Project 1

Title

Blackjack

Course

CIS -17C

Section

47469

Due Date

November 30, 2025

Author

Jade Thong

Introduction

This project is a Blackjack Game Simulation developed in C++. The game was chosen because I wanted to choose a game that was well-known and that most people know how to play. Card games are widely known throughout the world, especially in Casinos, therefore, I picked Blackjack. The rules of Blackjack are relatively simple—get closer to 21 than the dealer without exceeding it—implementing the full range of player options, including Hit, Stand, Double Down, and Split.

The development of this simulation took approximately one month to complete and resulted in a codebase spanning 761 lines of code, including comments and formatting. The program structure relies on three classes/structs—Card, Hand, and Player. Crucially, the efficiency and logic of the game are powered by a strategic use of STL containers. The complete source code for the project is hosted on GitHub, ensuring transparency and providing a reference for the implementation details. The repository for this Blackjack Simulation is located at: https://github.com/jadethong/CIS17C_Blackjack_Project.

Approach to Development

The project served as a practical exercise in applying several fundamental and advanced Data Structures concepts. The simulation required implementing custom structs and operator overloading to manage complex data types; specifically, the Card, Hand, and Player structs include overloaded operators (like operator<, operator==, and operator<<) to allow them to be used effectively with STL containers for sorting, uniqueness checks, and easy output. The program also made strategic use of pointers and memory management, employing Player

pointers within the std::queue to manage the turn order efficiently without the overhead of copying large player structures. Furthermore, lambda expressions and STL algorithms were extensively utilized; std::for_each and std::remove_if were applied for iteration and conditional list modification, while accumulation logic was handled by a custom lambda passed to std::accumulate. Finally, the shufDk() function utilizes the modern random number generation capabilities to implement a robust, time-seeded, list-based shuffling algorithm.

The development process for this project relied on a local file saving and backup strategy rather than a formal version control system like Git. All source files, primarily the main C++ code, were managed by creating sequential saved files or folder backups at critical stages of development. This manual approach ensured that working versions of the code could be preserved before attempting major refactors or adding complex new features like the splitting or doubling down logic. While this method was sufficient, it inherently lacks the detailed change history and branching/merging tools that are standard in professional game development. The final, definitive version of the project's source code, as described in this documentation, is hosted at the following URL: https://github.com/jadethong/CIS17C_Blackjack_Project.

Game Rules

Blackjack is a card game played between players and a dealer, where the objective is to have a hand score closer to 21 than the dealer without exceeding it (busting). The game begins with players placing their bets, followed by the initial deal of two cards to each player and two to the dealer (one face-up, one face-down, called the hole card). Face cards (King, Queen, Jack) and 10s are valued at 10, number cards are valued at their face amount, and Aces can be counted as

11 or 1. Players take turns choosing to Hit (take an additional card), Stand (keep the current total), Double Down (double the bet for one more card only), or Split (divide two cards of the same rank into two separate hands). The dealer must then play their hand according to fixed rules, typically hitting until their total is 17 or greater. Finally, the hands are settled: a Natural Blackjack (21 on the first two cards) pays 3:2, a win pays 1:1, and a score tie is a Push where the bet is returned.

Description of Code

The codebase is organized into a cohesive structure built around three fundamental structs and a comprehensive set of functions that utilize Standard Template Library (STL) containers to manage the game state. The file begins with necessary system includes, heavily featuring libraries like <list>, <map>, <stack>, and <queue> to enable the robust use of STL containers throughout the program. Following the system includes are Global Constants and Data Structures, which define the primary data models: the Card, Hand, and Player structs. This section also initializes critical read-only containers, such as the std::ma0, CRDVALS for rank-to-value mapping) and the std::list SUITS. Immediately after, Global Containers for Game State are declared, which are the dynamic STL objects that control the game flow: deck(std::list), disPile(std::stack), and playQue(std::queue). The code then proceeds through Function Prototypes and their Function Definitions, grouped logically for scoring, deck management (creation, shuffling, dealing), display utilities, and the core game logic (handling bets, player turns, and settlement), concluding with the primary execution functions, main()} and runGame().

The simulation’s data integrity is maintained by three custom structs. The Card struct represents the single unit of play, storing its rank (e.g., "A", "K"), suit (e.g., "♥"), and value (e.g., 11 or 10). To allow Card objects to work seamlessly within STL containers and be easily outputted, it features overloaded operators for equality operator==, sorting operator< and streaming operator<<. The Hand struct acts as a container for a player's cards, utilizing a std::list of Card objects named cards. Crucially, it tracks the bet placed on that specific hand and includes boolean flags isplit and ddown to manage the complex rules associated with hands that result from splitting or double down actions. Finally, the Player struct models all participants, including the dealer, holding fundamental data like id, name, and the current chips bankroll. A critical component is its hands field (std::list of Hand), which allows a player to manage multiple hands simultaneously if a split occurs. This struct also overloads operator< to facilitate the sorting of players by chip count for game summary and ranking purposes.

Sample Input/Output

Here is a sample input with one player, including input validation and round summary.

```
### Welcome to Blackjack Casino ###
Enter number of players (1-3): 1
Enter name for Player 1: Jade

================================================
                NEW ROUND STARTING
================================================
Jade (Chips: $1000), place your bet: -1
Invalid bet. Must be between $1 and $1000.
Jade (Chips: $1000), place your bet: 1000

--- Initial Deal ---

Dealer's upcard: [ 10♠ XX ]

--- Jade's Turn (Hand Bet: $1000) ---
Current Hand Score (14): [ 4♦ J♦ ]
Actions: (H)it / (S)tand
Choose action:  > H
Player hits. Dealing card.
Current Hand Score (22): [ 4♦ J♦ 8♠ ]
Hand Busted!
```

```
------------------------------------------------
               DEALER'S PLAY
------------------------------------------------
Dealer reveals hole card. Full Hand (20): [ 10♠ K♣ ]
Dealer Stands at 20.

------------------------------------------------
               FINAL SETTLEMENT
------------------------------------------------

--- Settlement for Jade's hand (Score: 22) ---
Player BUSTS. Bet of $1000 lost.

************************************************
Round Summary:

**Player 1 (Jade)** - Chips: $0

************************************************
Play another round? (Y/N): n

Thank you for playing Blackjack. Final Chip Counts:

**Player 1 (Jade)** - Chips: $0
Goodbye!
Program ended with exit code: 0
```

Checkoff Sheet, see next page for explanation

1. Container classes (Where in code did you put each of these Concepts and how were they used?
    1. Sequences (At least 1)
        1. list
        2. slist
        3. bit_vector
    2. Associative Containers (At least 2)
        1. set
        2. map
        3. hash
    3. Container adaptors (At least 2)
        1. stack
        2. queue
        3. priority_queue
2. Iterators
    1. Concepts (Describe the iterators utilized for each Container)
        1. Trivial Iterator
        2. Input Iterator
        3. Output Iterator
        4. Forward Iterator
        5. Bidirectional Iterator
        6. Random Access Iterator
3. Algorithms (Choose at least 1 from each category) 1. Non-mutating algorithms
        1. for_each
        2. find
        3. count
        4. equal
        5. search
    2. Mutating algorithms
        1. copy
        2. Swap
        3. Transform
        4. Replace
        5. fill
        6. Remove
        7. Random_Shuffle
    3. Organization
        1. Sort
        2. Binary search
        3. merge
        4. inplace_merge
        5. Minimum and maximum

Container classes

Sequence Containers

Type	Variable Name	Description	Location in Code	Usage Notes
std::list<std::string>	SUITS	A global constant list holding the Unicode strings for the four card suits (♠, ♡, ♢, ♣).	Global Constant	Used in createDk to iterate over all suits when building the deck.
std::list<Card>	deck	The main playing deck of cards.	Global Variable	Used for card storage, drawing cards (dealCrd), and shuffling (shufDk).
std::list<Card>	cards (within Hand struct)	The specific cards held by a player's or the dealer's hand.	struct Hand	Used for managing the cards in a hand, adding new cards on a 'Hit' (playHit), and moving cards during a 'Split' (playSplt).
std::list<Hand>	hands (within Player struct)	A list of hands a player currently has. Allows for multiple hands after a 'Split'.	struct Player	Managed in playRnd (initial hand) and modified in playSplt (insertion) and hdlPlay (iteration/progression).
std::list<Player>	plyrs	The main list of active players in the game.	runGame	Used to manage all players, from initial setup to removal (remove_if) if they run out of

				chips.
std::list<Card>	tmpDk	A temporary list used as a buffer during the custom list-based shuffling algorithm.	shufDk	Used with splice and erase/insert for efficient shuffling without data copies for the entire deck.

Associative Containers

Type	Variable Name	Description	Location in Code	Usage Notes
std::map<std::string, int> (Map)	CRDVALS	A global constant map storing the default integer value for each card rank (e.g., "K" -> 10, "A" -> 11).	Global Constant	Used in createDk to set the initial value for a Card object.
std::set<std::string> (Set)	VALRANKS	A global constant set of all valid card rank strings.	Global Constant	Used for validation or lookups of valid card ranks (though primarily used in createDk indirectly).
std::map<std::string, int> (Map)	initChp	A temporary map to store the initial chip count for each player at the start of a round.	playRnd	Used with std::for_each to quickly record player chips before bets are placed.

Container adapters

Type	Variable Name	Description	Location in Code	Usage Notes
std::stack<Card> (Stack)	disPile	The LIFO (Last-In, First-Out) discard pile for used cards.	Global Variable	Used to push cards from finished hands (discHnd) and pop them back into the deck when a reshuffle is needed (dealCrd).
std::queue<Player*> (Queue)	playQue	The FIFO (First-In, First-Out) container that tracks the order of player turns in a round.	Global Variable	Players are pushed in betting order (playRnd) and popped in turn order for the action phase (playRnd).

Iterators

Concept	Description of Usage	Container/Function
Trivial Iterator	Used implicitly when iterating over std::list in C++11 range-based for loops.	for (const auto& hand : p.hands) in prntStat and for (auto& hand : p.hands) in setHnd.
Input Iterator	Used by read-only algorithms, like std::accumulate and std::for_each when only reading elements from a container.	std::accumulate(hand.cards.cbegin(), ...) in calcScr.
Output Iterator	Used by std::for_each to write elements (specifically, pushing cards onto the disPile stack).	std::for_each(hand.cards.begin(), ...) in discHnd (using a lambda that performs an output action to the stack).
Forward Iterator	Iterators that can move forward and read, and are used by algorithms like	Used by std::list::iterator in discHnd, prntHnd, and playRnd iteration loops.

	std::for_each. Supports multi-pass operations.	
Bidirectional Iterator	Iterators that can move forward and backward. This is the category for std::list iterators.	std::list<Hand>::iterator curIt in hdlPlay and playSplt. The iterator is advanced (++it) and decremented (curIt--) and used for insert and erase.
Random Access Iterator	Simulated by combining the Bidirectional Iterator with std::advance function to jump to an arbitrary position n.	getNthCard (std::advance(it, n)) is used to achieve the effect of random access, though the performance is O(N).

Algorithms

Non-Mutating Algorithms

Algorithm	Location in Code	Description
std::for_each	discHnd	Iterates over all cards in a Hand's cards list and pushes them onto the disPile stack.
std::for_each	playRnd (3 instances)	Populates the initChp map; deals cards to all players in the playQue; displays final chips in the round summary.
std::accumulate	calcScr	Efficiently calculates the initial score of a hand by summing card values and counting the number of Aces using a lambda function that returns an aggregated std::pair.

Mutating Algorithms

Algorithm	Location in Code	Description
std::transform	hdlPlay and runGame	Converts the player's action choice and the "Play Again" choice to uppercase to

		standardize input for comparison.
std::advance	shufDk and playSplt	Used to move a list iterator a specific number of positions (n or 1) efficiently, as std::list iterators do not support direct addition/subtraction.

Organization

Algorithm	Location in Code	Description
std::remove_if	runGame and playRnd	runGame: Removes players from the plyrs list if their chip count falls below 1. playRnd: Cleans up any remaining empty Hand objects from a player's hands list after settlement.

Psuedocode

// Data Structures and Constants

STRUCT Card

    STRING rank

    STRING suit

    INTEGER value

END STRUCT

STRUCT Hand

    LIST of Card cards

    INTEGER bet

    BOOLEAN isplit // True if hand resulted from a split

```
    BOOLEAN ddown // True if Double Down was performed
END STRUCT

STRUCT Player
    INTEGER id
    STRING name
    INTEGER chips = 1000
    LIST of Hand hands
END STRUCT

MAP CRDVALS = {"A": 11, "2": 2, ..., "K": 10}
LIST SUITS = {"♠", "♥", "♦", "♣"}

GLOBAL LIST of Card deck
GLOBAL STACK of Card disPile
GLOBAL QUEUE of Player* playQue

// Main Program
FUNCTION main()
    SET output_precision to 0 decimal places
    CALL runGame()
END FUNCTION
```

```
// Main Game Loop
FUNCTION runGame()
    INITIALIZE LIST of Player plyrs
    INITIALIZE Player dealr = {id: 0, name: "Dealer", chips: 0}
    ADD empty Hand to dealr.hands

    PRINT "### Welcome to Blackjack Casino ###"

    // Setup Players
    READ numPlay
    WHILE numPlay < 1 OR numPlay > 3
        PRINT "Enter number of players (1-3):"
        READ numPlay
    END WHILE

    FOR i FROM 1 TO numPlay
        READ name
        CREATE new Player p = {i, name, 1000}
        ADD p to plyrs
    END FOR

    // Initial Deck Setup
    CALL createDk(4)
```

```
CALL shufDk()

STRING playAgn = "Y"
WHILE playAgn == "Y"
   TRY
      // Remove players out of chips
      REMOVE players from plyrs WHERE player.chips < 1
      IF plyrs IS EMPTY
         PRINT "All players are out of chips. Game Over."
         BREAK
      END IF

      // Reset Dealer Hand
      IF dealr.hands IS NOT EMPTY AND dealr.hands.front().bet > 0
         CALL discHnd(dealr.hands.front())
      ELSE IF dealr.hands IS EMPTY
         ADD empty Hand to dealr.hands
      END IF

      CALL playRnd(plyrs, dealr)

   CATCH std::exception e
      PRINT "CRITICAL GAME ERROR:" e.what()
```

```
            BREAK
        END TRY

        // Round Summary
        PRINT Round Summary
        FOR EACH Player p IN plyrs
            CALL prntStat(p, inclHnd: FALSE)
        END FOR

        READ playAgn
        CONVERT playAgn to UPPERCASE
    END WHILE

    PRINT Final Chip Counts
    FOR EACH Player p IN plyrs
        CALL prntStat(p, inclHnd: FALSE)
    END FOR
    PRINT "Goodbye!"
END FUNCTION

// Game Round Logic
FUNCTION playRnd(LIST of Player plyrs, Player dealr)
    PRINT "NEW ROUND STARTING"
```

```
// Reshuffle Check
IF deck.size() < 60
    PRINT "Performing full reshuffle."
    CALL createDk(4)
    CALL shufDk()
END IF

MAP initChp // Stores initial chips
FOR EACH Player p IN plyrs
    SET initChp[p.name] = p.chips
END FOR

// Place Bets and Setup Turn Queue
FOR EACH Player p IN plyrs
    READ betAmt
    WHILE betAmt < 1 OR betAmt > p.chips
        PRINT "Invalid bet."
        READ betAmt
    END WHILE
    ADD new Hand to p.hands
    SET p.hands.front().bet = betAmt
    DECREMENT p.chips BY betAmt
```

```
    ADD p to playQue
END FOR

// Initial Deal (Player 1, Dealer 1, Player 2, Dealer 2)
PRINT "--- Initial Deal ---"
QUEUE tempQ = playQue
WHILE tempQ IS NOT EMPTY
    CALL dealCrd(tempQ.front()->hands.front().cards) // Card 1 to all players
    REMOVE front of tempQ
END WHILE
CALL dealCrd(dealr.hands.front().cards) // Card 1 to dealer

tempQ = playQue
WHILE tempQ IS NOT EMPTY
    CALL dealCrd(tempQ.front()->hands.front().cards) // Card 2 to all players
    REMOVE front of tempQ
END WHILE
CALL dealCrd(dealr.hands.front().cards) // Card 2 (hole card) to dealer

// Display Dealer's Upcard
PRINT Dealer's upcard
CALL prntHnd(dealr.hands.front(), hide_1: TRUE)
```

```
// Check for Dealer Natural
Hand dealrH = dealr.hands.front()
BOOLEAN dealrNat = is_nat(dealrH)
IF dealrNat
   PRINT "DEALER NATURAL BLACKJACK!"
END IF

// Player Actions Phase
WHILE playQue IS NOT EMPTY
   Player* p = playQue.front()
   REMOVE front of playQue

   IF NOT dealrNat
      CALL hdlPlay(*p, dealrH)
   ELSE
      PRINT player: "Dealer has a Natural. Skip action phase."
   END IF
END WHILE

// Dealer Play
PRINT "DEALER'S PLAY"
INTEGER d_score = calcScr(dealrH)
PRINT "Dealer reveals hole card. Full Hand (" d_score "):"
```

```
CALL prntHnd(dealrH)

IF NOT dealrNat
    WHILE d_score < 17
        PRINT "Dealer Hits (score < 17)."
        CALL dealCrd(dealrH.cards)
        SET d_score = calcScr(dealrH)
        PRINT "Dealer's Hand (" d_score "):"
        CALL prntHnd(dealrH)
    END WHILE
    PRINT "Dealer Stands at " d_score "."
END IF

// Final Settlement Phase
PRINT "FINAL SETTLEMENT"
FOR EACH Player p IN plyrs
    FOR EACH Hand hand IN p.hands
        IF hand.bet > 0
            CALL setHnd(p, hand, dealrH)
        ELSE
            CALL discHnd(hand)
        END IF
    END FOR
```

```
        REMOVE hands from p.hands WHERE hand.cards IS EMPTY
    END FOR

    CALL discHnd(dealrH)
END FUNCTION

// Player Decision Logic
FUNCTION hdlPlay(Player p, Hand dlHnd)
    ITERATOR it = p.hands.BEGIN
    WHILE it IS NOT p.hands.END
        Hand curHnd = *it
        BOOLEAN done = FALSE

        // Skip completed Double Down hands
        IF curHnd.ddown
            INCREMENT it
            CONTINUE
        END IF

        // Handle Split Aces rule (must stand)
        IF curHnd.isplit AND curHnd.cards.size() == 2 AND curHnd.cards.front().rank == "A"
AND curHnd.cards.back().rank == "A"
            PRINT "Split Aces: Only one card is dealt to each. Must stand."
```

```
        INCREMENT it
        CONTINUE
    END IF

    BOOLEAN split_occurred = FALSE

    WHILE NOT done
        INTEGER score = calcScr(curHnd)
        PRINT "Current Hand Score (" score "):"
        CALL prntHnd(curHnd)

        IF score > 21
            PRINT "Hand Busted!"
            BREAK
        END IF
        IF score == 21
            PRINT "Hand is 21! Standing."
            BREAK
        END IF

        // Determine available actions
        BOOLEAN canSplt = (curHnd.cards.size() == 2 AND curHnd.cards.front().rank ==
curHnd.cards.back().rank AND NOT curHnd.isplit)
```

```
BOOLEAN canDbl = (curHnd.cards.size() == 2 AND p.chips >= curHnd.bet)

PRINT "Actions: (H)it / (S)tand"
IF canSplt PRINT " / (P)lit"
IF canDbl PRINT " / (D)ouble Down"
READ choice

CONVERT choice to UPPERCASE

IF choice == "H"
   CALL playHit(curHnd)
ELSE IF choice == "S"
   done = TRUE
ELSE IF choice == "D" AND canDbl
   CALL playDD(p, curHnd)
   done = TRUE
ELSE IF choice == "P" AND canSplt
   CALL playSplt(p, it) // playSplt modifies the list and the iterator 'it'
   split_occurred = TRUE
   BREAK
ELSE
   PRINT "Invalid or unavailable action."
END IF
```

```
        END WHILE

        IF NOT split_occurred
            INCREMENT it
        END IF
    END WHILE
END FUNCTION

// Card Scoring
FUNCTION calcScr(Hand hand)
    IF hand.cards IS EMPTY RETURN 0
    INTEGER score = 0
    INTEGER ace_cnt = 0

    FOR EACH Card c IN hand.cards
        IF c.rank == "A"
            INCREMENT ace_cnt
        ELSE
            INCREMENT score BY c.value
        END IF
    END FOR

    INCREMENT score BY ace_cnt * 11
```

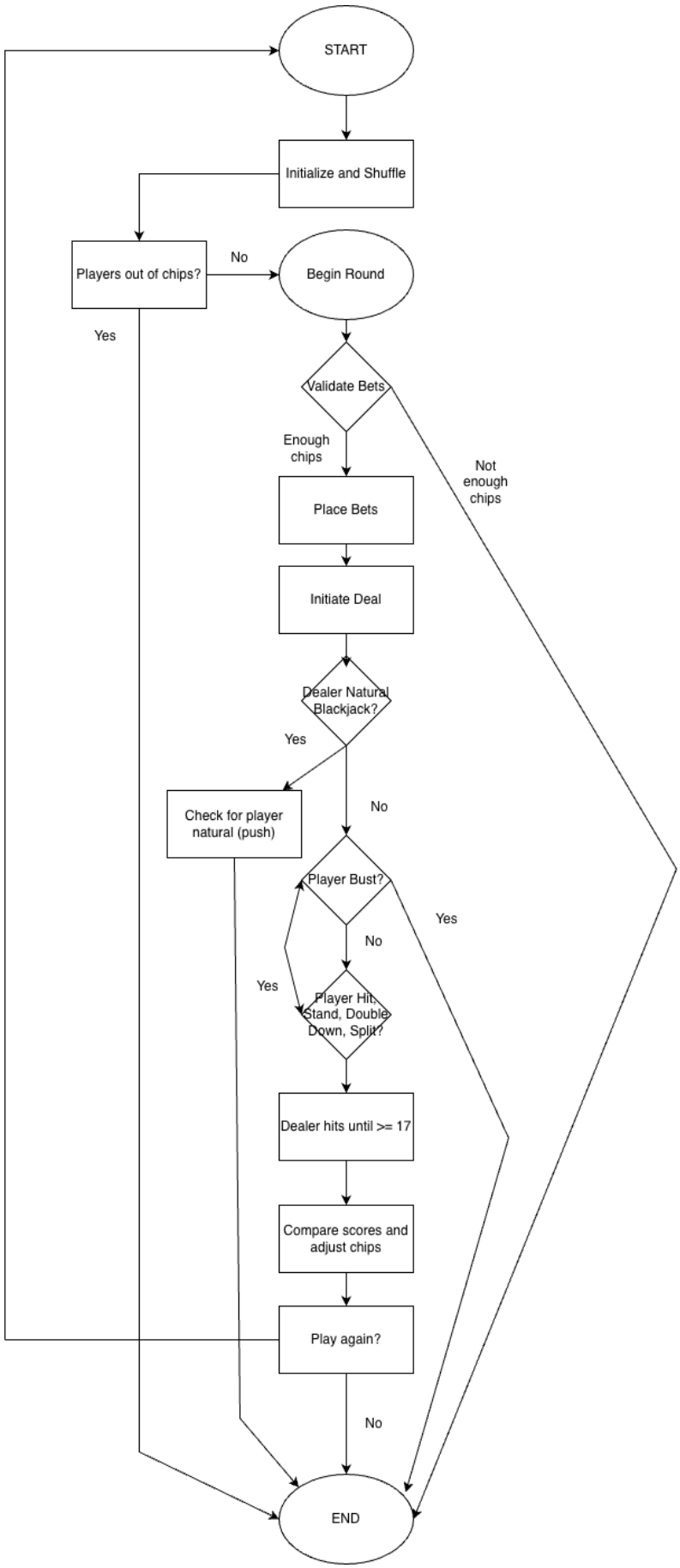
```
    WHILE score > 21 AND ace_cnt > 0
        DECREMENT score BY 10 // Convert an Ace from 11 to 1
        DECREMENT ace_cnt
    END WHILE
    RETURN score
END FUNCTION

// Hand Settlement
FUNCTION setHnd(Player p, Hand hand, Hand dlHnd)
    INTEGER p_score = calcScr(hand)
    INTEGER d_score = calcScr(dlHnd)

    PRINT "--- Settlement for" p.name "'s hand (Score:" p_score ") ---"

    IF p_score > 21 // Player bust
        PRINT "Player BUSTS. Bet lost."
    ELSE IF is_nat(hand) AND is_nat(dlHnd) // Natural vs Natural
        PRINT "PUSH (Natural vs. Natural). Bet returned."
        INCREMENT p.chips BY hand.bet
    ELSE IF is_nat(hand) // Player Natural
        INTEGER winAmt = hand.bet * 1.5
        PRINT "NATURAL BLACKJACK! Wins 1.5x."
        INCREMENT p.chips BY hand.bet + winAmt
```

```
    ELSE IF d_score > 21 // Dealer bust
        PRINT "Dealer BUSTS. Player wins."
        INCREMENT p.chips BY hand.bet * 2
    ELSE IF is_nat(dlHnd) // Dealer Natural
        PRINT "Dealer has NATURAL BLACKJACK. Bet lost."
    ELSE IF p_score > d_score // Player wins
        PRINT "Player Wins."
        INCREMENT p.chips BY hand.bet * 2
    ELSE IF p_score < d_score // Dealer wins
        PRINT "Dealer Wins."
    ELSE // Push
        PRINT "PUSH. Bet returned."
        INCREMENT p.chips BY hand.bet
    END IF

    CALL discHnd(hand)
END FUNCTION

// Deck Management (Simplified for Pseudocode)
FUNCTION createDk(INTEGER num_dk)
    CLEAR deck and disPile
    FOR num_dk times
        FOR EACH rank in CRDVALS
```

```
        FOR EACH suit in SUITS
            CREATE Card c with rank, suit, and value
            ADD c to deck
        END FOR
    END FOR
  END FOR
END FUNCTION

FUNCTION shufDk()
  IF deck IS EMPTY RETURN
  MOVE all cards from deck to a temporary LIST tmpDk
  SHUFFLE tmpDk using a random card selection and insertion method
  MOVE all cards from tmpDk back to deck
END FUNCTION

FUNCTION dealCrd(LIST of Card trgLst)
  IF deck IS EMPTY
    PRINT "Reshuffling Discard Pile"
    MOVE all cards from disPile STACK to deck LIST
    CALL shufDk()
    IF deck IS EMPTY THROW ERROR "No cards left"
  END IF
```

```
    MOVE Card from front of deck to trgLst
END FUNCTION


FUNCTION discHnd(Hand hand)
    MOVE all cards from hand.cards LIST to disPile STACK
    CLEAR hand.cards
    SET hand.bet = 0
END FUNCTION
```

Flowchart


START
Initialize and Shuffle
Players out of chips?
No
Begin Round
Yes
Validate Bets
Enough chips
Not enough chips
Place Bets
Initiate Deal
Dealer Natural Blackjack?
Yes
Check for player natural (push)
No
Player Bust?
Yes
No
Yes
Player Hit, Stand, Double Down, Split?
Dealer hits until >= 17
Compare scores and adjust chips
Play again?
No
END

Program

```
/*
* Author: Jade Thong
* Created: October 20, 2025, 6:33 PM
* Purpose: To simulate a Blackjack game
*/

// System Libraries Here
#include <iostream>
#include <string>
#include <list>
#include <map>
#include <set>
#include <stack>
#include <queue>
#include <algorithm>
#include <numeric>
#include <random>
#include <chrono>
#include <stdexcept>
#include <limits>
#include <sstream>
#include <iomanip> // For output formatting

// User Libraries Here
// Global Constants Only, No Global Variables
// Constants like PI, e, Gravity, Conversions, 2D array size only!

// Card Structure and Constants

// Card Structure
struct Card {
   std::string rank; // "A", "2", ..., "K"
   std::string suit; // "\u2660" (Spades), "\u2665" (Hearts), "\u2666" (Diamonds),
"\u2663" (Clubs)
   int value; // Primary value (Ace = 11)

   // Overloaded operators for list/set comparisons and unique checks
   // Required for std::find and comparisons
   bool operator==(const Card& oth) const {
       // Two cards are equal if both rank and suit match
```

```
        return rank == oth.rank && suit == oth.suit;
    }
    bool operator<(const Card& oth) const { // For sorting
        if (rank != oth.rank) return rank < oth.rank; // Sort by rank first
        return suit < oth.suit; // Then by suit
    }
    // For easy printing
    friend std::ostream& operator<<(std::ostream& os, const Card& c) {
        return os << c.rank << c.suit; // e.g., "A♠"
    }
};

// Map: Stores rank to its primary value
const std::map<std::string, int> CRDVALS = {
    // Ace is 11 by default; adjusted in scoring logic
    {"A", 11}, {"2", 2}, {"3", 3}, {"4", 4}, {"5", 5}, {"6", 6},
    {"7", 7}, {"8", 8}, {"9", 9}, {"10", 10}, {"J", 10}, {"Q", 10}, {"K", 10}
};
// List: Stores all suits
const std::list<std::string> SUITS = {"\u2660", "\u2665", "\u2666", "\u2663"};

// Set: Stores all valid ranks for validation
const std::set<std::string> VALRANKS = {
    "A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K" // Valid ranks
};

// Hand Structures
struct Hand {
    std::list<Card> cards; // Hand is a list of cards
    int bet = 0; // Bet amount for this hand
    bool isplit = false; // Flag to indicate if this hand is a result of a split
    bool ddown = false; // Flag for double down
};

// Player Structure
struct Player {
    int id; // Player ID
    std::string name; // Player Name
    int chips = 1000; // Starting chips
    // List: Player can have multiple hands (for splits)
    std::list<Hand> hands;
    // Operator for sorting by chips (descending)
```

```
        bool operator<(const Player& oth) const {
            return chips > oth.chips; // Descending order
        }
};

// Deck Management

// Function to simulate random access on a std::list
// Returns a pointer to the card at the nth position (0-indexed)
// Note: This is O(N) complexity, not O(1) like std::vector
Card* getNthCard(std::list<Card>& deck, int n) {
    // Check for out-of-bounds access
    if (n < 0 || n >= deck.size()) {
        return nullptr; // Return null if index is invalid
    }

    // Iterator to the beginning of the list
    auto it = deck.begin();

    // std::advance moves the iterator 'n' steps forward
    std::advance(it, n);

    // Return a pointer to the element the iterator points to
    return &(*it);
}

// Global Containers for Game State
std::list<Card> deck;          // Deck of cards
std::stack<Card> disPile;      // LIFO discard
std::queue<Player*> playQue;   // Tracks turn order

// Function Prototypes Here

// Clear input buffer
void clear_in() {
    std::cin.clear(); // Clear any error flags
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Discard
invalid input
}

// Scoring Functions
// Calculates the best Blackjack score for a hand.
```

```cpp
int calcScr(const Hand& hand) {
   if (hand.cards.empty()) { // No cards means score is 0
       return 0;
   }

   // Use std::accumulate to sum card values and count Aces
   auto result = std::accumulate(hand.cards.cbegin(), hand.cards.cend(),
std::make_pair(0, 0),
       // Pair: first = sum of non-Ace values, second = count of Aces
       [](std::pair<int, int> acc, const Card& c) {
           if (c.rank == "A") { // Ace handling
               acc.second += 1; // Count Aces
           } else {
               acc.first += c.value; // Sum non-Ace values
           }
           return acc;
       }
   );

   // Unpack results
   int initial = result.first; // Sum of non-Ace values
   int ace_cnt = result.second; // Number of Aces
   int score = initial; // Start with non-Ace sum

   // Add Aces as 11 initially
   score += ace_cnt * 11;

   // Adjust Ace values
   while (score > 21 && ace_cnt > 0) { // While busting and have Aces to adjust
       score -= 10; // Change 11 to 1 (11 - 1 = 10 difference)
       ace_cnt--; // One less Ace to adjust
   }

   // Final score
   return score; // Final score
}

// Checks if a hand is a natural Blackjack (21 with 2 cards).
bool is_nat(const Hand& hand) {
   // Natural only if 2 cards and not a split hand
   return calcScr(hand) == 21 && hand.cards.size() == 2 && !hand.isplit;
}
```

```
// Deck Management
// Creates a standard deck with the specified number of decks.
void createDk(int num_dk) { // Number of decks to create
   deck.clear(); // Clear existing deck
   while(!disPile.empty()) disPile.pop(); // Clear discard pile

   // Nested loops to create the deck(s)
   for (int d = 0; d < num_dk; ++d) {
       // Nested iteration over map and list
       for (const auto& r_pair : CRDVALS) {  // r_pair: (rank, value)
           for (const std::string& suit : SUITS) {
               Card c = {r_pair.first, suit, r_pair.second}; // Create card
               deck.push_back(c); // Add to deck
           }
       }
   }
}

// Shuffles the deck using a custom algorithm with list iterators
void shufDk() {
   if (deck.empty()) return; // Don't shuffle an empty deck

   // Temporary list for shuffling
   std::list<Card> tmpDk;
   tmpDk.splice(tmpDk.begin(), deck); // Efficiently moves all elements

   // Random number generator setup
   unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
   std::mt19937 g(seed);

   int n = tmpDk.size(); // Number of cards
   if (n == 0) return;

   // Custom list-based shuffle
   for (int i = 0; i < n * 2; ++i) { // Shuffle iterations n*2
       auto it_from = tmpDk.begin(); // Iterator to select card
       // Iterator: std::advance to a random position
       std::advance(it_from, g() % n);

       // Remove card from current position
       Card c = *it_from; // Copy card
```

```
        it_from = tmpDk.erase(it_from); // Erase returns iterator to next element

        // Insert card at new random position
        int trgPos = g() % n; // Target position
        auto it_to = tmpDk.begin(); // Iterator to insert position
        std::advance(it_to, trgPos); // Move to target position

        // Insert card
        tmpDk.insert(it_to, c);
    }
    // Move shuffled cards back to deck
    deck.splice(deck.begin(), tmpDk);
}

// Deals a card from the deck to the hand, reshuffling if necessary
// Uses list iterators for deck management
void dealCrd(std::list<Card>& trgLst) { // Target list to receive card
    // Reshuffle if deck is empty
    if (deck.empty()) {
        std::cout << "\n--- Reshuffling Discard Pile ---\n";

        // Move cards from stack to list for reshuffling
        while (!disPile.empty()) { // While discard pile not empty
            deck.push_back(disPile.top()); // List push_back
            disPile.pop(); // Stack pop
        }
        shufDk(); // Shuffle the deck
        if (deck.empty()) { // Still empty after reshuffle
             throw std::runtime_error("No cards left to deal or shuffle!");
        }
    }

    // Iterator: deck.begin() gives an iterator to the first element
    trgLst.push_back(*deck.begin());
    deck.pop_front();
}

// Moves all cards from a Hand to the discard pile.
void discHnd(Hand& hand) {
    // STL Algorithm: std::for_each to iterate and push to stack
    std::for_each(hand.cards.begin(), hand.cards.end(), [](const Card& c) {
        disPile.push(c); // Stack push
```

```
    });
    hand.cards.clear(); // List clear
    hand.bet = 0; // Reset bet
}

// Display Functions
// Print hand, optionally hiding second card

void prntHnd(const Hand& hand, bool hide_1 = false) {
    std::cout << "[ "; // Start hand display
    int count = 0; // Card counter

    // Iterator: std::list::const_iterator
    for (auto it = hand.cards.cbegin(); it != hand.cards.cend(); ++it) {
        // Hide second card if specified
        if (hide_1 && count == 1) {
            // Hidden card representation
            std::cout << "XX ";
        } else {
            std::cout << *it << " "; // Use overloaded operator
        }
        count++; // Increment card counter
    }
    std::cout << "]";
}

// Print player statistics and hands
void prntStat(const Player& p, bool inclHnd = true) {
    // Player header display with chips
    std::cout << "\n**Player " << p.id << " (" << p.name << ")** - Chips: $" << p.chips
<< "\n";
    // Include the hand details
    if (inclHnd) {
        int hndCnt = 1; // Hand counter
        // Iterator: Iterate over all hands
        for (const auto& hand : p.hands) {
            int score = calcScr(hand); // Calculate hand score
            std::cout << "  Hand " << hndCnt++; // Hand number
            if (hand.isplit) std::cout << " (Split)"; // Indicate split hand
            if (hand.ddown) std::cout << " (DD)"; // Indicate double down hand
            std::cout << " Bet: $" << hand.bet << " Score: (" << score << "): ";
            prntHnd(hand, false); // Print hand cards
```

```
            std::cout << "\n";
        }
    }
}

// Game Logic Functions

// Handles the "Hit" action for a specific hand.
// Reference to hand to modify
void playHit(Hand& hand) {
    std::cout << "Player hits. Dealing card.\n";
    dealCrd(hand.cards); // Deal card to hand
}

// Handles the "Split" action for a player.
// Modifies the player's hands list and the current hand iterator.
void playSplt(Player& p, std::list<Hand>::iterator& curIt) {
    Hand& origHnd = *curIt; // Reference to the original hand

    // Check if splitting is valid (two cards of the same rank)
    if (origHnd.cards.size() != 2 || origHnd.cards.front().rank !=
origHnd.cards.back().rank) { // Invalid split
        std::cout << "Cannot split this hand.\n";
        return;
    }

    // Check if player has enough chips to place the second bet
    if (p.chips < origHnd.bet) {
        std::cout << "Not enough chips to place a second bet for splitting.\n";
        return;
    }

    // Proceed with split
    std::cout << "Splitting Hand. Placing additional $" << origHnd.bet << " bet.\n";

    // Create the new hand (the second split hand)
    Hand newHnd; // New hand for the split
    newHnd.bet = origHnd.bet; // Same bet as original hand
    newHnd.isplit = true; // Mark as split hand

    // Move the second card from the original hand to the new hand
    // Iterator: Get iterator to the second card (list::begin() and advance)
```

```
    auto scndIt = origHnd.cards.begin();
    std::advance(scndIt, 1); // Move to second card

    // Move card to new hand
    newHnd.cards.push_back(*scndIt);
    origHnd.cards.erase(scndIt); // Remove from original hand

    // Insert the new hand *after* the original hand in the player's hands list
    // This allows the player to play the hands sequentially (original first, then new)
    curIt++; // Move iterator to the position AFTER the original hand
    p.hands.insert(curIt, newHnd); // Insert new hand
    curIt--; // Move iterator back to point to the original hand for continued play

    // Update chip count and deal second cards
    p.chips -= newHnd.bet;

    // Deal the second card to the original hand
    dealCrd(origHnd.cards);

    // Deal the second card to the new hand (now located one position forward)
    curIt++; // Iterator to the new hand
    dealCrd(curIt->cards);
    curIt--; // Iterator back to the original hand

    std::cout << "Split successful. Playing the first hand...\n";
}

// Handles the "Double Down" action for a player.
void playDD(Player& p, Hand& hand) { // Reference to player and hand
    // Check if Double Down is valid (only on initial two cards)
    if (hand.cards.size() != 2) { // If not initial two cards
        std::cout << "Double Down only allowed on initial two cards.\n";
        return;
    }
    if (p.chips < hand.bet) { // If not enough chips
        std::cout << "Not enough chips to Double Down.\n";
        return;
    }

    // Proceed with Double Down and output
    std::cout << "Player Doubles Down! Betting an additional $" << hand.bet << ".\n";
    p.chips -= hand.bet; // Deduct additional bet
```

```
    hand.bet *= 2; // Double the bet
    hand.ddown = true; // Mark hand as double down

    // Player gets exactly one card
    playHit(hand);

    // Show final hand and output score
    std::cout << "Final Hand Score: (" << calcScr(hand) << ")\n";
    prntHnd(hand);
    std::cout << "\n";
}

// Processes the outcome of a single hand against the dealer.
// Updates player chips based on the result.
// References to player, player's hand, and dealer's hand
void setHnd(Player& p, Hand& hand, const Hand& dlHnd) {
    int p_score = calcScr(hand); // Player's hand score
    int d_score = calcScr(dlHnd); // Dealer's hand score

    // Output settlement header
    std::cout << "\n--- Settlement for " << p.name << "'s hand (Score: " << p_score <<
") ---\n";

    // Player bust
    if (p_score > 21) {  // If score is over 21
        std::cout << "Player BUSTS. Bet of $" << hand.bet << " lost.\n";
        // Chips already deducted at bet time
    }
    // Both have naturals
    else if (is_nat(hand) && is_nat(dlHnd)) {
        std::cout << "PUSH (Natural vs. Natural). Bet of $" << hand.bet << "
returned.\n";
        p.chips += hand.bet; // Return original bet
    }
    // Natural blackjack for player
    else if (is_nat(hand)) {
        int winAmt = static_cast<int>(hand.bet * 1.5); // 1.5x winnings
        std::cout << "NATURAL BLACKJACK! Wins 1.5x. $" << winAmt << " won (Total
return: $" << hand.bet + winAmt << ").\n";
        p.chips += hand.bet + winAmt; // Return original bet + winnings
    }
    // Dealer bust
```

```
    else if (d_score > 21) { // If score is over 21 for dealer
        std::cout << "Dealer BUSTS (" << d_score << "). Player wins $" << hand.bet <<
".\n";
        p.chips += hand.bet * 2; // Return original bet + winnings
    }
    // Dealer has natural blackjack
    else if (is_nat(dlHnd)) {
        std::cout << "Dealer has NATURAL BLACKJACK. Bet of $" << hand.bet << "
lost.\n";
    }
    // Compare scores
    else if (p_score > d_score) { // Player wins
        std::cout << "Player Wins (" << p_score << " > " << d_score << "). Wins $" <<
hand.bet << ".\n";
        p.chips += hand.bet * 2; // Return original bet + winnings
    }
    else if (p_score < d_score) { // Dealer wins
        std::cout << "Dealer Wins (" << d_score << " > " << p_score << "). Bet of $" <<
hand.bet << " lost.\n";
    }
    else { // Push
        std::cout << "PUSH (" << p_score << " vs. " << d_score << "). Bet of $" <<
hand.bet << " returned.\n";
        p.chips += hand.bet; // Return original bet
    }

    discHnd(hand);
}


// Game Logic Functions

// Handles the main player decision phase (Hit, Stand, Split, Double Down).
void hdlPlay(Player& p, Hand& dlHnd) {
    std::string choice; // Player choice input

    // Use a while loop with an iterator to manage the list of hands,
    // allowing for insertion (splitting) and safe iteration.
    auto it = p.hands.begin(); // Iterator to current hand
    while (it != p.hands.end()) { // While there are hands to play
        Hand& curHnd = *it; // Reference to current hand
        bool done = false; // Flag to indicate if done playing this hand
```

```
        // Output current hand header
        std::cout << "\n--- " << p.name << "'s Turn (Hand Bet: $" << curHnd.bet << ")
---\n";

        // Skip hands that were just completed by a Double Down
        if (curHnd.ddown) {
            ++it; // Move to the next hand
            continue;
        }

        // Handle a split pair of Aces
        if (curHnd.isplit && curHnd.cards.size() == 2 && curHnd.cards.front().rank ==
"A" && curHnd.cards.back().rank == "A") {
            std::cout << "Split Aces: Only one card is dealt to each. Must stand.\n";
            ++it; // Move to the next hand after standing
            continue;
        }

        // Inner loop for playing the current hand
        bool split = false;

        // Loop until the player stands, busts, or completes the hand
        while (!done) {
            int score = calcScr(curHnd); // Calculate current hand score
            std::cout << "Current Hand Score (" << score << "): ";
            prntHnd(curHnd); // Print current hand
            std::cout << "\n";

            // Check for bust or 21
            if (score > 21) { // If score exceeds 21
                std::cout << "Hand Busted!\n";
                break;
            }
            if (score == 21) { // If score is exactly 21
                std::cout << "Hand is 21! Standing.\n";
                break;
            }

            // Prompt for action
            std::cout << "Actions: (H)it / (S)tand";
```

```
            // Check if Split and Double Down are available
            // Can split if two cards of same rank and not already a split hand
            bool canSplt = (curHnd.cards.size() == 2 && curHnd.cards.front().rank ==
curHnd.cards.back().rank && !curHnd.isplit);
            bool canDbl = (curHnd.cards.size() == 2 && p.chips >= curHnd.bet); // Can
double down if two cards and enough chips

            // Display available actions
            if (canSplt) std::cout << " / (P)lit"; // 'P' for sPlit to avoid confusion
with 'S'tand
            if (canDbl) std::cout << " / (D)ouble Down"; // 'D' for Double Down
            std::cout << "\nChoose action: ";

            std::cout << " > ";

            // Player chooses action
            std::cin >> choice;

            // Standardize input to uppercase
            std::transform(choice.begin(), choice.end(), choice.begin(), ::toupper);
            clear_in();

            // Handle player choice
            if (choice == "H") { // Hit
                playHit(curHnd); // Deal card to hand
            } else if (choice == "S") { // Stand
                done = true; // End turn for this hand
            } else if (choice == "D" && canDbl) { // Double Down
                playDD(p, curHnd); // Handle Double Down
                done = true; // Double Down ends the turn for this hand
            } else if (choice == "P" && canSplt) {
                // The player_split function modifies the list and the iterator 'it'
                playSplt(p, it);
                split = true; // Mark that a split occurred
                break;
            } else {
                std::cout << "Invalid or unavailable action.\n"; // Prompt again
            }
        } // end while (!done_playing)

        // Handle iterator progression after hand completion
        if (split) {
```

```
            // After a split, stay on the current iterator to play the new hand next
            // 'it' already points to the original hand; increment to the new hand
        } else {
            // Normal progression: move to the next hand in the list
            ++it;
        }

    } // end while (it != p.hands.end())
}

// Plays a single round of Blackjack for all players and the dealer.
void playRnd(std::list<Player>& plyrs, Player& dealr) {

    // Bets and Initial Deal
    std::cout << "\n" << std::string(50, '=') << "\n"; // Round header
    std::cout << "                NEW ROUND STARTING\n";
    std::cout << std::string(50, '=') << "\n";

    // Reshuffle check
    if (deck.size() < 60) { // Threshold for reshuffle
        std::cout << "Deck size (" << deck.size() << ") is low. Performing full
reshuffle.\n";
        createDk(4); // Recreate deck with 4 decks
        shufDk(); // Shuffle the deck
    }
    //  Store initial chips for all players (STL Map)
    std::map<std::string, int> initChp;

    // STL Algorithm: std::for_each to populate the map
    std::for_each(plyrs.cbegin(), plyrs.cend(), [&](const Player& p) {
        initChp[p.name] = p.chips; // Map insert/assignment
    });

    // Place Bets and set up Turn Queue (STL Container: std::queue)
    std::for_each(plyrs.begin(), plyrs.end(), [&](Player& p) {
        int betAmt = 0; // Bet amount input

        // Prompt for bet until valid
        while (betAmt < 1 || betAmt > p.chips) { // Invalid bet
            std::cout << p.name << " (Chips: $" << p.chips << "), place your bet: ";
            std::cin >> betAmt; // Input bet amount
            clear_in();
```

```
            if (betAmt < 1 || betAmt > p.chips) { // Invalid bet
                std::cout << "Invalid bet. Must be between $1 and $" << p.chips <<
".\n";
            }
        }
        // Set up player's initial hand and deduct chips
        p.hands.emplace_back(); // Add initial hand
        p.hands.front().bet = betAmt; // Set bet for the hand
        p.chips -= betAmt; // Deduct bet from chips
        playQue.push(&p); // Add player to the turn order queue
    });

    // Initial Deal (Player, Dealer, Player, Dealer)
    std::cout << "\n--- Initial Deal ---\n";
    // Deal card 1 to all players (in order)
    std::queue<Player*> tempQ = playQue; // Copy queue for iteration
    while (!tempQ.empty()) { // While queue not empty
        dealCrd(tempQ.front()->hands.front().cards); // Deal to player's first hand
        tempQ.pop(); // Remove from temp queue
    }
    // Deal card 1 to dealer
    dealCrd(dealr.hands.front().cards);

    // Deal card 2 to all players
    tempQ = playQue;
    while (!tempQ.empty()) { // While queue not empty
        dealCrd(tempQ.front()->hands.front().cards); // Deal to player's first hand
        tempQ.pop(); // Remove from temp queue
    }
    // Deal card 2 to dealer
    dealCrd(dealr.hands.front().cards); // Dealer's hole card

    // Display initial hands
    std::cout << "\nDealer's upcard: ";
    prntHnd(dealr.hands.front(), true); // Hide the second card
    std::cout << "\n";

    // Check for naturals
    Hand& dealrH = dealr.hands.front(); // Dealer's hand
    bool dealrNat = is_nat(dealrH); // Check if dealer has natural

    if (dealrNat) {
```

```
        std::cout << "\n**DEALER NATURAL BLACKJACK!**\n";
    }

    // Player Actions Phase
    while (!playQue.empty()) { // While there are players to process
        Player* p = playQue.front(); // Queue: front()
        playQue.pop(); // Queue: pop()

        // If dealer has natural, only check for push, otherwise players play
        if (!dealrNat) {
            hdlPlay(*p, dealrH);
        } else {
            std::cout << "\n" << p->name << ": Dealer has a Natural. Skip action
phase.\n";
        }
    }

    // Dealer Play
    std::cout << "\n" << std::string(50, '-') << "\n";
    std::cout << "               DEALER'S PLAY\n";
    std::cout << std::string(50, '-') << "\n";

    int d_score = calcScr(dealrH); // Dealer's initial score
    std::cout << "Dealer reveals hole card. Full Hand (" << d_score << "): ";
    prntHnd(dealrH); // Print dealer's full hand
    std::cout << "\n";

    if (!dealrNat) { // Only play if dealer doesn't have natural
        while (d_score < 17) { // Dealer hits on soft 17
            std::cout << "Dealer Hits (score < 17).\n";
            dealCrd(dealrH.cards); // Deal card to dealer
            d_score = calcScr(dealrH); // Recalculate score
            std::cout << "Dealer's Hand (" << d_score << "): ";
            prntHnd(dealrH); // Print dealer's hand
            std::cout << "\n";
        }
        std::cout << "Dealer Stands at " << d_score << ".\n";
    }

    // Final Settlement Phase
    std::cout << "\n" << std::string(50, '-') << "\n";
    std::cout << "               FINAL SETTLEMENT\n";
```

```
    std::cout << std::string(50, '-') << "\n";

    // STL Algorithm: std::for_each to iterate over all players
    std::for_each(plyrs.begin(), plyrs.end(), [&](Player& p) {
        // Iterator: Iterate over all hands a player might have (original + split
hands)
        for (auto& hand : p.hands) { // For each hand
            if (hand.bet > 0) { // Only settle hands that were bet on
                setHnd(p, hand, dealrH); // Settle the hand
            } else {
                discHnd(hand); // Clean up empty hands if any somehow remain
            }
        }
        // Cleanup: Use std::list::remove_if to clean up all empty hands
        p.hands.remove_if([](const Hand& h){ // Lambda to check if hand is empty
            return h.cards.empty(); // Remove if empty
        });
    });

    // Discard dealer's hand
    discHnd(dealrH);
}

// Main Game Loop
void runGame() {
    std::list<Player> plyrs; // List of players
    Player dealr = {0, "Dealer", 0}; // Dealer player
    dealr.hands.emplace_back(); // Dealer always has one hand

    std::cout << "### Welcome to Blackjack Casino ###\n";

    // Setup Players
    int numPlay = 0; // Number of players input
    while (numPlay < 1 || numPlay > 3) {
        std::cout << "Enter number of players (1-3): ";
        std::cin >> numPlay; // Input number of players
        clear_in();
    }

    for (int i = 1; i <= numPlay; ++i) { // Get player names
        std::string name; // Player name input
        std::cout << "Enter name for Player " << i << ": ";
```

```
        std::getline(std::cin, name); // Input player name
        plyrs.emplace_back(Player{i, name, 1000}); // Add player to list
    }

    // Initial Deck Setup
    createDk(4); // Create deck with 4 standard decks
    shufDk(); // Shuffle the deck

    // Play Again Loop
    std::string playAgn = "Y";

    while (playAgn == "Y") { // While player wants to play again
        try {
            // Check for players who are out of money
            // STL Algorithm: std::remove_if (using lambda) to manage player list
            plyrs.remove_if([&](const Player& p) { // Lambda to check if player is out
of chips
                if (p.chips < 1) { // If player has no chips
                    std::cout << "\n" << p.name << " is out of chips and leaves the
game.\n";
                    return true;
                }
                return false; // Keep player otherwise
            });

            // Check if any players remain
            if (plyrs.empty()) { // No players left
                std::cout << "\nAll players are out of chips. Game Over.\n";
                break;
            }

            // Check if dealer's hand needs reset
            if (!dealr.hands.empty() && dealr.hands.front().bet > 0) { // If dealer's
hand has cards
                 discHnd(dealr.hands.front()); // Discard dealer's hand
            } else if (dealr.hands.empty()) { // If dealer has no hands
                dealr.hands.emplace_back(); // Create dealer's hand
            }

            // Play a round of Blackjack
            playRnd(plyrs, dealr);
```

```
        } catch (const std::exception& e) { // Catch any critical errors
            std::cerr << "CRITICAL GAME ERROR: " << e.what() << "\n";
            break;
        }

        // Round Summary after each round
        std::cout << "\n" << std::string(50, '*') << "\n";
        std::cout << "Round Summary:\n"; // Summarize player chips

        // STL Algorithm: std::for_each to display final chips
        std::for_each(plyrs.begin(), plyrs.end(), [](const Player& p) {
            prntStat(p, false); // Print player stats without hands

        });

        // Prompt to play again or not
        std::cout << "\n" << std::string(50, '*') << "\n";
        std::cout << "Play another round? (Y/N): ";
        std::cin >> playAgn; // Input choice
        std::transform(playAgn.begin(), playAgn.end(), playAgn.begin(), ::toupper);

        // Clear input buffer
        clear_in();

    }

    // If player chooses not to play again or all players are out
    std::cout << "\nThank you for playing Blackjack. Final Chip Counts:\n";
    std::for_each(plyrs.begin(), plyrs.end(), [](const Player& p) { // Final stats
        prntStat(p, false); // Print player stats without hands
    });

    std::cout << "Goodbye!\n";
}

int main (int argc, char** argv) {
    // Set Random Number Seed Here (System clock used in shuffle_deck)

    // Declare all Variables Here (Done within run_game_loop)

    // Input or initialize values Here
```

```
    // Process/Calculations Here
    // Setting fixed point notation for chips display
    std::cout << std::fixed << std::setprecision(0);

    runGame();

    // Output Located Here

    // Exit
    return 0;
}
```

References

1. Textbook
2. Standard Template Library
3. https://en.cppreference.com/w/cpp/standard_library.html
4. https://bicyclecards.com/how-to-play/blackjack - For Blackjack rules