Lab Title: Infinite Deck Simple Blackjack
Course: CS 1410 – Object Oriented Programming

Objectives: enumerations, structures, user-defined functions, function definitions, parameters, argument-passing, function scope, pass-by-value, and pass-by-reference.

Overview

In this lab, you will create a simple game of blackjack with an infinite number of decks (in the real game of Blackjack, a finite number of cards are used which allows for a strategy known as counting. This lab will forgo this principle in exchange for simplicity). The purpose of this lab is to practice programming with enumerations and structures, which will be used to simulate playing cards. The lab will also serve to reinforce prior control structures, including functions, decisions, and loops. The lab is divided into three parts. In the first part, one `enum` and one `struct` are defined to simulate a playing card. In the second part, three functions are created to assist with game mechanics. Logic for a single hand of blackjack (with simple rules) will be programmed in the third part. A single `.cpp` file will be used for all parts of the lab.

Part 1: Enumeration and Structure

In this part of the lab, you will create an enumeration and a structure, which will be used to simulate a playing card. Complete the steps.

1. Define an `enum` named `Suit`. The enumeration will contain constants for the values `HEARTS`, `DIAMONDS`, `CLUBS`, `SPADES`. You may use the default values or assign integer values as you deem appropriate.

2. Define a `struct` named `Card`. The `struct` will have two data fields: one named `value` of type `int`, and another named `suit` of type `Suit`. (The value will represent the card's face value; the suit will represent the card's suit).

Part 2: Functions with Struct

In this section, you will write three functions that will later assist with game logic. The first function will generate a card at random. The second function will show the card on the console. The third function is used to deal a new card to a hand and update the hand totals. Complete the steps as described.

Function: **`getRandomCard`** – The purpose of this function is to generate a valid face card at random. When called, it will return a `Card` `struct` that has a random `value` and `suit`. Complete the steps as outlined.

Define a function named `getRandomCard`. The function has *no parameters*; the function has a return type of `Card` (the `struct` created in Part 1). The body of the function will perform the following:

1. Define a new `Card` `struct`.
2. Initialize the `value` field to a random number between 2 and 14.
3. Initialize the `suit` field to a random suit (`HEARTS`, `DIAMONDS`, `CLUBS`, `SPADES`) using the enumeration. You may use a `switch` or `if` statements to accomplish this. (Do not assign integer values directly).
4. Return the `Card` `struct` to the caller.

Function: **`showCard`** – The purpose of this function is to display a card on the console. The card should be provided by the caller and passed as an argument. Complete the steps.

Define a function named `showCard`. The function has a single parameter of type `Card`. The function does not return any value. The body of the function will perform the following:

1. Use the `value` field of the `Card` parameter to display the value of the card. Any value between 2 and 10 should display as itself. Any value greater than 10 should be shown as follows: 11 = J, 12 = Q, 13 = K, 14 = A. You may use a `switch` or `if` statements to accomplish this.
2. Use the `suit` field of the `Card` parameter to display the suit of the card. Consoles with UTF-8 (or ASCII compatible) encoding can use char values 3, 4, 5, and 6 (i.e. `char(3)`, `char(4)`, etc.) to display a "suit" symbol. Otherwise, simply use words or adorn the card with any standard text character. Again, you may use a `switch` or `if` statements to accomplish this.
3. Decorate the card with keyboard (or ASCII) characters as desired.

A sample call is provided for testing purposes (it is not part of the final product):

```
for (int i = 0; i < 16; i++) {
    Card card = getRandomCard();
    showCard(card);
}
```

Sample output (with ASCII characters):

```
|10♦||5♥||6♣||Q♣||J♣||5♦||7♠||3♣||4♥||J♣||3♣||4♣||10♦||7♠||10♦||A♥|
```

Alternate output (with standard letters):

```
|10S||4S||3D||5S||JS||6H||3D||4S||JH||JD||10H||AH||6H||8D||AD||5D|
```

Function `dealCardAndTotal` – The purpose of this function is to help with the totals used in the game logic. A brief explanation of the game is required at this juncture.

In Blackjack, the objective is to get a hand with cards that total as close to 21 as possible, without going over. Each card between 2 and 10 is worth its face value (2 is worth 2, 3 is worth 3, and so on). Each royal card counts for 10, regardless of royalty status (a Jack is worth 10, a Queen is worth 10, and so on). Aces, however, can count as either 1 or 11—depending on whether the higher value would cause the player to exceed 21. For example, if a player holds a King and an Ace, the hand total is 21. However, if the player holds a King, a 5, and an Ace, the hand total is 16.

The `dealCardAndTotal` function should generate a new card in the hand and compute the new total, while taking into account royal cards and aces. Complete the steps as provided.

Define a function called `dealCardAndTotal`. The function does not have a return value. The function has two parameters of type `int`. The first parameter will be used to track the number of aces in the hand; name it `aceCount` or something similar. (This will be important for adjusting the value of the hand if aces are present). The second parameter will be used to track the running total; name it `total` or something similar. It will be important to update both variables within the body of the function, so they will be *pass-by-reference*. Within the body of the function, complete the following:

1. Define and initialize a new random card using the `getRandomCard` function.

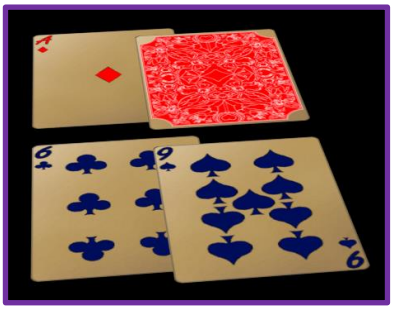
2. Display the new card using the `showCard` function.

3. Complete the following decision set:
    a. If the value of the card is equal to 14 (an ace) then:
        - Increase the ace count variable by 1.
        - Increase the total by 11.
    b. Else if the value of the card is greater than 10 then:
        - Increase the total by 10.
    c. Else:
        - Increase the total by the card's actual value.

4. Create a loop that does the following:
    a. While the ace count is greater than 0 and the hand total is greater than 21:
        - Decrease the ace count by 1.
        - Decrease the total by 10.

Part 3: Game Logic

Using the structures and functions defined in Parts 1 and 2, program the game logic for a single round of blackjack in the `main` function (or other function invoked by `main`). A single round is played as follows:



1. The dealer deals two cards, but only the first card is revealed.

2. The player is dealt two cards, and both cards are shown.

3. The player must decide whether to "hit" (receive another card) or to "stand" (end the turn). The player may receive as many new cards as desired as long as the total of all cards remains less than 21. (Recall that the purpose of the game is to get as close as possible to 21 without going over). If the player's hand exceeds 21, the player "busts", which is an automatic loss.

4. Once the player stands, it is the dealer's turn to play. (If the player busts, the dealer automatically wins and does not take a turn). The rules for the dealer are simple: the dealer must take cards until the total is at least 17. If the dealer's total is 17 or more, they are required to stand. If the dealer's total exceeds 21, the dealer busts and the player wins.

5. If neither the player nor the dealer busts, then the one with a hand closest to 21 wins. If both the player and the dealer have the same total, then a "push" occurs (this is a tie: a no-win, no-loss scenario).

Several sample interactions are shown:

Player busts:

```
Dealer: |3♣||?|

Player: |4♠||3♣|
Player Total: 7

(H)it or (S)tand: h
|9♥|
Player total: 16

(H)it or (S)tand: h
|10♦|
Player total: 26
Player BUSTS!
```

Dealer busts:

```
Dealer: |2♣||?|

Player: |9♠||7♣|
Player Total: 16

(H)it or (S)tand: s

Dealer: |2♣||9♥||2♣||A♦||Q♥|
Dealer total: 24
Dealer BUSTS!
```

Player wins:

```
Dealer: |7♣||?|

Player: |10♦||10♥|
Player Total: 20

(H)it or (S)tand: s

Dealer: |7♣||6♣||6♦|
Dealer total: 19
Player WINS!
```

Player loses:

```
Dealer: |2♣||?|

Player: |9♥||10♦|
Player Total: 19

(H)it or (S)tand: s

Dealer: |2♣||K♦||9♥|
Dealer total: 21
Player LOSE!
```

Push:

```
Dealer: |4♥||?|

Player: |6♥||5♥|
Player Total: 11

(H)it or (S)tand: h
|J♣|
Player total: 21

(H)it or (S)tand: s

Dealer: |4♥||4♦||4♥||9♠|
Dealer total: 21
PUSH!
```

It is recognized that the rules presented above represent an overly simplistic version of the game of blackjack. No form of betting or payouts is introduced. Other rules are overlooked, such as the ability to "double-down" on a bet or "split" a matching pair. As an additional challenge, you may try and incorporate these features into your game. However, it is not required to do so.

Test your blackjack program by executing it several times; ensure that all scenarios (win, loss, push, bust, etc.) are appropriately accounted for. Submit the `.cpp` file online for grading. A grade rubric is provided below.

Grade Breakdown	
A .cpp file containing a simple game of blackjack has been submitted.	5 points
The project contains an enumeration to represent the four standard card suits: hearts, diamonds, clubs, spades.	5 points
The project contains a structure to represent a card with two data fields: one for a card value and one for a card suit.	5 points
The program contains a `getRandomCard` function, which appropriately generates and returns a card structure (with suit and value) at random.	15 points
The program contains a `showCard` function, used to display a decorated card with value and suit upon the console.	10 points
The program contains a `dealCardAndTotal` function, which shows a new card and appropriately handles hand totals, including correct values for royal cards and the 1-or-11 nature of aces; the function appropriately updates totals using *pass-by-reference*.	20 points
The game logic for a single hand of blackjack has been coded and follows at least the simple rules as outlined in this document. Partial points will be awarded for partially completed/correct logic.	40 points
Total	**100 points**