Project Four: Blackjack

Out: July 13, 2017; Due: July 24, 2017

I. Motivation

To give you experience in implementing abstract data types (ADTs), using interfaces (abstract base classes), and using interface/implementation inheritance.

II. Introduction

In this project, we will implement a simplified version of the card game called **Blackjack**, also sometimes called 21. It is a relatively simple game played with a standard deck of 52 playing cards. There are two participants, a **dealer** and a **player**. The player starts with a bankroll, and the game progresses in rounds called **hands**.

At the start of each hand, the player decides how much to wager on this hand. It can be any amount between some **minimum** allowable wager and the player's total bankroll, *inclusive*.

After the wager, the dealer deals a total of four cards: the first face-up to the player, the second face-up to himself, the third face-up to the player, and the fourth face-down to himself. The second card of the dealer which is face-down is called the **hole card**.

The player then examines his cards, forming a total. Each card 2-10 is worth its spot value; each face card (jack, queen, king) is also worth 10. An ace is worth either 1 or 11--whichever is more advantageous to the player (i.e., as close to 21 as possible without going over). If the total includes an ace counted as 11, the total is called "**soft**", otherwise it is called "**hard**".

The game progresses first with the player, then the dealer. The player's goal is to build a hand that is as close to 21 as possible without going over---the latter is called a "**bust**", and a player who busts loses the hand without forcing the dealer to play. As long as the player believes another card will help, the player "**hits**"---asks the dealer for another card. Each of these additional cards is dealt **face-up**. This process ends either when the player decides to "**stand**"---ask for no cards, or the player busts. Note that a player can stand with two cards; one need not hit at all in a hand.

If the player is dealt an ace plus any ten or face card (jack, queen, king), the player's hand is called a "**natural 21**", and the player's wager is paid off with 3 to 2 odds, without examining the dealer's cards. In other words, if the player had wagered 10, the player would win 15 (i.e., his bankroll will increase by 15) if dealt a natural 21. Note that, since we are working with integers, you'll have to be a bit careful with the 3/2 payout. For example, a wager of 5 would pay 7 if a natural 21 is dealt, since (3*5)/2 is 7 in integer arithmetic.

If the player neither busts nor is dealt a natural 21, play then progresses to the dealer. The dealer **must** hit until he either reaches a total greater than or equal to 17 (hard or soft), or busts. If the dealer busts, the player wins. Otherwise, the two totals are compared. If the dealer's total is higher, the player's bankroll decreases by the amount of his wager. If the player's total is higher, his bankroll increases by the amount of his wager. If the totals are equal, the bankroll is unchanged; this is called a "**push**".

Note that this is a very simplified form of the game: we do not split pairs, allow double-down bets, or take insurance, etc. (http://en.wikipedia.org/wiki/Blackjack)

III. Programming Assignment

You will provide one or more implementations of four separate abstractions for this project: a deck of cards, a blackjack hand, a blackjack player, and a game driver. All files referenced in this specification are located in the Projects/Project-4 folder on Canvas.

You may copy them to your private directory space, but **do not modify them in any way**. This will help ensure that your submitted project compiles correctly. For this project, the penalty for code that does not compile will be **severe**, regardless of the reason.

1. The Deck ADT

Your first task is to implement the following ADT representing a deck of cards:

```
class DeckEmpty { // An exception type
};

const int DeckSize = 52;

class Deck {

    // A standard deck of 52 playing cards---no jokers

    Card deck[DeckSize]; // The deck of cards
```

```
    int next; // The next card to deal

public:

    Deck();
    // EFFECTS: constructs a "newly opened" deck of cards. first the
    // spades from 2 to A, then the hearts, then the clubs, then the
    // diamonds. The first card dealt should be the 2 of Spades.

    void reset();
    // EFFECTS: resets the deck to the state of a "newly opened" deck
    // of cards.

    void shuffle(int n);
    // REQUIRES: n is between 0 and 52, inclusive.
    // MODIFIES: this
    // EFFECTS: cut the deck into two segments: the first n cards,
    // called the "left", and the rest called the "right". Note that
    // either right or left might be empty. Then, rearrange the deck
    // to be the first card of the right, then the first card of the
    // left, the 2nd of right, the 2nd of left, and so on. Once one
    // side is exhausted, fill in the remainder of the deck with the
    // cards remaining in the other side. Finally, make the first
    // card in this shuffled deck the next card to deal. For example,
    // shuffle(26) on a newly-reset() deck results in: 2-clubs,
    // 2-spades, 3-clubs, 3-spades ... A-diamonds, A-hearts.
    //
    // Note: if shuffle is called on a deck that has already had some
    // cards dealt, those cards should first be restored to the deck
    // in the order in which they were dealt, preserving the most
    // recent post-shuffled/post-reset state. After shuffling, the
    // next card to deal is the first one in the deck.

    Card deal();
    // MODIFIES: this
    // EFFECTS: deals the "next" card and returns that card.
    // If no cards remain, throws an instance of DeckEmpty.

    int cardsLeft();
    // EFFECTS: returns the number of cards in the deck that have not
    // been dealt since the last reset/shuffle.
};
```

The Deck ADT is specified in `deck.h`. The Deck ADT depends on the following Card type:

```
    enum Suit {
        SPADES, HEARTS, CLUBS, DIAMONDS, SUIT_SIZE
    };

    extern const char *SuitNames[SUIT_SIZE];
```

```
    enum Spot {
        TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
        JACK, QUEEN, KING, ACE, SPOT_SIZE
    };

    extern const char *SpotNames[SPOT_SIZE];

    struct Card {
        Spot spot;
        Suit suit;
    };
```

which is declared in `card.h`, implemented by `card.cpp`, and included by `deck.h`. The file `card.cpp` defines `SpotNames` and `SuitNames` for you, so that `SuitNames[HEARTS]` evaluates to "Hearts", and so on.

You are asked to put your implementation of this ADT in a file named `deck.cpp`.

2. The Hand Interface

Your second task is to implement the following ADT representing a blackjack hand:

```
struct HandValue {
    int  count;    // Value of hand
    bool soft;     // true if hand value is a soft count
};

class Hand {
    // OVERVIEW: A blackjack hand of zero or more cards

    // Note: this really is the only private state you need!
    HandValue curValue;

public:

    Hand();
    // EFFECTS: establishes an empty blackjack hand.

    void discardAll();
    // MODIFIES: this
    // EFFECTS: discards any cards presently held, restoring the state
    // of the hand to that of an empty blackjack hand.

    void addCard(Card c);
    // MODIFIES: this
    // EFFECTS: adds the card "c" to those presently held.
```

```
    HandValue handValue() const;
    // EFFECTS: returns the present value of the blackjack hand.  The
    // count field is the highest blackjack total possible without
    // going over 21.  The soft field should be true if and only if at
    // least one ACE is present, and its value is counted as 11 rather
    // than 1.  If the hand is over 21, any value over 21 may be
    // returned.
    //
    // Note: the const qualifier at the end of handValue means that
    // you are not allowed to change any member variables inside
    // handValue. Adding this prevents the accidental change by you.
};
```

The Hand ADT is specified in `hand.h` The Hand ADT depends on the `Card` type, and includes `card.h`. You are asked to put your implementation of this ADT in a file named `hand.cpp`.

3. The Player Interface

Your third task is to implement two different blackjack players. The interface for a Player is:

```
class Player {
    // A virtual base class, providing the player interface

public:
    virtual int bet(unsigned int bankroll,
                    unsigned int minimum) = 0;
    // REQUIRES: bankroll >= minimum
    // EFFECTS: returns the player's bet, between minimum and bankroll
    // inclusive

    virtual bool draw(Card dealer,           // Dealer's "up card"
                    const Hand &player) = 0; // Player's current hand
    // EFFECTS: returns true if the player wishes to be dealt another
    // card, false otherwise.

    virtual void expose(Card c) = 0;
    // EFFECTS: allows the player to "see" the newly-exposed card c.
    // For example, each card that is dealt "face up" is expose()d.
    // Likewise, if the dealer must show his "hole card", it is also
    // expose()d.  Note: not all cards dealt are expose()d---if the
    // player goes over 21 or is dealt a natural 21, the dealer need
    // not expose his hole card.

    virtual void shuffled() = 0;
    // EFFECTS: tells the player that the deck has been re-shuffled.
```

```
};
```

The Player ADT is specified in `player.h` The Player ADT depends on the `Hand` type, and includes `hand.h`. You are to implement two different derived classes from this interface.

The first derived class is the Simple player, who plays a simplified version of basic strategy for blackjack. The simple player always places the minimum allowable wager, and decides to hit or stand based on the following rules and whether or not the player has a "**hard count**" or "**soft count**":

The first set of rules apply if the player has a "**hard count**", i.e., his best total counts an Ace (if any) for 1, not 11, or if his hand does not contain an Ace.

- If the player's hand totals 11 or less, he always hits.
- If the player's hand totals 12, he stands if the dealer shows 4, 5, or 6; otherwise he hits.
- If the player's hand totals between 13 and 16 inclusive, he stands if the dealer shows a 2 through a 6 inclusive; otherwise he hits.
- If the player's hand totals 17 or greater, he always stands.

The second set of rules applies if the player has a "**soft count**"---his best total includes **one** Ace worth 11. (Note that a hand would never count two Aces as 11 each--that's a bust of 22.)

- If the player's hand totals 17 or less, he always hits.
- If the player's hand totals 18, he stands if the dealer shows a 2, 7, or 8, otherwise he hits.
- If the player's hand totals 19 or greater, he always stands.

Note: the Simple player does nothing for **expose** and **shuffled** events.

The second derived class is the Counting player. This player counts cards in addition to playing the basic strategy. The intuition behind card counting is that when the deck has more face cards (worth 10) than low-numbered cards, the deck is favorable to the player. The converse is also true.

The Counting player keeps a running "count" of the cards he has seen from the deck. Each time he sees (via the `expose()` method) a 10, Jack, Queen, King, or Ace, he subtracts one from the count. Each time he sees a 2, 3, 4, 5, or 6, he adds one to the count. When he sees that the deck is `shuffled()`, the count is reset to zero. Whenever the count is +2 or greater **and** he has enough bankroll (**larger than or equal to** the double of the minimum), the Counting player bets double the minimum, **otherwise** (i.e., including the situation where count >= +2 but the bankroll is less than the double of the minimum) he bets the minimum. The Counting player should not re-implement methods of the Simple player unnecessarily.

The code for both of these Players must be implemented in a file named `player.cpp`. You must also declare a static global instance of each of the two Players you implement in your `player.cpp` file. Finally, you should implement the following "access" functions that return pointers to each of these two global instances in your `player.cpp` file.

```
extern Player *get_Simple();
extern Player *get_Counting();
```

4. The Driver program

Finally, you are to implement a driver program that can be used to simulate this version of blackjack given your implementation of the ADTs described above.

You are asked to put your implementation of this driver program in a file named `blackjack.cpp`.

The driver program, when run, takes arguments as follows:

```
<bankroll> <min-bet> <hands> <simple|counting> [<input_file_name>]
```

The first argument is an integer denoting the player's starting bankroll. The second argument is the minimum bet of the game. The third argument is the maximum number of hands to play in the simulation. You can assume that these three numbers provided by the user are positive (≥ 1) and within an upper limit of 10000. The fourth argument is one of the two strings "simple" or "counting", denoting which of the two players to use in the simulation. The first four arguments are mandatory. The fifth argument is optional. If there is this fifth argument, it specifies a given input file, based on which the driver should execute. The detailed behavior will be discussed later. You can assume that we specify correct command-line arguments. Thus, no error checking is needed for handling command-line arguments.

For example, suppose that your program is called `blackjack`. It may be invoked by typing in a terminal:

```
./blackjack 100 5 3 simple
```

Then your program simulates the simple player playing 3 hands with an initial bankroll of 100 and the minimum bet of 5.

It may also be invoked as follows:

```
./blackjack 100 5 3 counting test.in
```

In this case, your program simulates the counting player playing 3 hands with an initial bankroll of 100 and the minimum bet of 5. Meanwhile, your program should read from an external file named “test.in”, and execute according to the input file.

The driver first shuffles the deck. When there is no input file specified, in order to shuffle the deck, you choose **seven** cuts between 13 and 39 inclusive **at random**, shuffling the deck with each of these cuts. We refer to this type of shuffle as **random shuffle**. We have supplied a header, `rand.h`, and an implementation, `rand.cpp`, that define a function `get_cut()` that provides these random cuts.

When there is an input file specified, the driver program should first read from the given input file, which is formatted as follows:

```
15 20 39 22 38 13 27 30
```

The input file just has a single line, which is composed of a sequence of integers. It specifies how the deck should be shuffled **initially**. The *i*-th integer specifies the position of the *i*-th cut in the initial shuffle. Each integer is guaranteed to be valid, i.e., between 0 and 52, inclusive. The number of integers in the sequence may not be seven, which is the number of cuts in the random shuffle. You can assume that this sequence is non-empty. Note that the input file only specifies the cuts for the initial shuffle. If there are any more shuffles occurred later, they are random shuffles.

Each time the deck is shuffled, first announce it:

```
cout << "# Shuffling the deck\n";
```

And announce each of the cut points:

```
cout << "cut at " << cut << endl;
```

then be sure to tell the player via `shuffle()`.

Note: you should always print the message corresponding to the initial shuffle before you do anything further.

While the player's bankroll is larger than or equal to the minimum bet and there are hands left to be played:

- Announce the hand:

  ```
  cout << "# Hand " << thishand << " bankroll " << bankroll << endl;
  ```

  where the variable `thishand` is the hand number, **starting from 1**.

- If there are fewer than 20 cards left, reshuffle the deck using **random shuffle**. Note that this happens only at the beginning of each hand. It does not occur during a hand even if the number of cards is fewer than 20.

- Ask the player for a wager and announce it:

  ```
  cout << "# Player bets " << wager << endl;
  ```

- Deal four cards: one face-up to the player, one face-up to the dealer, one face-up to the player, and one face-down to the dealer. Announce the face-up cards using `cout`. For example:

  ```
  Player dealt Ace of Spades
  Dealer dealt Two of Hearts
  ```

  Use the `SpotNames` and `SuitNames` arrays for this, and be sure to `expose()` any face-up cards to the player.

- If the player is dealt a natural 21, immediately pay the player 3/2 of his bet. Note that, since we are working with integers, you'll have to be a bit careful with the 3/2 payout. For example, a wager of 5 would pay 7 if a natural 21 is dealt, since (3*5)/2 is 7 in integer arithmetic. In this case, announce the win:

  ```
  cout << "# Player dealt natural 21\n";
  ```

- If the player is not dealt a natural 21, have the player play his hand. Draw cards until the player either stands or busts. Announce and `expose()` each card dealt as above.

- Announce the player's total

  ```
  cout << "Player's total is " << player_count << endl;
  ```

  where the variable `player_count` is the total value of the player's hand. If the player busts, say so:

  ```
  cout << "# Player busts\n";
  ```

deducting the wager from the bankroll and moving on to the next hand.

- If the player hasn't busted, announce and expose the dealer's hole card. For example:

  ```
  Dealer's hole card is Ace of Spades
  ```

  (Note: the hole card is NOT exposed if either the player busts or is dealt a natural 21.)

- If the player hasn't busted, play the dealer's hand. The dealer must hit until reaching seventeen or busting. Announce and expose each card as above.

- Announce the dealer's total

  ```
  cout << "Dealer's total is " << dealer_count << endl;
  ```

  where the variable `dealer_count` is the total value of the dealer's hand. If the dealer busts, say so

  ```
  cout << "# Dealer busts\n";
  ```

  crediting the wager from the bankroll and moving on to the next hand.

- If neither the dealer nor the player bust, compare the totals and announce the outcome. Credit the bankroll, debit it, or leave it unchanged as appropriate.

  ```
  cout << "# Dealer wins\n";
  cout << "# Player wins\n";
  cout << "# Push\n";
  ```

- If the player's bankroll is larger than or equal to the minimum bet and there are hands left to be played, then continue to play the next hand (i.e., start again from the first bullet point "Announce the hand").

Finally, when the player either has too little money to make a minimum wager or the allotted hands have been played, announce the outcome:

```
cout << "# Player has " << bankroll
     << " after " << thishand << " hands\n";
```

where the variable `thishand` is the current hand number. In the special case where the initial bankroll is less than the minimum, we have `thishand = 0`, since the player hasn't played any hand yet. Furthermore, in this special case, the initial shuffle of the deck should still be announced before you print the status of the player.

IV. Implementation Requirements and Restrictions

- You may `#include <iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, `<cstdlib>`, and `<cassert>`. No other system header files may be included, and you may not make any call to any function in any other library.
- Output should only be done where it is specified.
- You may not use the `goto` command.
- You may not have any global variables in the driver. You may use global state in the class implementations, but it must be static and (except for the two players) const.
- You may assume that functions are called consistent with their advertised specifications. This means you need not perform error checking. However, when testing your code in concert, you may use the `assert()` macro to program defensively.

V. Source Code Files and Compiling

There are five header files (`card.h`, `deck.h`, `hand.h`, `player.h`, and `rand.h`) and two C++ source files (`card.cpp` and `rand.cpp`) located in the Project-4-Related-Files.zip from our Canvas Resources:

You should copy these files into your working directory. **DO NOT modify them!**

You need to write four other C++ source files: `deck.cpp`, `hand.cpp`, `player.cpp`, and `blackjack.cpp`. They are discussed above and summarized below:

`deck.cpp`:	your Deck ADT implementation
`hand.cpp`:	your Hand ADT implementation
`player.cpp`:	your two player ADT implementations
`blackjack.cpp`:	your simulation driver

After you have written these files, you can type the following command in the terminal to compile the program:

```
g++ -Wall –o blackjack blackjack.cpp card.cpp deck.cpp hand.cpp
player.cpp rand.cpp
```

This will generate a program called `blackjack` in your working directory. In order to guarantee that the TAs compile your program successfully, you should name you source code files exactly like how they are specified above. For this project, the penalty for code that does not compile will be **severe**, regardless of the reason.

VI. Testing

For this project, you should write individual, focused test cases for all the ADT implementations. For these ADTs, determine the behaviors required of the implementation. Then, for each of these behaviors:

- Determine the **specific** behavior that the implementation must exhibit.
- Write a program that, when linked against the implementation of the ADT, tests for the presence/absence of that behavior.

For example, if you identify two behaviors in Deck implementation, you would have two files, each testing one behavior. You can name them as follows:

```
deck.case.1.cpp
deck.case.2.cpp
```

Your test cases for this project are considered "acceptance tests". The tests for your Hand/Deck ADT (each of which includes a `main()` function) should be linked against a correct `card.cpp` and a possibly incorrect `hand.cpp`/`deck.cpp` when you compile your program. The tests for your Player ADT (each of which includes a `main()` function) should be linked against a correct `card.cpp`, a correct `hand.cpp`, and a possibly incorrect `player.cpp` when you compile your program.

Your test case must decide, based on the results from calls to Hand/Deck/Player methods, whether the Hand/Deck/Player ADT is correct or incorrect. If your case believes the Hand/Deck/Player to be correct, it should return 0 from `main()`. If your case believes the Hand/Deck/Player to be incorrect, it should return any value other than zero (the value -1 is commonly used to denote failure). Do not compare the output of your test cases against correct/incorrect implementations. Instead, look at the return value of your program when it is run in Linux to see if you return the right value based upon whether your test finds an error in the implementation of the ADT you are testing.

In Linux you can check the return value of your program by typing

```
echo $?
```

immediately after running your program. You also may find it helpful to add error messages to your output.

Here is an example of code that tests a hypothetical "integer add" function (declared in `addInts.h`) with an "expected" test case:

```
// Tests the addInts function
#include "addInts.h"
int main()
{
    int x = 3;
    int y = 4;
    int answer = 7;
    int candidate = addInts(x, y);
    if (candidate == answer) {
        return 0;
    } else {
        return -1;
    }
}
```

You should write a collection of Hand/Deck/Player implementations with different, specific bugs, and make tests to identify the incorrect code.

We have supplied one example of a test source file `example.cpp` which tests the `shuffle(int)` method of the Deck ADT.

We have also supplied one simple set of output produced by a correct deck, hand, simple player, and driver. It is called `sample.txt`. To test your entire project, type the following into the Linux terminal once your program has been compiled:

```
./blackjack 100 5 3 simple > test.out
diff test.out sample.txt
```

If the `diff` program reports any differences at all, you have a bug.

VII. Submitting and Due Date

You should submit four source code files `deck.cpp`, `hand.cpp`, `player.cpp`, and `blackjack.cpp`. (You do not need to submit a `Makefile` for this project.) These files should be submitted via the online judgment system. See announcement from the TAs for details about submission. The due date is 11:59 pm on July 24th, 2017.

VIII. Grading

Your program will be graded along three criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style

Functional Correctness is determined by running a variety of test cases against your program, checking against our reference solution. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. For example, significant code duplication will lead to General Style deductions.