codeJack: The Blackjack Simulator 2.0

Submission Details

Due: `Oct 21, 2024 11:59 PM CST`

- *The due date for different sections may vary. Please verify the exact due date with your instructor to ensure timely submission.*

Total Possible Points: 100 points

Documentation Header Reminder

Before you start your assignment, you will need to add documentation similar to what we demonstrated in the first few lectures.

```
// Programmer: San Yeung
// Date: 9/4/1002
// File: fahr2celc.cpp
// Assignment: HW2
// Purpose: this file contains the main function of the program which
//   will input Fahrenheit temps from the user, then convert and output
//   the corresponding Celcius temperature.
```

Function Documentation Reminder

Having proper documentation for C++ functions is important. Make sure that each function includes the following:

- General Description: A brief description of what the function does.
- Precondition: Describe any requirements that must be met before calling the function, e.g., function parameter conditions.
- Postcondition: Explain what the function accomplishes as the aftermath and report any significant changes it makes to persistent variables or data.

Multiple File Compilation

For this assignment (and onward), you will submit **multiple** C++ compilable files containing a program written in C++. Name your files a meaningful name and give it a proper extension (.h, .hpp, .cpp).

To compile multiple files in the same directory, simply execute "fg++ *.cpp -o your_executable_file_name" on the CS Linux virtual machine's command line environment. The syntax *.cpp will grab all the .cpp files to be compiled (header files need NOT to be included). Remember, fg++ is just a shortcut for invoking the g++ command with some useful flags turned on, i.e. g++ -Wall -W -s -pedantic-errors, to ensure the quality of the program follows the C++ standard.

Objective

The objective of this assignment is to reinforce key C++ programming concepts, including the use of structs for structured data management, operator overloading for struct operations, and the management of arrays.

Simulation Specification

Your task is to continue building on the previous Blackjack simulator. This revised simulation will still involve a single player (the user) playing against the dealer (the computer).

1. **Starting Money**
    a. Upon starting the simulation, the program will randomly assign a starting balance to the player AFTER prompting the player to enter their name. This balance is to be used for placing bets within the game. The generated random balance should **always** include two decimal places to reflect cents, with a range within $12.00 to $7000.00, inclusively.
2. **Game Setup**
    a. A standard card deck of 52 cards is divided into four suits (hearts, diamonds, clubs, and spades), each with 13 cards: Ace through 10, and the three face cards, which are the Jack, Queen, and King. **This new version of the Blackjack game no longer assumes each card has an infinite number of copies.**

i. The values of the cards are as follows: 2-10 are valued at their face value, Jack/Queen/King are valued at 10, and Aces can be either 1 or 11.

ii. You **must** define a struct named Card that represents a playing card in the game. This struct should include at least two fields: one for the card's value and another for its suit. You are permitted to add additional information to include in your struct.

b. **Prompt for Wager:** At the beginning of each round, prompt the player to enter the amount of money they would like to wager on the upcoming hand. Ensure the player input is a valid amount (not more than their current balance).

c. **Hand Initialization:** Each round of the new game begins by the program dealing two **unique** random cards to the player and the dealer. Each card's value can be *randomly* generated in the range [1, 13]. While the suit of a card doesn't affect its value in this Blackjack simulation, you are still required to *randomly* assign one of the four suits to each card drawn.

i. Since each card in the deck is unique and can only be dealt once, **your program must check to ensure that a card (with its specific value and suit) has not already been dealt in the current round**, whether in the player's or dealer's hand. It's assumed that the deck is reshuffled at the conclusion of each round.

ii. You **must** use arrays to store the cards dealt to the player and the dealer during each round. In order to ensure these arrays are capable of holding the maximum number of cards a hand might have in Blackjack, set the array capacity to 20.

iii. An example output right after dealing might look like this:

```
Player's Hand: 10 of Diamonds, 3 of Hearts
Dealer's Hand: King of Spades, [Hidden Card]
```

d. **Check for Blackjack:** After the initial dealing, check if either has a hand of Blackjack. A Blackjack is the highest hand and consists of an Ace and a 10-valued card (10, Jack,

Queen, or King). This hand is superior to other hands, including those that total 21 but with more than two cards.

i. If only the player has Blackjack, they win.
ii. If only the dealer has Blackjack, the player loses.
iii. If both have Blackjack, it's a tie.

3. Player's Turn

a. **Basic Actions:** After the player's hand and the dealer's up card are dealt, proceed to allow the player to choose "Hit" (get another card) or "Stand" (end their turn). The player can hit multiple times if desired as long as their hand total does not exceed 21; otherwise, the hand is considered to be a bust and the game ends.**The Double Down Rule:** Besides the standard "Hit" option, the player can also choose a special strategic choice known as "Double Down" which allows the player to **double** their wager on their hand in exchange for receiving **ONLY one additional** card. After receiving this one additional card, the player is NOT ALLOWED to take any further actions; they must stand, regardless of the total of their hand. Note that the player can only choose this option if they have enough balance to double down.

i. Note that this option is **mutually exclusive** with the regular "Hit" option. This means that in a round where the player opts to double down after receiving the initial hand, they forfeit the opportunity to "Hit" normally. A player must decide between taking a regular "Hit" to possibly receive multiple additional cards without altering their wager, or "Double Down" for a single card with the chance to double their potential winnings.

4. Dealer's Turn

a. First, reveal the dealer's hidden card. The dealer **must** hit if their total is 16 or less and stand on 17 or higher. If the dealer busts, the player gets the win.

5. **Determining the Winner**
    a. If neither the player nor the dealer has busted, compare the total values of the hands. The winner is the one who has the total point closest to 21.
    b. In the event of a tie (when both hand values are equal), neither the player nor the dealer is declared the winner. In addition, the player's wager needs to be returned to them.
6. **Continuous Play and Exit Condition**
    a. As each round concludes, the simulation should then prompt the player to decide whether they wish to play another round. If the player decides not to play, then the program should end gracefully after displaying a summary of the player's performance.
    b. The game should also automatically terminate if the player's balance depletes, signifying that they can no longer place any wagers.

Function Requirements

The following specific, named functions are required. However, the choice of return type and parameters is left to your discretion. It will be your responsibility to decide on the most suitable function signatures based on your own implementation and design choices. So, be creative and effective in your solutions!

1. `generateRandomCard()`: Generate a random card value within an acceptable range.
    a. **Suggested Signature:** `int generateRandomCard(const int min, const int max);`
2. `isBlackjack()`: Determine whether a hand is a Blackjack.
3. `printHand()`: Display a player's or dealer's hand of cards in a sorted order, first by suit and then by rank.
    a. **Suggested Signature:** `void printHand(Card hand[], const int handSize);`
    b. **Sorting Requirements:** Before displaying the hand, all cards should be sorted first by suit in the order of Diamonds, Hearts, Clubs, and Spades, and then by rank from Ace through King.
    c. **This function should be called whenever a hand is displayed.**

d. **Example Output:**

```
Player's Hand: Ace of Diamonds, 2 of Diamonds, Ace of
Hearts, King of Spades
```

4. sortElements(): A generic **template function** designed to sort an array of elements in ascending order based on user-defined comparison criteria. You must implement your own sorting algorithm (bubble sort, insertion sort, selection sort, etc.), i.e., using pre-existing sort functions from C++ libraries, like <algorithm>, is disallowed.
   a. **Suggested Signature:** void sortElements(Element array[], const int arraySize);
   b. **This function should be called by the** printHand() **function whenever a hand is displayed.**
5. Relational Operator Overloading (one of <, <=, >, >=): You are required to overload **at least** one relational operator to be used in the implementation of your sorting algorithm.
6. Insertion Operator Overloading: You are required to overload the insertion (output stream) operator (<<) for displaying a card object. Whenever a card needs to be displayed within the program, this overloaded operator **must** be utilized to output the card's details to the console.
7. updatePlayerBalance(): Update the player's balance after each round, taking into account the outcome of the game and whether they followed the strategic advice provided.
   a. **Payout Schemes:** The payout for a player winning a hand against the dealer is 1:1, meaning the player wins an amount equal to their wager. For example, if the player wagers $20 and wins the hand, they would receive their original wager back **plus** an additional $20 in winnings. The payout for winning a hand with Blackjack is 1.5:1.
8. displayOutcome(): Display a message indicating the game outcome of the current round and any changes to the player's balance. An example output might look something like this:

```
Congratulations! You hit Blackjack!
```

```
Balance Update: +$30
Your new balance is $1030.
```

9. displayGameSummary(): Provide a detailed summary of the player's entire game session upon exiting the program, including **total wins, losses, ties, final balance,** and any other relevant statistics to offer additional insights into their performance. An example output might look something like this:

```
=== Game Summary ===
Total Rounds Played: 15
Total Wins (Regular): 5
Total Wins (Blackjack): 2
Total Losses (Regular): 4
Total Losses (Blackjack): 2
Total Ties (Regular): 2
Total Ties (Blackjack): 0
Final Balance: $1030
Net Gain/Loss: +$30

Thank you for playing CodeJack: The Blackjack Simulator!
We hope to see you again soon.
```

[BONUS Functions 10 Points]

10. adviseOptimalActionOnLuck() (**must** be implemented as a template function): This function offers strategic recommendations to the player if the probability threshold is met; otherwise, it suggests a random action.
    a. **Suggested Signature:**

```
template <typename T>
string adviseOptimalOptionOnLuck(const T probability_threshold,
  const int playerTotal, const int dealerUpCard);
```

    b. Parameter probability_threshold: A *randomly generated* probability threshold for the chance of a favorable outcome that, if met, recommends the player the optimal game option to take based on the hands in the current game. This parameter can be an integer or floating-point value.

c. **When to Call this Function:** This function should be invoked right after the initial cards have been dealt to both the player and the dealer (and after confirming the *absence* of Blackjack in the player's hand).
d. **Probability Threshold Met (Strategic Recommendation):** If the generated chance occurrence inside the function meets or exceeds the threshold, the function assesses the player's and dealer's hands to offer a strategic move based on the following:
    i. If playerTotal is 11 or less: Always recommend "Hit".
    ii. If playerTotal is between 12 and 16:
        1. If dealerUpCard is 7 or higher: Recommend "Hit" (since the dealer has a strong hand).
        2. Otherwise, recommend "Stand" in the hope that the dealer will bust.
    iii. If playerTotal is 17 or higher: Recommend "Stand" (since the chance of busting is high).
    iv. *Special Case:* If playerTotal is between 9 and 11, then recommend "Double Down" since the probability of reaching 19 to 21 is exceptionally high.
    v. *Strategic Prioritization with an Ace in Hand:*
        1. When the player's hand includes an Ace, the "Stand" recommendation should take priority over the other options since the reward outweighs the risk. Following the priority to "Stand", "Double Down" (i.e., move #iv) should be preferred over "Hit". Following that, move #ii should be preferred over move #i.
        2. **Example:** Imagine the player is dealt an initial hand consisting of an Ace and a 6. This gives it a flexible score of either 7 or 17. Since the condition satisfies both move #i and move #iii, recommending the "Stand" option is the correct one due to the strategic prioritization specified above.
e. **Probability Threshold Not Met (Random Recommendation):** If the probability threshold is not met, the function should randomly select "Hit", "Stand", or "Double Down" to suggest as the action.

f. **Example:** Suppose the initial cards dealt to the player are 8 of Spades and 2 of Hearts, and the program randomly generates a probability threshold for this round set to be 60% and calls this function. Inside the function, the randomly generated "chance occurrence" – let's say it's 65% – exceeds the threshold. Assuming that the dealer's up card is 9 of Clubs, the function assesses the player's total to be between 9 and 11 and recommends "Double Down".

Example Expected Output Format

The following example is provided to illustrate the overall workflow and output format of the program. Please be aware that the actual outputs from your program will vary.

IMPORTANT!!: Please follow exactly the specific input and action sequence as doing so will simplify the grading process.

- Ensure all inputs during each round are taken in the specified order: **wager input**, **choice of action(s)**, and **decision to replay**.
- Failure to comply may result in the rejection of your submission.

```
TBD
```

Important Notes

- For this assignment, use 370 to seed the random number generation process.
- If the parameters and return type of a function is not specified, then it is your responsibility to determine the most appropriate function signatures for them.
- You may create additional custom functions for this assignment as you see fit.
- You may use functions from other C++ libraries if 1) they are explicitly stated in the assignment or 2) the usage of the library functions has been thoroughly introduced in class. Otherwise, the usage of other functions is prohibited without the permission of your instructor.

- DO include input/range validations as long as they are appropriate. While input data type validation is not required for this assignment, it is good practice to validate the data values for range.
- Use modular programming! Break down the assignment into smaller functions that perform specific tasks. This will help make your code easier to read, test, and maintain.
- Thoroughly test each function before moving on to the implementation of the next function! This is referred to as the iterative/test-based development approach. It emphasizes on testing and refining each function before moving onto the next one. This helps to identify and fix errors early on, and ensure that the final program is reliable and robust.
- Test the program with different inputs. This will help you ensure that the program is functioning as expected and will also help you identify any bugs or errors.
- Read the error messages carefully if the program is not working as expected. Understanding the error messages will help you to identify and fix the problem.
- Don't forget to use an adequate amount of comments to explain your code and make it easy to understand.
- You must use proper indentation (two whitespaces) to make the code more readable and easy to understand.
- Don't forget to have fun and be creative with your program design and implementation! Programming can be both challenging and interesting at the same time!
- Finally, don't hesitate to ask for help from the instructor or TA if you are having trouble with the assignment.

Grading Criteria:

1. **Problem-Solving and Logical Accuracy** (25%)
    - The solution effectively addresses the problem.
    - Logic used in the code is sound and free of major errors.
    - Program behaves as expected under different conditions.
2. **Completeness** (20%)
    - All components of the assignment are addressed.
    - Edge cases or unique scenarios are considered and handled appropriately.

- The program handles invalid inputs gracefully.
- The program meets basic function requirements and compiles successfully without any major errors or warnings.
- **Non-Compiling Work Deduction**
  - If your submission does not compile, an initial deduction of 50% of the total points will be applied.
  - Students whose submissions fail to compile due to **minor issues** are eligible to contact their assigned grader for feedback and may resubmit corrected work.

3. **Optimization and Efficiency** (20%)
   - The solution takes advantage of efficient methods, algorithms, and data structures where applicable.
   - Redundant or unnecessary code or computations are minimized.
4. **Syntax and Structure** (20%)
   - Code follows a consistent format and adheres to C++ standards.
   - Proper indentation is used to enhance code readability.
   - Use of appropriate C++ conventions, functions, and constructs.
   - Demonstrates the ability to organize code across multiple files for modularity.
5. **Clarity and Understandability** (10%)
   - Variables and functions are appropriately named and clearly defined.
   - Logical flow is evident and coherent from start to finish.
6. **Documentation and Comments** (5%)
   - Important steps are commented for clarity.
   - Complex sections of code have accompanying explanations.
   - Each function is accompanied by clear documentation.