Card Game Function Report

This report outlines the purpose and functionality of each significant function in your C++ card game, providing a clear, step-by-step explanation.

Global Variables and Constants

Before diving into the functions, it's important to understand the global variables and constants that many functions interact with:

- SUITS, FACES, CARDS: Defines the standard number of suits (4), faces (13), and total cards (52) in a deck.
- start_card, end_card, total_cards: Used to manage the dealing of cards from the shuffled deck. total_cards tracks how many cards are left.
- computer_cards, player_cards: std::vector<std::vector<int>> storing the current hand for the computer and player, respectively. Each inner vector {suit, face} represents a card.
- wDeck[SUITS][FACES]: A 2D array representing the full deck. wDeck[row][column] = card_number indicates which card (1-52) is at that suit/face position.
- wSuit[SUITS], wFace[FACES]: Arrays of character pointers for displaying suit and face names.
- player_score, computer_score: Integers tracking the scores of the player and computer.
- gameFont, cardTextures, cardBackTexture: SFML-related variables for loading fonts and card images.
- GameState currentGameState: An enum to manage the different states of the game (PLAYING, EVALUATING, GAME_OVER).
- playerMessage, computerMessage, statusMessage: std::string variables for displaying messages to the user in the UI.
- evaluationClock, evaluationDisplayTime: SFML clock for managing the duration of the evaluation phase.
- patternButtons: A std::vector of Button objects for the player to select a hand pattern.

Function Explanations

1. shuffle(int wDeck[][FACES])

- **Purpose:** Randomly shuffles the wDeck array by assigning unique card numbers (1 to 52) to random suit and face combinations.
- **Steps:**
  - Initializes a loop that runs CARDS (52) times, representing each card in the

deck.

    - Inside the loop, it enters a do-while loop:
        - Generates random row (suit) and column (face) indices.
        - Checks if the generated wDeck[row][column] position is already occupied (i.e., not 0).
        - The do-while loop continues until an unoccupied spot is found.
    - Once an empty spot is found, it assigns the current card number to wDeck[row][column].

2. cards_adder(std::vector<std::vector<int>> &current_hand, int wDeck[][FACES], int start, int end)

- **Purpose:** Deals a specified range of cards from the wDeck into a current_hand vector.
- **Steps:**
    - Iterates through card numbers from start to end.
    - For each card number, it searches through the entire wDeck to find its {suit, face} coordinates.
    - Once found, it adds this {suit, face} pair as a new inner std::vector<int> to the current_hand.

3. discarder(std::vector<std::vector<int>> &current_hand, std::string check_pattern)

- **Purpose:** Removes specific cards from a current_hand based on the declared check_pattern. This is used when certain patterns (like one pair, two pair, three of a kind, four of a kind, high card) involve discarding only a portion of the hand.
- **Steps (General Logic):**
    - Initializes an empty discard_vector to store cards that need to be removed.
    - Uses a series of if statements to check the check_pattern:
        - **"highcard":** Adds the first card (current_hand[0]) to discard_vector. (Note: this implementation seems to discard just the first card for high card, not necessarily the actual high card, and might need adjustment for full high card game logic).
        - **"onepair":** Iterates through the hand to find two cards with the same face value. Adds both cards of the pair to discard_vector.
        - **"twopair":** Calculates face frequency. If two pairs are found, it identifies the ranks of those pairs and adds all cards matching those ranks to discard_vector.
        - **"fourofakind":** Calculates face frequency. If a face appears 4 times, all four cards of that rank are added to discard_vector.

        - **"threeofakind":** Calculates face frequency. If a face appears 3 times, all three cards of that rank are added to discard_vector.
    - After identifying cards to discard, it creates a to_remove vector.
    - It then iterates through discard_vector and compares each card's face value to cards in current_hand. If a match is found, it's added to to_remove.
    - Finally, it uses std::remove and std::erase to remove all cards listed in to_remove from current_hand. (Note: The current implementation of populating to_remove seems to be discarding all cards with the same *face value* as a discarded card, which might be overly aggressive for some patterns. For example, if a "onepair" of 7s is discarded, it might remove all 7s in the hand, even if they weren't part of the initial discard criteria. This area could benefit from refinement for precise discarding.)

4. high_card_check(const std::vector<std::vector<int>> &current_hand)

- **Purpose:** Checks if a hand qualifies as a "high card" hand (meaning no other recognized poker pattern is present).
- **Steps:**
    - Calls one_pair_check(current_hand).
    - If one_pair_check returns true (meaning there *is* at least one pair), high_card_check returns 0 (false).
    - Otherwise (if no pair is found), it returns 1 (true).
    - **Note:** This function only checks for the absence of "one pair." A more robust high_card_check would ideally verify the absence of *all* higher-ranking poker hands.

5. one_pair_check(const std::vector<std::vector<int>> &current_hand)

- **Purpose:** Determines if the current_hand contains at least one pair of cards (two cards of the same face value).
- **Steps:**
    - Uses nested loops to compare the face value of every card with every other card in the hand.
    - If two cards with the same face value are found, it immediately returns 1 (true).
    - If no pair is found after checking all combinations, it returns 0 (false).

6. two_pair_check(const std::vector<std::vector<int>> &current_hand)

- **Purpose:** Determines if the current_hand contains exactly two pairs of cards.
- **Steps:**
    - Initializes a freq array (size 13) to count the occurrences of each face value.
    - Iterates through the current_hand and increments the frequency for each

card's face value.
    - Counts how many face values have a frequency of 2 (indicating a pair).
    - If pairs is exactly 2, it returns 1 (true).
    - Otherwise, it returns 0 (false).

7. three_face_check(const std::vector<std::vector<int>> &current_hand)

- **Purpose:** Determines if the current_hand contains three cards of the same face value (three of a kind).
- **Steps:**
    - Initializes a freq array (size 13) to count face value occurrences.
    - Iterates through the current_hand and updates frequencies.
    - Iterates through the freq array. If any face value has a frequency of 3, it returns 1 (true).
    - Otherwise, it returns 0 (false).

8. four_face_check(const std::vector<std::vector<int>> &current_hand)

- **Purpose:** Determines if the current_hand contains four cards of the same face value (four of a kind).
- **Steps:**
    - Similar to three_face_check, it uses a freq array to count face occurrences.
    - Checks if any face value has a frequency of 4. If so, returns 1 (true).
    - Otherwise, returns 0 (false).

9. flush_check(const std::vector<std::vector<int>> &current_hand)

- **Purpose:** Determines if all cards in the current_hand are of the same suit (a flush).
- **Steps:**
    - Takes the suit of the first card in the hand as a reference.
    - Iterates through the rest of the cards in the hand.
    - If any card's suit does not match the reference suit, it returns 0 (false).
    - If all cards have the same suit, it returns 1 (true).

10. straight_check(const std::vector<std::vector<int>> &current_hand)

- **Purpose:** Determines if the current_hand contains five cards in sequential order of face values (a straight).
- **Steps:**
    - Extracts the face values of the five cards into a temporary face_arr.
    - Sorts face_arr in ascending order.
    - Iterates through the sorted face_arr, checking if the difference between consecutive card faces is exactly 1.

  - Counts how many such consecutive differences are found (sum).
  - If sum is 4 (meaning all five cards are sequential), it returns 1 (true).
  - Otherwise, returns 0 (false).

11. straight_flush_check(const std::vector<std::vector<int>> &current_hand)

- **Purpose:** Determines if the current_hand is both a straight and a flush.
- **Steps:**
  - Calls straight_check(current_hand) and flush_check(current_hand).
  - If both return 1 (true), it means the hand is a straight flush, and it returns 1 (true).
  - Otherwise, returns 0 (false).

12. housefull_check(const std::vector<std::vector<int>> &current_hand)

- **Purpose:** Determines if the current_hand contains a full house (three of a kind and a pair).
- **Steps:**
  - First, it calls three_face_check(current_hand). If no three of a kind is present, it immediately returns 0 (false).
  - If a three of a kind exists:
    - Calculates face frequencies.
    - Identifies the rank of the three of a kind (stored in key).
    - Counts how many *other* ranks (not the key rank) have a frequency of 2 (indicating a pair).
    - If exactly one such pair is found, it returns 1 (true).
    - Otherwise, returns 0 (false).

13. royal_flush(const std::vector<std::vector<int>> &current_hand)

- **Purpose:** Determines if the current_hand is a royal flush (a straight flush consisting of Ten, Jack, Queen, King, and Ace of the same suit).
- **Steps:**
  - First, it calls flush_check(current_hand). If not a flush, returns 0 (false).
  - If it is a flush:
    - Creates five_faces vector from the current hand's face values.
    - Defines majestic_flush array with the face values for a royal flush (0 for Ace, 10 for Ten, 11 for Jack, 12 for Queen, 13 for King). (Note: Ace is typically high in a royal flush, but its value is 0. This implementation correctly uses 0).
    - Compares the face values in five_faces with majestic_flush.
    - Counts how many matches (matches) are found.
    - If matches is 5, it means all royal flush cards are present, and it returns 1

    (true).
  - Clears five_faces and returns 0 (false) otherwise.

14. computer_hand_evaluater(std::vector<std::vector<int>> &current_hand, int &score, int wDeck[][FACES])

- **Purpose:** Evaluates the computer's hand, assigns scores based on the best poker hand found, discards cards (for some patterns), and deals new ones. It also sets the computerMessage for UI display.
- **Steps:**
  - Clears any previous computerMessage.
  - Checks if total_cards is zero or less; if so, it returns, preventing further dealing.
  - It checks for poker hands in descending order of rank (Royal Flush to High Card).
  - **For each pattern:**
    - If the hand matches a pattern (e.g., royal_flush(current_hand)):
      - Sets computerMessage to indicate the detected pattern.
      - Adds a predefined score to score.
      - **If the pattern is a full-hand replacement (e.g., Royal Flush, Straight Flush, Flush, Straight, Full House):**
        - Clears current_hand.
        - Decrements total_cards by 5.
        - Calls cards_adder to deal 5 new cards to current_hand from the deck, updating start_card.
      - **If the pattern is a partial replacement (e.g., Four of a Kind, Three of a Kind, Two Pair, One Pair, High Card):**
        - Calls discarder to remove the relevant cards.
        - Decrements total_cards by the number of discarded cards.
        - Calls cards_adder to deal new cards to fill the hand, updating start_card.
  - If no specific pattern is found, it defaults to a "High Card" evaluation:
    - Calculates the highest face value in the hand.
    - Adds this face value to score.
    - Discards one card using discarder("highcard").
    - Deals one new card.

15. player_hand_evaluater(std::vector<std::vector<int>> &current_hand, int &score, int wDeck[][FACES], std::string pattern)

- **Purpose:** Evaluates the player's hand based on the pattern declared by the player, assigns scores, discards cards, and deals new ones.

- **Steps:**
  - Checks if total_cards is zero or less; if so, returns.
  - Uses a series of else if statements to match the pattern declared by the player with the actual hand check functions.
  - **For each declared pattern:**
    - If the current_hand *actually* matches the declared pattern (e.g., pattern == "royalflush" && royal_flush(current_hand)):
      - Adds a predefined score to score.
      - **If the pattern is a full-hand replacement (e.g., Royal Flush, Straight Flush, Flush, Straight, Full House):**
        - Clears current_hand.
        - Decrements total_cards by 5.
        - Calls cards_adder to deal 5 new cards to current_hand from the deck, updating start_card.
      - **If the pattern is a partial replacement (e.g., Four of a Kind, Three of a Kind, Two Pair, One Pair, High Card):**
        - Calls discarder to remove the relevant cards based on the pattern string.
        - Decrements total_cards by the number of discarded cards.
        - Calls cards_adder to deal new cards to fill the hand, updating start_card.
  - If the declared pattern does not match the actual hand, the score is not increased, and cards are not discarded/dealt (this behavior determines if the player's declaration was correct).

SFML Helper Functions

16. struct Button (and its methods)

- **Purpose:** Defines a structure to easily create and manage interactive buttons in the SFML window.
- **Members:**
  - sf::RectangleShape shape: The visual rectangular shape of the button.
  - sf::Text text: The text displayed on the button.
  - std::string patternName: A string to identify the poker pattern associated with the button (e.g., "royalflush").
- **Constructor:**
  - Sets the button's position, size, fill color, outline, and text properties.
  - Centers the text within the button's shape.
- **draw(sf::RenderWindow &window):**

    - Draws the button's shape and text onto the SFML window.
- **isClicked(const sf::Vector2i &mousePos) const:**
    - Checks if a given mousePos (mouse click coordinates) is within the global bounds of the button's shape. Returns true if clicked, false otherwise.

17. std::string getCardFilename(int suit, int face)

- **Purpose:** Generates the file path for a card texture based on its suit and face values.
- **Steps:**
    - Converts the integer suit and face into their corresponding string names using wSuit and wFace.
    - Converts these strings to lowercase.
    - Constructs a filename string in the format "cards/face_of_suit.png" (e.g., "cards/ace_of_hearts.png").

18. bool loadResources()

- **Purpose:** Loads all necessary game assets (font, card back texture, and individual card textures) into memory.
- **Steps:**
    - Attempts to load font/Arial.ttf into gameFont. Returns false and prints an error if it fails.
    - Attempts to load cards/card_back.png into cardBackTexture. Returns false and prints an error if it fails.
    - Uses nested loops to iterate through all possible suit and face combinations (0-3 for suits, 0-12 for faces).
    - For each combination:
        - Creates an sf::Texture object.
        - Calls getCardFilename to get the path for the specific card image.
        - Attempts to load the image into the sf::Texture. Prints an error if it fails (but continues loading other textures).
        - Stores the loaded texture in the cardTextures map, using a std::pair<int, int> (suit, face) as the key.
    - Returns true if all initial resource loads (font, card back) succeed.

19. void createButtons()

- **Purpose:** Initializes and populates the patternButtons vector with Button objects for each poker hand pattern the player can declare.
- **Steps:**
    - Defines starting coordinates, width, height, and spacing for the buttons.
    - Defines a std::vector of std::pair<std::string, std::string> where each pair

contains the internal pattern name (e.g., "highcard") and the display text (e.g., "High Card").

  - Iterates through this list of patterns:
    - Calculates the x and y position for each button based on its index and the defined layout parameters.
    - Constructs a Button object using emplace_back, passing the calculated position, dimensions, pattern names, and the loaded gameFont.

20. void drawHand(sf::RenderWindow &window, const std::vector<std::vector<int>> &hand, float yPos, bool hidden)

- **Purpose:** Draws a hand of cards (either player's or computer's) onto the SFML window.
- **Steps:**
  - Calculates the starting X position to center the hand horizontally on the window.
  - Iterates through each card in the hand vector.
  - For each card:
    - Creates an sf::Sprite.
    - If hidden is true (e.g., for the computer's hand before evaluation), it sets the sprite's texture to cardBackTexture.
    - Otherwise (for player's hand or during evaluation phase), it looks up the correct card texture from cardTextures using the card's suit and face. If not found (error case), it defaults to the card back.
    - Sets the scale of the card sprite to fit desired dimensions (80x120 pixels).
    - Sets the position of the card sprite.
    - Draws the cardSprite onto the window.

21. void drawText(sf::RenderWindow &window, const std::string &str, sf::Vector2f pos, int size, sf::Color color = sf::Color::White)

- **Purpose:** A utility function to draw text at a specific position on the SFML window.
- **Steps:**
  - Creates an sf::Text object with the given str (string), gameFont, and size.
  - Sets the text's position and fill color.
  - Draws the sf::Text object onto the window.

22. void drawCenteredText(sf::RenderWindow &window, const std::string &str, float yPos, int size, sf::Color color = sf::Color::White)

- **Purpose:** A utility function to draw text horizontally centered on the SFML window at a given Y position.

- **Steps:**
  - Creates an sf::Text object.
  - Calculates the local bounds of the text.
  - Sets the text's origin to its center, which is crucial for accurate centering.
  - Sets the text's position by using window.getSize().x / 2.0f for horizontal centering and the provided yPos.
  - Sets the fill color.
  - Draws the sf::Text object onto the window.

23. int main()

- **Purpose:** The entry point of the program; sets up the game window, initializes game state, runs the main game loop, and handles events and rendering.
- **Steps:**
  - **Initialization:**
    - Creates an sf::RenderWindow.
    - Sets the framerate limit.
    - Calls loadResources() to load assets; exits if loading fails.
    - Calls createButtons() to set up the player's action buttons.
    - Seeds the random number generator (srand(time(NULL))).
    - Calls shuffle(wDeck) to randomize the deck.
    - Calls cards_adder to deal initial hands to the player and computer.
  - **Main Game Loop (while (window.isOpen())):**
    - **Event Handling (while (window.pollEvent(event))):**
      - Checks for sf::Event::Closed (window close button) and closes the window if detected.
      - If the currentGameState is PLAYING and a mouse button is pressed:
        - Checks if the left mouse button was pressed.
        - Gets the mouse cursor position.
        - Iterates through patternButtons:
          - If a button is clicked:
            - Saves the player_score before evaluation.
            - Calls player_hand_evaluater with the player's declared pattern.
            - Updates playerMessage based on whether the score changed (indicating a correct declaration).
            - Calls computer_hand_evaluater to evaluate the computer's hand.
            - Checks if total_cards is low; if so, sets currentGameState to GAME_OVER.

            - Otherwise, sets currentGameState to EVALUATING and restarts the evaluationClock.
            - Breaks out of the button loop after a click.
    - **Game State Logic (if (currentGameState == EVALUATING)):**
        - If in EVALUATING state, checks if evaluationDisplayTime has passed.
        - If the time is up, switches currentGameState back to PLAYING and clears UI messages.
    - **Rendering:**
        - Clears the window with a green background.
        - Draws "Computer's Hand" title (centered).
        - Draws computer_cards, hidden unless in GAME_OVER or EVALUATING state.
        - Draws "Player's Hand" title (centered).
        - Draws player_cards (always visible).
        - Draws player score, computer score, and cards left.
        - If currentGameState is PLAYING, draws all pattern buttons.
        - Draws playerMessage and computerMessage.
        - Draws the statusMessage (centered), with red color if GAME_OVER.
        - If currentGameState is GAME_OVER, determines and draws the final winner message (centered, yellow).
        - Displays all drawn elements on the window (window.display()).
- Returns 0 when the game loop ends (window closed).