = 1;

/\*

\* The third poker card index

\*/

static const int INDEX\_2 = 2;

/\*

\* The fourth poker card index

\*/

static const int INDEX\_3 = 3;

/\*

\* The fifth poker card index

\*/

static const int INDEX\_4 = 4;

/\*

\* The base score of a royal flush

\*/

static const int ROYAL\_FLUSH\_SCORE = 10000000;

/\*

\* The base score of a straight flush

\*/

static const int STRAIGHT\_FLUSH\_SCORE = 9000000;

/\*

\* The base score of Four of a Kind

\*/

static const int FOUR\_OF\_A\_KIND\_SCORE = 8000000;

/\*

\* The base score of Full House

\*/

static const int FULL\_HOUSE\_SCORE = 7000000;

/\*

\* The base score of a Flush

\*/

static const int FLUSH\_SCORE = 6000000;

/\*

\* The base score of a Straight

\*/

static const int STRAIGHT\_SCORE = 5000000;

/\*

\* The base score of Three of a Kind

\*/

static const int THREE\_OF\_A\_KIND\_SCORE = 4000000;

/\*

\* The base score of Two Pairs

\*/

static const int TWO\_PAIR\_SCORE = 3000000;

/\*

\* The base score of a Pair

\*/

static const int A\_PAIR\_SCORE = 2000000;

/\*

\* The coefficient is used for calculating a player's score

\*/

static const int BASE\_SCORE = 10;

/\*

\* Vector contains all the user's current cards sorted by rank

\*/

vector<Card\*> crSRnks;

/\*

\* Vector contains all the user's current cards sorted by suit

\*/

vector<Card\*> crSSuits;

public:

/\*\*

\* The default constructor of PokerCardTable

\*/

PokerCardTable();

/\*\*

\* The default destructor of PokerCardTable

\*/

~PokerCardTable();

/\*\*

\* This function interacts with players to drive them through a game

\*/

virtual CardTableHelper::GAME\_BOOL populateConsole();

private:

/\*\*

\* This function generates a new one that has not been dealt before

\* for the replacement process

\* @return a new Card structure

\*/

Card\* getNewCardForReplacement();

/\*\*

\* This functions collects all the cards user want to replace and

\* replaces them with new ones

\*/

void replaceCards();

/\*\*

\* This functions creates a list of current hand cards sorted by rank.

\*/

void sortCardsByRank(const vector<Card\*>& cards);

/\*\*

\* This functions creates a list of current hand cards sorted by suit.

\*/

void sortCardsBySuit(const vector<Card\*>& cards);

/\*\*

\* This functions check if there is an ACE, and all five poker cards are flush and straight

\* @return true/false

\*/

bool isRoyalFlush();

/\*\*

\* This functions check if all five poker card are flush and straight

\* @return true/false

\*/

bool isStraightFlush();

/\*\*

\* This function checks if all five poker cards have the same suit.

\* After sorting by suit, if the lowest card has the same suit as the highest one,

\* then all five poker cards will have the same suit.

\* @return true/false

\*/

bool isFlush();

/\*\*

\* This function checks if all five poker cards make a straight:

\* Case 1: There is an ACE and the other fours are {TEN, JACK, QUEEN, KING}

\* or {TWO, THREE, FOUR, FIVE}

\* Case 2: The ranks of all cards are increasing continuously

\* @return true/false

\*/

bool isStraight();

/\*\*

\* This function checks for four of the same card in a rank

\* Two cases: 4 + 1 or 1 + 4

\* @return true/false

\*/

bool isFourOfAKind();

/\*\*

\* This function checks for 3 of the same card in a rank

\* and 2 of the same card in another rank

\* Two cases: 3 + 2 or 2 + 3

\* @return true/false

\*/

bool isFullHouse();

/\*\*

\* This function checks for 3 of the same card after checking four of a kind and full house.

\* Three cases: 3 + 1 + 1 or

\* 1 + 3 + 1 or

\* 1 + 1 + 3.

\* @return true/false

\*/

bool isThreeOfAKind();

/\*\*

\* This function checks for 2 different pairs after checking four of a kind, full house,

\* and three of a kind.

\* Three cases: 2 + 2 + 1 or

\* 2 + 1 + 2 or

\* 1 + 2 + 2.

\* @return true/false

\*/

bool isTwoPairs();

/\*\*

\* This function checks for two of the same card after checking four of a kind, full house,

\* three of a kind and two pairs

\* Four cases: 2 + 1 + 1 + 1 or

\* 1 + 2 + 1 + 1 or

\* 1 + 1 + 2 + 1 or

\* 1 + 1 + 1 + 2 or

\* @return true/false

\*/

bool isPair();

/\*\*

\* This function checks if the current hand win or lose

\* @return -1/0/1: /drawn/lose/win/

\*/

virtual CardTableHelper::GAME\_BOOL isPlayerWin();

/\*\*

\* This function deallocates the memories of a set of elements in cards and clears the cards

\* @param cards : a vector of Card Structure pointers

\* @param poss : a set of positions in cards

\*/

void deleteCardsBy(vector<Card\*>& cards, const set<int>& poss);

/\*\*

\* This function reset all the current cards for a new game

\*/

virtual void clean();

/\*\*

\* This function calculates the total score of cards in a hand by:

\* 10^4\*the-first-highest-rank + 10^3\*the-second-highest-rank

\* + 10^2\*the-third-highest-rank + 10^1-the-fourth-highest-rank + the-lowest-rank

\* @param cards

\* @return the total score of cards in a hand

\*/

int getHandScore();

/\*\*

\* This function calculates the total Royal Flush score of cards in a hand

\* @param cards

\* @return

\*/

int getRoyalFlushScore();

/\*\*

\* This function calculates the total Straight Flush score of cards in a hand

\* @param cards

\* @return

\*/

int getStraightFlushScore();

/\*\*

\* This function calculates the total Flush score of cards in a hand

\* @param cards

\* @return

\*/

int getFlushScore();

/\*\*

\* This function calculates the total Flush score of cards in a hand

\* @param cards

\* @return

\*/

int getStraightScore();

/\*\*

\* This function calculates the total Four Of A Kind score of cards in a hand

\* @param cards

\* @return

\*/

int getFourOfAKindScore();

/\*\*

\* This function calculates the total Full House score of cards in a hand

\* @param cards

\* @return

\*/

int getFullHouseScore();

/\*\*

\* This function calculates the total Three Of A Kind score of cards in a hand

\* @param cards

\* @return

\*/

int getThreeOfAKindScore();

/\*\*

\* This function calculates the total Two Pair score of cards in a hand by:

\* TWO\_PAIR\_SCORE + 10^2\*First-Hight-Two-Pair + 10\*Second-Hight-Two-Pair + Low-Card

\* @param cards

\* @return

\*/

int getTwoPairScore();

/\*\*

\* This function calculates the total Pair score of cards in a hand by:

\* A\_PAIR\_SCORE + 10^3\*A-Pair + 10^2\*First-Hight-Card + 10\*Second-Hight-Card + Low-Card

\* @param cards

\* @return

\*/

int getAPairScore();

/\*\*

\* This function calculates the final Poker score of cards in a hand base on the wining conditions

\* @param cards

\* @return The final Poker score

\*/

int calPokerScore(const vector<Card\*>& cards);

/\*\*

\* This method creates one more chance for the dealer to win a game

\*/

void replaceAllDealerCards();

};

#endif /\* POKERCARDTABLE\_H \*/

* 1. BlackJackCardTable.h

/\*

\* File: BlackJackCardTable.h

\* Author: HaoHuynh

\*

\* Created on November 20, 2015, 10:05 AM

\*/

#ifndef BLACKJACKCARDTABLE\_H

#define BLACKJACKCARDTABLE\_H

#include "AbstractCardTable.h"

class BlackJackCardTable : public AbstractCardTable {

private:

/\*

\* Initiated BlackJack cards for each player.

\*/

static const int BJ\_INIT\_N\_CARDS = 2;

/\*

\* Limit the number of BlackJack cards a player can have.

\*/

static const int BJ\_PLAYER\_CARD\_LIMIT = 12;

/\*

\* BlackJack player request to hit

\*/

static const int BJ\_PLAYER\_HIT = 1;

/\*

\* The highest BlackJack winning score

\*/

static const int BJ\_UPPER\_WINNER\_LIMIT = 21;

/\*

\* The lowest BlackJack winning score

\*/

static const int BJ\_LOWER\_WINNER\_LIMIT = 17;

/\*

\* BlackJack bust means the total score is over 21 and who has this score will be the loser

\*/

static const int BJ\_BUST = 0;

/\*

\* The BlackJack hard value of a Ace

\*/

static const int BJ\_HARD\_ACE = 1;

/\*

\* The BlackJack soft value of a Ace

\*/

static const int BJ\_SOFT\_ACE = 11;

/\*

\* The BlackJack value of Jack, Queen, and King

\*/

static const int BJ\_JACK\_QUEEN\_KING = 10;

/\*

\* Used for testing the soft-17 case in BlackJack

\*/

bool isSAce;

public:

/\*\*

\* The default constructor of BlackJackCardTable

\*/

BlackJackCardTable();

/\*\*

\* The default destructor of BlackJackCardTable

\*/

~BlackJackCardTable();

/\*\*

\* This function interacts with players to drive them through a game

\*/

virtual CardTableHelper::GAME\_BOOL populateConsole();

private:

/\*\*

\* This function checks if the current hand win or lose

\* @return -1/0/1: /drawn/lose/win/

\*/

virtual CardTableHelper::GAME\_BOOL isPlayerWin();

/\*\*

\* This function reset all the current cards for a new game

\*/

virtual void clean();

/\*\*

\* This function calculates the total BlackJack score of the current hand or the dealer

\* @param cards

\* @return BJ score

\*/

int calculateScore(vector<Card\*> cards);

/\*\*

\* This function sends all current dealer cards to console

\* @param isPF : is the player done his or her turn.

\*/

void displayDealerCards(bool isPD = false);

/\*\*

\* Process player's request for hitting or staying.

\* Calculate user's BJ score after the player's turn.

\*/

void processPlayerTurn();

/\*\*

\* Try getting the BJ score of dealer between hard 17 and 21.

\* Calculate the actual BJ score.

\*/

void processDealerTurn();

};

#endif /\* BLACKJACKCARDTABLE\_H \*/

* 1. CardTableHelper.h

/\*

\* File: PokerHelper.h

\* Author: HaoHuynh

\*

\* Created on October 10, 2015, 10:35 AM

\*/

#ifndef POKERHELPER\_H

#define POKERHELPER\_H

#include <string>

#include <fstream>

#include <iostream>

using namespace std;

class CardTableHelper; // Forward Declaration

// Function Prototypes for Overloaded Stream Operators

ostream &operator <<(ostream &, const CardTableHelper &);

/\*

\* Class Helper

\*/

class CardTableHelper {

private:

/\*

\* Relative maximum lines of a monitor

\*/

static const int MAX\_MONITOR\_LINES = 200;

//The content of menu options

static const char\* MENU\_CONTENT;

/\*\*

\* The default file that stores the current bank roll

\*/

static const string DEFAULT\_FILE\_NAME;

/\*

\* Singleton instance

\*/

static const CardTableHelper cTHelper;

public:

//Initiated money a player can have

static const int MAX\_BANK\_ROLL = 100;

//Out of money

static const int MIN\_BANK\_ROLL = 0;

//Gambling Menu Options

static enum GAMBLING\_MENU {

POKER\_GAME = 1, BLACK\_JACK = 2, SAVE = 3, LOAD = 4, RESTART = 5, EXIT = 6

} GamblingMenu;

/\*\*

\* The bool type for this program

\*/

static enum GAME\_BOOL {

DRAWN = -1, LOSE = 0, WIN = 1

} GameBool;

public:

/\*\*

\* Clear the screen after a certain event.

\*/

static void clearMonitor();

/\*\*

\* Validate input of an integer number from lowerLimit to upperLimit

\* @param number

\* @param lowerLimit

\* @param upperLimit

\*/

static void validateValueOf(int& number, int lowerLimit, int upperLimit);

/\*\*

\* Clean up the cin buffer

\*/

static void cleanCin();

/\*\*

\* This function saves the current bank roll to a file

\* @param crBkRoll : the current bank roll

\* @param fName : a file name

\*/

template <typename T>

static void save(T crBkRoll) {

fstream\* fs = new fstream(DEFAULT\_FILE\_NAME.c\_str(), ios::out | ios::binary);

fs->exceptions(fstream::failbit | fstream::badbit);

try {

fs->write(reinterpret\_cast<char\*> (&crBkRoll), sizeof (crBkRoll));

} catch (fstream::failure e) {

cerr << "File IO exception!!! \n";

}

fs->close();

}

/\*\*

\* This function loads the current bank roll from a file

\* @param crBkRoll : the current bank roll

\* @param fName : a file name

\*/

template <typename T>

static void load(T& crBkRoll) {

fstream\* fs = new fstream(DEFAULT\_FILE\_NAME.c\_str(), ios::in | ios::binary);

fs->exceptions(fstream::failbit | fstream::badbit);

try {

fs->read(reinterpret\_cast<char\*> (&crBkRoll), sizeof (crBkRoll));

} catch (fstream::failure e) {

cerr << "File IO exception!!! \n";

}

fs->close();

}

//Singleton pattern

static CardTableHelper getInstance();

// Friend function

friend ostream &operator <<(ostream &, const CardTableHelper &);

};

#endif /\* POKERHELPER\_H \*/