COMP 322
Winter Semester 2022

INSTRUCTOR: DR. CHAD ZAMMAR
chad.zammar@mcgill.ca

Assignment 3: Object Oriented Design.

Due date: 08 April 2022, 11:59 PM.

Before you start:

- Collaboration and research for similar problems on the internet are recommended. However, your submission should reflect individual work and personal effort.
- Some of the topics may not be covered in class due to our limited time. You are encouraged to find answers online. You can also reach out to your instructor or TAs for guidance.
- Please submit your assignment before the due date to avoid penalties or worse risking your assignment being rejected.
- Submit two files called **assignment3-part1.cpp and assignment3-part2.cpp** containing all the declarations of your classes, together with their implementation and a main() function to run your code.

Make sure your code is clear and readable. **Readability of your code as well as the quality of your comments will be graded**.

- No submission by email. Submit your work to mycourse.
- If your code does not compile successfully it will not be graded.
- Be happy when working on your assignment, because a happy software developer has more inspiration than a sad one :).

Part One [70 points]

In this assignment we will tackle the implementation of a very simplified version of the famous BlackJack card game. We will drop all the complicated rules and we will consider that the game is played by one player against the computer.

The simplified rules are like this:

- Each card has a numerical value.
    - Numbered cards are counted at their face value (two counts as 2 points, three, 3 points, and so on)
    - An Ace count as either 1 point or 11 points (whichever suits the player best)
    - Jack, queen and king count 10 points each
- The player will compete against the computer which represents the casino.
- The goal of the player is to try to reach a total point sum of 21 without exceeding it. Whoever exceeds 21 first loses (technically known as busting).
- At the beginning of each round, the player is dealt two open cards and the computer is dealt one open card. The cards are open, meaning that the values are known for both the player and the computer (no hidden cards).
- The player will see the sum of the points from his 2 open cards and decide whether or not to draw an additional card.
- The player may draw one additional card at a time for as long as he likes or until he busts (sum of drawn cards exceeds 21). If he busts, he loses the round.
- When the player decides that he won't draw anymore and he is happy with whatever total amount he got, the computer will draw and open an additional card.
- The computer will keep on drawing additional cards, one at a time as long as the sum of its cards is less or equal 16.

- If the computer busts, the player wins.
- If the computer doesn't bust (total sum is less or equal 21), then the total values are compared between the computer and the player. Whomever has a higher value wins the round.
- If the two totals are the same, no one wins the round (technically called a push).

We will add a twist to the rules to give the casino a higher probability of winning. To do so, we need to keep track of the history of all the rounds to know how many rounds the player has won VS the casino. When we find that the casino is losing more rounds than the player (casino is winning less than 55% of the time), we need to trick the randomness of the cards in order to give the casino a tiny push and maintain a percentage of winning around 55% for the casino.

Here is a list of all the needed **classes** to code the game:

- **Card**: represents a card. Each card has a **rank** and a **type** which can be easily represented via **enums**.
    - Rank is one of the following {ACE = 1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING}
    - Type is one of the following {CLUBS, DIAMONDS, HEARTS, SPADES}
    - Card class must have the following methods:
        - **getValue** that would return the numerical value of a card
        - **displayCard** that would print to the screen the card information. For example **1H** means ace of hearts**, 2D** means two of diamonds**, QS** means queen of spades and so on.
- **Hand**: represents the set of cards that the player or the computer holds. This class should contain a list of cards that can be implemented as an array or a vector or any data structure that you prefer. It should also have the following methods:
    - **add** that would add a card to the hand
    - **clear** that would clear all the cards from a hand (removing them)
    - **getTotal** that would get the total sum of the cards numerical values

- **Deck**: represents the deck of cards and the actions that can be performed on the cards like shuffling and dealing. This class may inherit from the Hand class for convenience but I keep it up to you to decide whether you would inherit or not. Deck should implement the following methods:
    - **Populate** will create a standard deck of 52 cards
    - **shuffle** will shuffle the cards
    - **deal** will deal one card to a hand
- **AbstractPlayer**: represents a generic abstract player that can be a human or the computer. This is an abstract class meaning that it has a pure virtual method. The methods that this class have are:
    - **virtual bool isDrawing() const = 0;** which is a pure virtual method that indicates whether a player wants to draw another card.
    - **isBusted** that returns true if a player has busted (sum of cards exceeds 21).

      Note that AbstractPlayer may inherit from Hand for convenience but this is left totally to you to decide.

- **HumanPlayer**: represents the human player. This class inherits AbstractPlayer and should have the following methods:
    - **isDrawing** implements the inherited method that indicates whether a player wants to draw another card
    - **announce** a method that prints information about whether the player wins, loses or has a push situation.
- **ComputerPlayer**: represents the computer (the casino). This class inherits AbstractPlayer and should have the following method:
    - **isDrawing** implements the inherited method that indicates whether the computer should be drawing another card. Remember that the rules for the computer are different. The computer should keep on drawing an additional card as long as the sum of its cards is less or equal 16. This method should take the percentage of winning into consideration. If the percentage is less than 55%, then you need to implement some logic to make sure that the casino gets better chances of winning.

- **BlackJackGame**: a class representing the overall game. This class must have the following data members:
    - A Deck data member **m_deck**
    - A ComputerPlayer member **m_casino**

      It should also implement the following method:

    - **play** that plays the game of blackjack

Use the following main function to run the game:

```
int main()
{
    cout << "\tWelcome to the Comp322 Blackjack game!" << endl << endl;

    BlackJackGame game;

    // The main loop of the game
    bool playAgain = true;
    char answer = 'y';
    while (playAgain)
    {
        game.play();

        // Check whether the player would like to play another round
        cout << "Would you like another round? (y/n): ";
        cin >> answer;
        cout << endl << endl;
        playAgain = (answer == 'y' ? true : false);
    }

    cout <<"Gave over!";
    return 0;
}
```

The output while playing the game should be something similar to this:

```
Welcome to the Comp322 Blackjack table!
Casino: 8S [8]
Player: 4C 9D [13]
Do you want to draw? (y/n):y
Player: 4C 9D 7C [20]
Do you want to draw? (y/n):n
Casino: 8S 2H [10]
Casino: 8S 2H 9S [19]
Player wins.

Would you like another round? (y/n):y
Casino: 1S [11]
Player: 1C 2H [13]
Do you want to draw? (y/n):y
Player: 1C 2H 10D [13]
Do you want to draw? (y/n):y
Player: 1C 2H 10D KH [23]
Player busts.
Casino wins.

Would you like another round? (y/n):y
Casino: QS [10]
Player: 4C 4H [8]
Do you want to draw? (y/n):y
Player: 4C 4H 3S [11]
Do you want to draw? (y/n):y
Player: 4C 4H 3S JC [21]
Casino: QS 1C [21]
Push: No one wins.

Would you like another round? (y/n):n
Gave over!
```

Grading scheme:

- Proper calculation of the Ace value (1 vs 11) as per the instructions: 3 points
- Rank and type implemented properly as enum: 2 points
- Card::getValue() and Card::displayCard(): 10 points (5 each)
- Hand:: add, clear, getTotal: 12 points (4 each)
- Deck:: populate, shuffle, deal: 12 points (4 each)
- AbstractPlayer::isBusted: 4 points
- AbstractPlayer::isDrawing pure virtual and const: 4 points
- HumanPlayer::isDrawing , annouce : 8 points (4 each)
- ComputerPlayer::isDrawing: 5 points
- BlakJackGame::play(): 6 points
- Keeping track of winning history and maintaining a 55% chance of winning to the casino: 4 points

Part Two [30 points]

Modify your implementation to accommodate for a multi-hand game. This means that the same player can have 1, 2 or 3 hands at the same table. The maximum number of hands that a player can have at the same time is 3.

If the player chooses to have 3 hands, this means that the player will act as if he (or she) were 3 different players. The player can win some hands and lose some others. The player will be drawing for the 3 different hands one after the other.

Clone your code from part 1 and modify it to accommodate the new requirements. Please make sure to submit the versions separately.

Same rules apply as before. Modify the output to reflect the new reality.