PROJECT 2

BLACKJACK

A CARD GAME

Ireoluwa Dairo

CSC/CIS 17A

47538

FALL 2024

INTRODUCTION

The Blackjack game implementation in C++ demonstrates advanced object-oriented programming techniques and modern software design patterns. Built using a comprehensive class hierarchy, the game simulates classic casino Blackjack where players compete against a dealer to achieve a hand value closest to 21 without exceeding it. The project consists of a combined total of 1872 lines of code with less than 20% being comments.

Summary

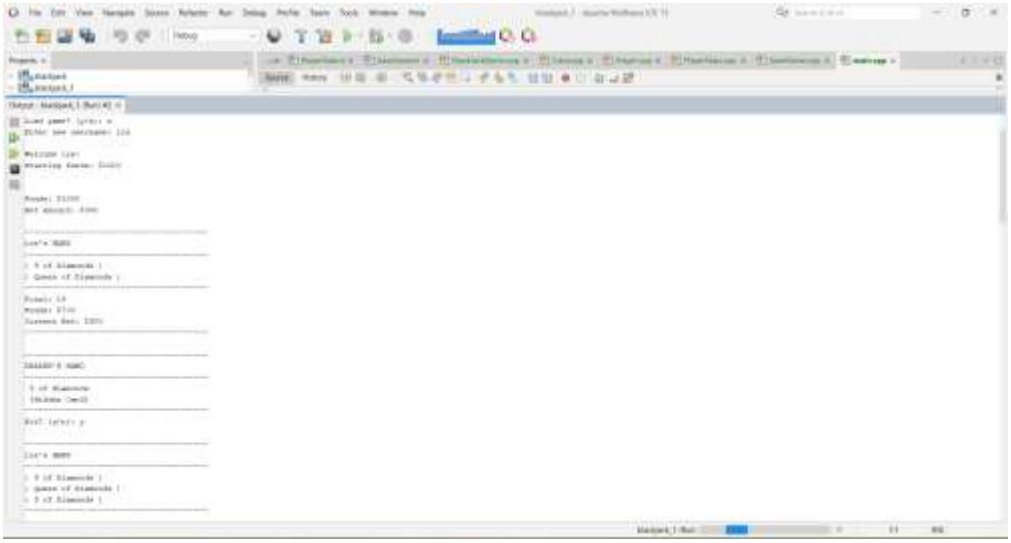
The project architecture centers around several key classes that work together to create a robust gaming experience. The Game class acts as a singleton controller, coordinating between a template-based Deck class for card management and polymorphic Player classes that define behavior for both human players and the dealer. Smart pointers ensure proper memory management, while comprehensive exception handling provides robustness. Statistics tracking through the Player Stats class offers detailed performance metrics. The implementation spans multiple source files with clear separation between specification (.h) and implementation (.cpp) files. The code demonstrates the following C++ concepts:

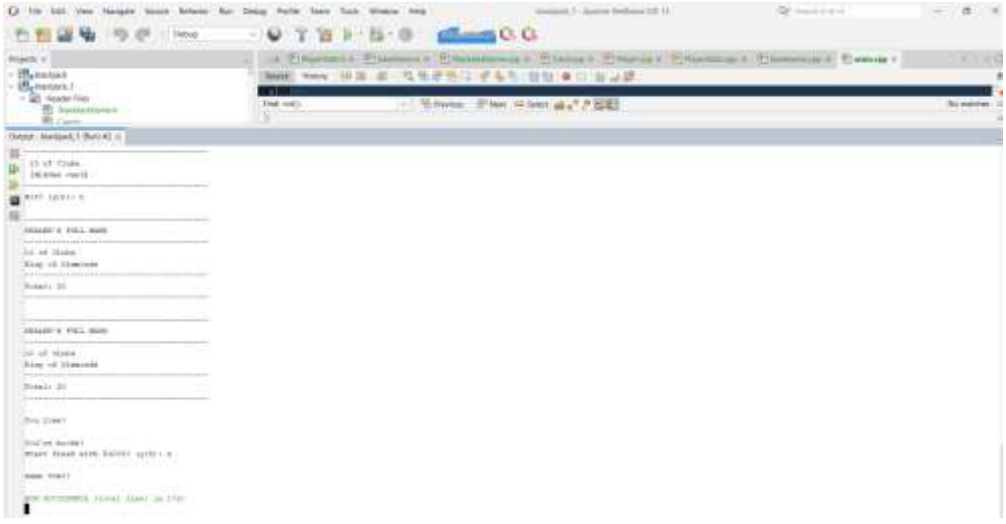
1. Classes - Through comprehensive class hierarchies and encapsulation
2. Inheritance - Via the Player base class and its derivatives
3. Operator Overloading - In the Card class for comparison and assignment
4. Polymorphism - Through virtual functions in the Player hierarchy
5. Templates - With the Deck class implementation

Description

The implementation flow begins with game initialization, either loading a saved game or starting fresh with a $1000 balance. Players place bets and receive two initial cards, with one dealer card visible. The game then follows standard Blackjack rules - players can hit or stand, the dealer must hit on 16 and stand on 17, and hand values are compared to determine winners. The architecture supports this through polymorphic player classes that define different behaviors for humans and the dealer. The Deck template class manages card distribution using vectors for dynamic storage, while exception handling ensures error management. Statistics tracking records wins and losses, while a save/load system maintains game state between sessions.

Sample Input/Output





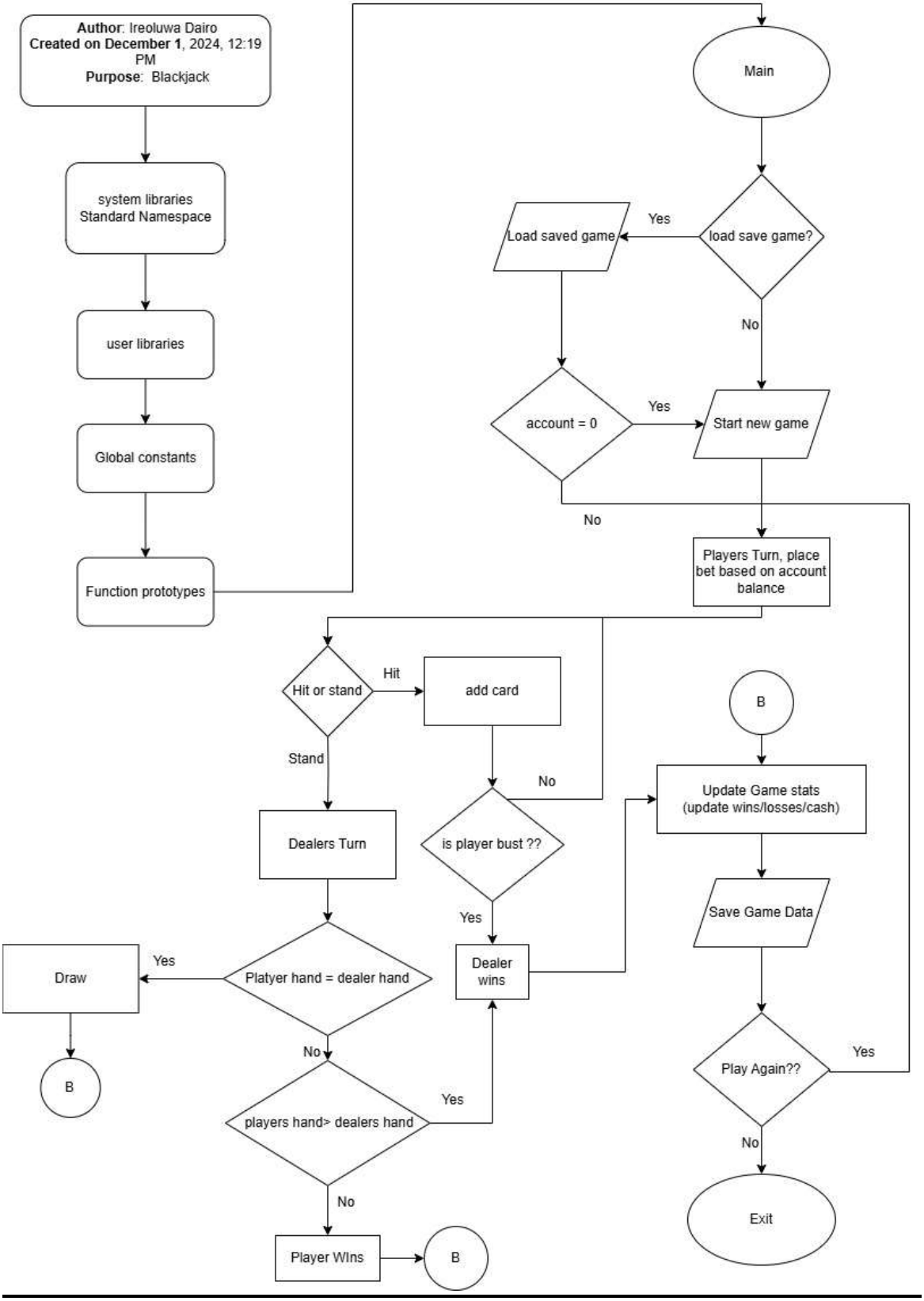
Flowcharts/Pseudocode/UML

```
PROGRAM BlackjackGame

CLASS Card
   PRIVATE:
      value: integer
      suit: enum (HEARTS, DIAMONDS, CLUBS, SPADES)
      static cardCount: integer

   PUBLIC:
      Constructor(value, suit)
      getValue(): integer
      getSuit(): Suit
      overload operators (==, !=, =)

CLASS Deck<T> // Template class
   PRIVATE:
      cards: vector of T
      dealtCards: vector of T

   PUBLIC:
      Constructor(numDecks)
      shuffle()
      drawCard(): T
      isEmpty(): boolean
      remainingCards(): integer

CLASS Player (Abstract)
   PROTECTED:
      name: string
      hand: vector of Card
      funds: integer
      bet: integer

   PUBLIC:
      virtual wantHit(): boolean
      virtual dispHand()
      addCard(Card)
      getVal(): integer
      isBust(): boolean
      placeBet(amount): boolean

CLASS Human INHERITS Player
   PUBLIC:
      Constructor(name)
```

```
        wantHit(): boolean // Interactive
        dispHand() // Shows all cards

CLASS Dealer INHERITS Player
    PUBLIC:
        Constructor()
        wantHit(): boolean // Automatic based on hand value
        dispHand() // Shows one card hidden
        showAll() // Shows all cards

CLASS PlayerStats
    PRIVATE:
        playerName: string
        totalGames: integer
        gamesWon: integer
        totalProfit: integer

    PUBLIC:
        addGame(isWin, betAmount, profit)
        getWinRate(): double
        getTotalProfit(): integer

CLASS Game (Single Instance)
    PRIVATE:
        static instance: Game*
        deck: Deck<Card>
        player: unique_ptr<Human>
        dealer: unique_ptr<Dealer>
        stats: PlayerStats

    PUBLIC:
        static get(): Game& // Single instance access
        start()

    PRIVATE:
        init()
        loadGame()
        newGame()
        mainLoop()
        round()
        deal()
        playerTurn()
        dealerTurn()
        endRound()
        getBet(): integer
        saveGame()
```

```
MAIN PROGRAM FLOW:
  BEGIN
    GET game instance

    TRY
      game.start()

      PROCEDURE start:
        CALL init()
          ASK user to load game
          IF load game THEN
            CALL loadGame()
          ELSE
            CALL newGame()

        CALL mainLoop()
          LOOP until exit
            CALL round()
              GET bet from player
              IF valid bet THEN
                deal initial cards
                player's turn
                IF not bust THEN
                  dealer's turn
                END IF
                determine winner
                update stats
              END IF

            save game
            IF funds <= 0 OR player quits THEN
              EXIT loop
            END IF
          END LOOP

        save final state

    CATCH any exceptions
      display error message
    END TRY
  END

END PROGRAM
```

FLOWCHART


Author: Ireoluwa Dairo
Created on December 1, 2024, 12:19 PM
Purpose: Blackjack
system libraries
Standard Namespace
user libraries
Global constants
Function prototypes
Main
load save game?
Yes
Load saved game
No
account = 0
Yes
Start new game
No
Players Turn, place bet based on account balance
Hit or stand
Hit
add card
Stand
No
Dealers Turn
is player bust ??
Yes
Dealer wins
Platyer hand = dealer hand
Yes
Draw
B
No
players hand> dealers hand
Yes
No
Player WIns
B
B
Update Game stats (update wins/losses/cash)
Save Game Data
Play Again??
Yes
No
Exit

Structure

The program implements an object-oriented structure using key classes and variables that manage the Blackjack game's functionality. The Game class serves as the central controller, using a deck variable (Deck<Card>) to handle cards, plyr and dlr unique pointers to manage player and dealer objects, user string for player identification, and save struct for game state persistence. The Player class hierarchy uses protected variables including name (string), hand (vector<Card>), funds (int), and bet (int), which are inherited by both Human and Dealer classes. The Card class contains value (int) and suit (enum) variables, while the Deck template class employs vectors for both active and used cards. Additional tracking is handled by the PlayerStats class through variables like totalGames, gamesWon, totalProfit, and highestWin. This structure replaces Project 1's global variables with encapsulated class members, using smart pointers and STL containers for improved memory management and data organization.

Concepts

All concept from sections in Chapter 13 to Chapters 16 on the checklist

References

I made modifications to my Project 1

Challenges

Converting the code's comments to Doxygen format proved challenging and unsuccessful despite multiple attempts

Program

BLACKJACK.H

```
/**
 * @file BlackJackGame.h
 * @author Ireoluwa Dairo
 * @brief Main game controller class for Blackjack card game
 * @date 2024-11-27
 */

#ifndef BLACKJACK_H
#define BLACKJACK_H

#include "Deck.h"
#include "Player.h"
#include "SaveGame.h"
#include <memory>

using namespace std;

/**
 * @class Game
 * @brief Main controller class implementing Blackjack game logic
 *
 * This class implements the Singleton pattern and manages the complete
 * lifecycle of a Blackjack game including player turns, dealer actions,
 * betting, and game state persistence.
 */
class Game {
public:
    /**
     * @brief Get the single instance of the Game class
     * @return Game& Reference to the singleton game instance
     */
    static Game& get();

    /**
     * @brief Starts a new game session
     *
     * Initializes game components and begins the main game loop.
     */
    void start();

    // Delete copy/move operations to ensure singleton pattern
    Game(const Game&) = delete;
    Game& operator=(const Game&) = delete;

private:
    /**
     * @brief Private constructor for singleton pattern
     */
    Game();

    /**
     * @brief Destructor
     */
    ~Game();

    /**
     * @brief Initialize game components and settings
     */
    void init();

    /**
     * @brief Load a previously saved game state
     */
    void loadGame();

    /**
     * @brief Initialize and start a new game
     */
    void newGame();

    /**
     * @brief Execute the main game loop
     */
    void mainLoop();
```

```
    /**
     * @brief Execute one complete round of Blackjack
     */
    void round();

    /**
     * @brief Deal initial cards to player and dealer
     */
    void deal();

    /**
     * @brief Handle player's turn including hit/stand decisions
     */
    void plyrTurn();

    /**
     * @brief Handle dealer's turn according to house rules
     */
    void dlrTurn();

    /**
     * @brief Process end of round including determining winner and paying bets
     */
    void endRound();

    /**
     * @brief Get bet amount from player
     * @return int The amount bet by the player
     */
    int getBet();

    /**
     * @brief Save current game state
     */
    void saveGame();

    /** @brief Deck of cards used in the game */
    Deck<Card> deck;

    /** @brief Pointer to human player object */
    unique_ptr<Human> plyr;

    /** @brief Pointer to dealer object */
    unique_ptr<Dealer> dlr;

    /** @brief Current player's username */
    string user;

    /** @brief Save game data handler */
    Save save;

    /** @brief Pointer to singleton instance */
    static Game* inst;
};

#endif
```

CARD.H

```
/**
 * @file Card.h
 * @author Ireoluwa Dairo
 * @brief Playing card class implementation
 * @date 2024-11-27
 */

#ifndef CARD_H
#define CARD_H

#include <string>
#include <iostream>

/**
 * @class Card
 * @brief Represents a single playing card in a standard deck
 * This class implements a playing card with a value (1-13) and suit.
 * It includes functionality for comparing cards, converting to string
 * representations, and tracking the total number of cards created.
 */
class Card {
public:
```

```
/**
 * @brief Enumeration of card suits
 *
 * Uses enum class for type safety and scoping
 */
enum class Suit { HEARTS, DIAMONDS, CLUBS, SPADES };

/**
 * @brief Default constructor
 */
Card();

/**
 * @brief Constructor with value and suit
 * @param value Card's numeric value (1-13)
 * @param suit Card's suit
 * @throws std::invalid_argument if value is outside valid range
 */
Card(int value, Suit suit);

/**
 * @brief Copy constructor
 * @param other Card to copy from
 */
Card(const Card& other);

/**
 * @brief Destructor
 */
~Card();

/**
 * @brief Get card's numeric value
 * @return int Value between 1-13
 */
int getValue() const;

/**
 * @brief Get card's suit
 * @return Suit Enumerated suit value
 */
Suit getSuit() const;

/**
 * @brief Get string representation of card's suit
 * @return std::string Suit name as string
 */
std::string getSuitString() const;

/**
 * @brief Get string representation of card's value
 * @return std::string Card value as string (e.g., "Ace", "King", etc.)
 */
std::string getValueString() const;

/**
 * @brief Equality comparison operator
 * @param other Card to compare with
 * @return bool True if cards have same value and suit
 */
bool operator==(const Card& other) const;

/**
 * @brief Inequality comparison operator
 * @param other Card to compare with
 * @return bool True if cards differ in value or suit
 */
bool operator!=(const Card& other) const;

/**
 * @brief Assignment operator
 * @param other Card to assign from
 * @return Card& Reference to this card
 */
Card& operator=(const Card& other);

/**
 * @brief Get total number of Card objects created
 * @return int Number of cards created
 */
```

```
    static int getNumCardsCreated();

    /**
     * @brief Stream output operator
     * @param os Output stream
     * @param card Card to output
     * @return std::ostream& Reference to output stream
     */
    friend std::ostream& operator<<(std::ostream& os, const Card& card);

private:
    int value;    /**< Card's numeric value (1-13) */
    Suit suit;    /**< Card's suit */
    static int cardCount;  /**< Counter for total card instances */

    /**
     * @brief Validate card value is within legal range
     * @param val Value to validate
     * @throws std::invalid_argument if value is outside valid range
     */
    void validateValue(int val);
};

// Inline method implementations
inline int Card::getValue() const { return value; }
inline Card::Suit Card::getSuit() const { return suit; }

#endif // CARD_H
```

DECK.H

```
/**
 * @file Deck.h
 * @author Ireoluwa Dairo
 * @brief Template class for a card deck implementation
 * @date 2024-11-27
 */

#ifndef DECK_H
#define DECK_H

#include <vector>
#include <algorithm>
#include <random>
#include <stdexcept>
#include "Card.h"

using namespace std;

/**
 * @class Deck
 * @brief Template class representing a deck of cards
 * @tparam T Card type that comprises the deck (defaults to Card class)
 * Implements a deck of cards with shuffle, draw and tracking functionality.
 * Supports multiple deck initialization and duplicate card detection.
 */
template <typename T = Card>
class Deck {
public:
    /** @brief Default constructor. Initializes a single deck */
    Deck();

    /**
     * @brief Constructor for multiple decks
     * @param numDecks Number of decks to initialize
     */
    Deck(int numDecks);

    /**
     * @brief Copy constructor
     * @param other Deck to copy from
     */
    Deck(const Deck& other);

    /**
     * @brief Assignment operator
     * @param other Deck to assign from
     * @return Deck& Reference to this deck
     */
    Deck& operator=(const Deck& other);
```

```
    /** @brief Destructor */
    ~Deck();

    /**
     * @brief Shuffle the deck. Randomizes card order and clears dealt cards tracking
     */
    void shuffle();

    /**
     * @brief Draw a card from the deck
     * @return T The drawn card
     * @throws runtime_error if deck is empty
     * @note Automatically reinitializes deck if duplicate card detected
     */
    T drawCard();

    /**
     * @brief Check if deck is empty
     * @return bool True if no cards remain
     */
    bool isEmpty() const;

    /**
     * @brief Get number of remaining cards
     * @return int Number of undealt cards
     */
    int remainingCards() const;

    /**
     * @brief Array subscript operator
     * @param index Index of card to access
     * @return T Card at specified index
     * @throws out_of_range if index is invalid
     */
    T operator[](int index) const;

    /** @brief Friend class for debugging purposes */
    template <typename U>
    friend class DeckLogger;

protected:
    /**
     * @brief Initialize deck(s) with cards
     * @param numDecks Number of decks to initialize
     */
    void initializeDeck(int numDecks);

private:
    vector<T> cards;      /**< Vector of undealt cards */
    vector<T> dealtCards; /**< Vector tracking dealt cards */
    random_device rd;     /**< Random device for shuffling */
    mt19937 gen;          /**< Mersenne Twister random generator */
};

// Template implementation
template <typename T>
Deck<T>::Deck() : gen(rd()) {
    initializeDeck(1);  // Default to one deck
}

template <typename T>
Deck<T>::Deck(int numDecks) : gen(rd()) {
    initializeDeck(numDecks);
}

template <typename T>
void Deck<T>::initializeDeck(int numDecks) {
    cards.clear();
    dealtCards.clear();  // Clear dealt cards when reinitializing
    for (int deck = 0; deck < numDecks; ++deck) {
        for (int suit = 0; suit < 4; ++suit) {
            for (int value = 1; value <= 13; ++value) {
                cards.push_back(T(value, static_cast<typename T::Suit>(suit)));
            }
        }
    }
}

template <typename T>
void Deck<T>::shuffle() {
```

```
    std::random_shuffle(cards.begin(), cards.end());  // Changed to random_shuffle
    dealtCards.clear();  // Clear dealt cards tracking when shuffling
}

template <typename T>
T Deck<T>::drawCard() {
    if (isEmpty()) {
        cout << "Deck is empty!" << endl;
        return T();
    }

    T card = cards.back();
    cards.pop_back();

    // Check for duplicates
    for (const auto& dealtCard : dealtCards) {
        if (card == dealtCard) {
            // Reinitialize deck if duplicate found
            initializeDeck(6);
            shuffle();
            return drawCard();
        }
    }

    dealtCards.push_back(card);
    return card;
}

template <typename T>
bool Deck<T>::isEmpty() const {
    return cards.empty();
}

template <typename T>
int Deck<T>::remainingCards() const {
    return cards.size();
}

// Copy constructor
template <typename T>
Deck<T>::Deck(const Deck& other) : gen(rd()) {
    cards = other.cards;
    dealtCards = other.dealtCards;
}

// Assignment operator
template <typename T>
Deck<T>& Deck<T>::operator=(const Deck& other) {
    if (this != &other) {
        cards = other.cards;
        dealtCards = other.dealtCards;
    }
    return *this;
}

// Destructor
template <typename T>
Deck<T>::~Deck() {
    cards.clear();
    dealtCards.clear();
}

// Subscript operator
template <typename T>
T Deck<T>::operator[](int index) const {
    if (index < 0 || index >= cards.size()) {
        cout << "Index out of range" << endl;
        return T();
    }
    return cards[index];
}
#endif // DECK_H
```

PLAYER.H

```
/**
* @file Player.h
* @author Ireoluwa Dairo
* @brief Base player class and derived player types for blackjack
* @date 2024-11-27
*/
```

```
#ifndef PLAYER_H
#define PLAYER_H

#include <string>
#include <vector>
#include "Card.h"
#include "PlayerStats.h"

using namespace std;

/**
* @class Player
* @brief Abstract base class for blackjack players
* Defines common functionality for managing hands, bets, funds and statistics
*/
class Player {
public:
  /** @brief Default constructor */
  Player();

  /**
   * @brief Constructor with name and initial funds
   * @param name Player's name
   * @param funds Initial funds (defaults to 1000)
   */
  Player(const string& name, int funds = 1000);

  /** @brief Virtual destructor */
  virtual ~Player();

  /**
   * @brief Pure virtual method for hit/stand decision
   * @return bool True if player wants another card
   */
  virtual bool wantHit() = 0;

  /** @brief Pure virtual method to display player's hand */
  virtual void dispHand() const = 0;

  /**
   * @brief Add card to player's hand
   * @param card Card to add
   */
  virtual void addCard(const Card& card);

  /**
   * @brief Calculate hand value
   * @return int Total value of cards in hand
   */
  virtual int getVal() const;

  /**
   * @brief Check if hand value exceeds 21
   * @return bool True if player has bust
   */
  virtual bool isBust() const;

  /** @brief Clear player's hand */
  virtual void clear();

  /**
   * @brief Set bet amount
   * @param amt Bet amount
   * @return bool True if bet is valid and set
   */
  bool setBet(int amt);

  /**
   * @brief Add winnings to funds
   * @param amt Amount won
   */
  void addWin(int amt);

  /**
   * @brief Deduct lost bet from funds
   * @param amt Amount lost
   */
  void loseBet(int amt);
```

```
  /**
   * @brief Set player's funds
   * @param amt New fund amount
   */
  void setFund(int amt);

  /**
   * @brief Record game result in stats
   * @param won Whether game was won
   * @param amt Amount won/lost
   */
  void addGame(bool won, int amt);

  /** @brief Get win rate from stats
   * @return int Win percentage
   */
  int getRate() const;

  /**
   * @brief Get player statistics
   * @return PStats& Reference to player stats
   */
  PStats& getStat();

  /** @brief Get player's name
   * @return string Player name
   */
  string getName() const;

  /** @brief Get current funds
   * @return int Available funds
   */
  int getFund() const;

  /** @brief Get current bet
   * @return int Current bet amount
   */
  int getBet() const;

  /**
   * @brief Equality comparison
   * @param other Player to compare with
   * @return bool True if players are equal
   */
  bool operator==(const Player& other) const;

  /**
   * @brief Inequality comparison
   * @param other Player to compare with
   * @return bool True if players are not equal
   */
  bool operator!=(const Player& other) const;

protected:
  string name;        /**< Player's name */
  vector<Card> hand;  /**< Current hand of cards */
  int funds;          /**< Available funds */
  int bet;          /**< Current bet amount */
  PStats stats;     /**< Player statistics */
};

/**
* @class Human
* @brief Human player class
* Implements interactive player decisions
*/
class Human : public Player {
public:
  /**
   * @brief Constructor
   * @param name Player's name
   */
  Human(const string& name);

  bool wantHit() override;
  void dispHand() const override;
};

/**
* @class Dealer
```

```
* @brief Dealer class
* Implements dealer-specific logic and display
*/
class Dealer : public Player {
public:
  /** @brief Constructor */
  Dealer();

  bool wantHit() override;
  void dispHand() const override;

  /** @brief Display all cards in dealer's hand */
  void showAll() const;
};

#endif
```

SAVEGAME.H

```
/**
* @file SaveGame.h
* @author Ireoluwa Dairo
* @brief Game state  implementation
* @date 2024-11-27
*/

#ifndef SAVEGAME_H
#define SAVEGAME_H

#include <string>
#include <fstream>
using namespace std;

/**
* @struct Save
* @brief Structure containing saveable game state data
*/
struct Save {
  char name[30];  /**< Player name */
  int funds;      /**< Current funds */
  int wins;       /**< Current session wins */
  int games;      /**< Current session games */
  int gmsAll;     /**< Total games played */
  int winsAll;    /**< Total games won */
};

/**
* @class SaveGame
* @brief Static class for managing game state persistence
*/
class SaveGame {
public:
  /**
   * @brief Load saved game data
   * @param data Save structure to load into
   * @param user Username to load data for
   * @return bool True if load successful
   */
  static bool load(Save& data, const string& user);

  /**
   * @brief Save current game data
   * @param data Save structure containing data to save
   * @param user Username to save data for
   * @return bool True if save successful
   */
  static bool save(const Save& data, const string& user);

  /**
   * @brief Check if save data exists for user
   * @param user Username to check
   * @return bool True if save data exists
   */
  static bool valid(const string& user);

private:
  /**
   * @brief Generate filename for user's save data
   * @param user Username
   * @return string Filename for save data
   */
```

```
  static string fname(const string& user);
};

#endif
```

BLACKJACKGAME.CPP

```
/**
* @file BlackJackGame.cpp
* @author Ireoluwa Dairo
* @brief Implementation of BlackJack game controller class
* @date 2024-11-27
*/

#include "BlackjackGame.h"
#include <iostream>
#include <cstring>
using namespace std;

/** @brief Initialize static singleton instance pointer to null */
Game* Game::inst = nullptr;

/**
* @brief Get single instance of Game
* @return Game& Reference to game instance
* @details Creates instance if it doesn't exist
*/
Game& Game::get() {
  if (!inst) {
    inst = new Game();
  }
  return *inst;
}

/**
* @brief Constructor
* Initializes game with 6 decks
*/
Game::Game() : deck(6) {
  init();
}

/**
* @brief Destructor
* Saves game state and cleans up singleton
*/
Game::~Game() {
  saveGame();
  delete inst;
  inst = nullptr;
}

/**
* @brief Start game session
* Initializes, runs main loop, and saves on exit
*/
void Game::start() {
  init();
  mainLoop();
  saveGame();
}

/**
* @brief Initialize game state
* Prompts for load/new game choice
*/
void Game::init() {
  char choice;
  do {
    cout << "Load game? (y/n): ";
    cin >> choice;
    cin.ignore();
  } while (choice != 'y' && choice != 'n');

  if (choice == 'y') {
    loadGame();
  } else {
    newGame();
  }
}
```

```
/**
* @brief Load saved game state
* Loads player data and handles zero funds case
*/
void Game::loadGame() {
    do {
        cout << "Enter username: ";
        getline(cin, user);
    } while (!SaveGame::valid(user));

    if (SaveGame::load(save, user)) {
        plyr = make_unique<Human>(save.name);
        plyr->setFund(save.funds);
        plyr->getStat().setAll(save.gmsAll, save.winsAll);
        dlr = make_unique<Dealer>();

        cout << "\nWelcome back " << save.name << "!\n";
        cout << "Funds: $" << save.funds << "\n";
        cout << "Wins: " << save.wins << "\n";
        cout << "Games: " << save.games << "\n";

        char show;
        cout << "\nView win rate? (y/n): ";
        cin >> show;
        cin.ignore();

        if (show == 'y' || show == 'Y') {
            cout << "Win Rate: " << plyr->getRate() << "%\n\n";
        }

        if (save.funds == 0) {
            char pick;
            cout << "\nYou have $0!\n";
            cout << "Start with $1000? (y/n): ";
            cin >> pick;
            cin.ignore();

            if (pick == 'y' || pick == 'Y') {
                plyr->setFund(1000);
                save.funds = 1000;
                saveGame();
                cout << "\nRefreshed! Balance: $1000\n";
            } else {
                cout << "\nNo funds. Game Over!\n";
                exit(0);
            }
        }
    } else {
        cout << "No save found.\n";
        newGame();
    }
}

/**
* @brief Create new game
* Sets up new player with initial funds
*/
void Game::newGame() {
    do {
        cout << "Enter new username: ";
        getline(cin, user);
    } while (!SaveGame::valid(user));

    strncpy(save.name, user.c_str(), 29);
    save.name[29] = '\0';
    save.funds = 1000;
    save.wins = 0;
    save.games = 0;
    save.gmsAll = 0;
    save.winsAll = 0;

    plyr = make_unique<Human>(user);
    dlr = make_unique<Dealer>();

    cout << "\nWelcome " << user << "!\n";
    cout << "Starting funds: $" << save.funds << "\n\n";
}

/**
* @brief Deal initial cards
```

```
* Deals two cards each to player and dealer
*/
void Game::deal() {
    plyr->addCard(deck.drawCard());
    dlr->addCard(deck.drawCard());
    plyr->addCard(deck.drawCard());
    dlr->addCard(deck.drawCard());

    plyr->dispHand();
    dlr->dispHand();
}

/**
* @brief Handle player's turn
* Allows player to hit until stand or bust
*/
void Game::plyrTurn() {
    while (!plyr->isBust() && plyr->wantHit()) {
        plyr->addCard(deck.drawCard());
        plyr->dispHand();

        if (plyr->isBust()) {
            cout << "\nBust!\n";
        }
    }
}

/**
* @brief Handle dealer's turn
* Executes dealer's play according to house rules
*/
void Game::dlrTurn() {
    dlr->showAll();

    while (dlr->wantHit()) {
        dlr->addCard(deck.drawCard());
        dlr->showAll();

        if (dlr->isBust()) {
            cout << "\nDealer busts!\n";
        }
    }
}

/**
* @brief Process end of round
* Determines winner and updates statistics
*/
void Game::endRound() {
    dlr->showAll();

    int pVal = plyr->getVal();
    int dVal = dlr->getVal();
    int bet = plyr->getBet();

    if (plyr->isBust()) {
        cout << "\nBust!\n";
        plyr->addGame(false, bet);
        save.games++;
        return;
    }

    if (dlr->isBust()) {
        cout << "\nDealer busts!\n";
        plyr->addWin(bet * 2);
        plyr->addGame(true, bet);
        save.wins++;
        save.games++;
        return;
    }

    if (pVal > dVal) {
        cout << "\nYou win!\n";
        plyr->addWin(bet * 2);
        plyr->addGame(true, bet);
        save.wins++;
    } else if (pVal < dVal) {
        cout << "\nYou lose!\n";
        plyr->addGame(false, bet);
    } else {
```

```
        cout << "\nPush!\n";
        plyr->addWin(bet);
        plyr->addGame(false, 0);
    }
    save.games++;

    Stats st = plyr->getStat().getAll();
    save.gmsAll = st.games;
    save.winsAll = st.wins;
}

/**
* @brief Execute one complete round
* Handles betting, dealing and turn sequence
*/
void Game::round() {
    plyr->clear();
    dlr->clear();

    int bet = getBet();
    if (!plyr->setBet(bet)) {
        cout << "Invalid bet!\n";
        return;
    }

    deck.shuffle();
    deal();
    plyrTurn();

    if (!plyr->isBust()) {
        dlrTurn();
    }

    endRound();
}

/**
* @brief Save current game state
* Updates and persists save data
*/
void Game::saveGame() {
    save.funds = plyr->getFund();
    SaveGame::save(save, user);
}

/**
* @brief Get bet amount from player
* @return int Valid bet amount
*/
int Game::getBet() {
    int bet;
    do {
        cout << "\nFunds: $" << plyr->getFund()
             << "\nBet amount: $";
        cin >> bet;

        if (bet > plyr->getFund()) {
            cout << "Can't bet more than you have!\n";
        }
        if (bet < 1) {
            cout << "Min bet is $1\n";
        }
    } while (bet > plyr->getFund() || bet < 1);

    return bet;
}

/**
* @brief Main game loop
* Handles round sequence and continuation
*/
void Game::mainLoop() {
    while (true) {
        round();
        saveGame();

        if (save.funds == 0) {
            char choice;
            cout << "\nYou're broke!\n";
            cout << "Start fresh with $1000? (y/n): ";
```

```
            cin >> choice;

            if (choice == 'y' || choice == 'Y') {
                plyr->setFund(1000);
                save.funds = 1000;
                saveGame();
                cout << "\nRefreshed! Balance: $1000\n";
            } else {
                cout << "\nGame Over!\n";
                break;
            }
        }

        char cont;
        cout << "\nPlay again? (y/n): ";
        cin >> cont;
        if (cont != 'y' && cont != 'Y') break;

        if (deck.remainingCards() < 15) {
            deck = Deck<Card>(6);
        }
    }
}
```

CARD.CPP

```
/**
* @file Card.cpp
* @author Ireoluwa Dairo
* @brief Implementation of Card class
* @date 2024-11-27
*/

#include "Card.h"

using namespace std;

/** Initialize static counter for card instances */
int Card::cardCount = 0;

/**
* @brief Default constructor
* Creates Ace of Hearts by default
*/
Card::Card() : value(1), suit(Suit::HEARTS) {
    cardCount++;
}

/**
* @brief Parameterized constructor
* @param value Card value (1-13)
* @param suit Card suit
*/
Card::Card(int value, Suit suit) : suit(suit) {
    validateValue(value);
    this->value = value;
    cardCount++;
}

/**
* @brief Copy constructor
* @param other Card to copy from
*/
Card::Card(const Card& other) : value(other.value), suit(other.suit) {
    cardCount++;
}

/**
* @brief Destructor
* Decrements card count
*/
Card::~Card() {
    cardCount--;
}

/**
* @brief Validate card value is within legal range
* @param val Value to validate
* @throws invalid_argument if value outside 1-13 range
*/
void Card::validateValue(int val) {
```

```
    if (val < 1 || val > 13) {
        cout << "Invalid card value, Must be between 1 and 13.\n";
        return;
    }
}

/**
* @brief Get string representation of card suit
* @return string Name of suit
*/
string Card::getSuitString() const {
    switch(suit) {
        case Suit::HEARTS: return "Hearts";
        case Suit::DIAMONDS: return "Diamonds";
        case Suit::CLUBS: return "Clubs";
        case Suit::SPADES: return "Spades";
        default: return "Unknown";
    }
}

/**
* @brief Get string representation of card value
* @return string Card value or face card name
*/
string Card::getValueString() const {
    switch(value) {
        case 1: return "Ace";
        case 11: return "Jack";
        case 12: return "Queen";
        case 13: return "King";
        default: return to_string(value);
    }
}

/**
* @brief Equality comparison operator
* @param other Card to compare with
* @return bool True if cards match in value and suit
*/
bool Card::operator==(const Card& other) const {
    return (value == other.value) && (suit == other.suit);
}

/**
* @brief Inequality comparison operator
* @param other Card to compare with
* @return bool True if cards differ in value or suit
*/
bool Card::operator!=(const Card& other) const {
    return !(*this == other);
}

/**
* @brief Assignment operator
* @param other Card to assign from
* @return Card& Reference to this card
*/
Card& Card::operator=(const Card& other) {
    if (this != &other) {
        value = other.value;
        suit = other.suit;
    }
    return *this;
}

/**
* @brief Get total number of Card objects created
* @return int Number of cards currently in existence
*/
int Card::getNumCardsCreated() {
    return cardCount;
}

/**
* @brief Stream output operator
* @param os Output stream
* @param card Card to output
* @return ostream& Reference to output stream
*/
ostream& operator<<(ostream& os, const Card& card) {
```

```
  os << card.getValueString() << " of " << card.getSuitString();
  return os;
}
```

PLAYER.CPP

```
/**
* @file Player.cpp
* @author Ireoluwa Dairo
* @brief Implementation of Player base class and derived Human/Dealer classes
* @date 2024-11-27
*/

#include "Player.h"
#include <iostream>
using namespace std;

/** Base Player Class Implementation */

/**
* @brief Default constructor
* Initializes anonymous player with default values
*/
Player::Player() {
  name = "Anon";
  funds = 1000;
  bet = 0;
  stats = PStats(100);
}

/**
* @brief Parameterized constructor
* @param n Player name
* @param f Initial funds
*/
Player::Player(const string& n, int f) {
  name = n;
  funds = f;
  bet = 0;
  stats = PStats(100);
}

/**
* @brief Destructor
*/
Player::~Player() {
  clear();
}

/**
* @brief Add card to player's hand
* @param card Card to add
*/
void Player::addCard(const Card& card) {
  hand.push_back(card);
}

/**
* @brief Calculate hand value considering aces
* @return int Total hand value
*/
int Player::getVal() const {
  int total = 0;
  int aces = 0;

  // Calculate non-ace values first
  for (const auto& card : hand) {
    int val = card.getValue();
    if (val > 10) val = 10;
    if (val == 1) {
      aces++;
    } else {
      total += val;
    }
  }

  // Handle aces
  for (int i = 0; i < aces; ++i) {
    if (total + 11 <= 21) {
      total += 11;
    } else {
```

```
            total += 1;
        }
    }

    return total;
}

/**
* @brief Check if player has bust
* @return bool True if hand value exceeds 21
*/
bool Player::isBust() const {
    return getVal() > 21;
}

/**
* @brief Clear hand and bet
*/
void Player::clear() {
    hand.clear();
    bet = 0;
}

/**
* @brief Set bet amount
* @param amt Amount to bet
* @return bool True if bet is valid and set
*/
bool Player::setBet(int amt) {
    if (amt > funds || amt < 1) {
        return false;
    }
    bet = amt;
    funds -= amt;
    return true;
}

/**
* @brief Add winnings to funds
* @param amt Amount won
*/
void Player::addWin(int amt) {
    funds += amt;
}

/**
* @brief Handle lost bet
* @param amt Amount lost
*/
void Player::loseBet(int amt) {
    // Bet already deducted when placed
}

/**
* @brief Set player's funds
* @param amt New fund amount
*/
void Player::setFund(int amt) {
    funds = amt;
}

/**
* @brief Record game result
* @param won Whether game was won
* @param amt Amount won/lost
*/
void Player::addGame(bool won, int amt) {
    try {
        GameRes result(won, amt);
        stats.add(result);
    } catch (const runtime_error& e) {
        cerr << "Failed to record game result: " << e.what() << endl;
    }
}

/** Getter Methods */
int Player::getRate() const { return stats.getRate(); }
string Player::getName() const { return name; }
int Player::getFund() const { return funds; }
int Player::getBet() const { return bet; }
```

```
PStats& Player::getStat() { return stats; }

/**
* @brief Equality comparison operator
* @param other Player to compare with
*/
bool Player::operator==(const Player& other) const {
  return name == other.name && funds == other.funds;
}

/**
* @brief Inequality comparison operator
* @param other Player to compare with
*/
bool Player::operator!=(const Player& other) const {
  return !(*this == other);
}

/** Human Class Implementation */

/**
* @brief Human constructor
* @param n Player name
*/
Human::Human(const string& n) {
  name = n;
  funds = 1000;
  bet = 0;
  stats = PStats(100);
}

/**
* @brief Get hit decision from human player
* @return bool True if player wants to hit
*/
bool Human::wantHit() {
  char resp;
  cout << "Hit? (y/n): ";
  cin >> resp;
  return (resp == 'y' || resp == 'Y');
}

/**
* @brief Display human player's hand
*/
void Human::dispHand() const {
    cout << "\n---------------------------------------\n";
    cout << getName() << "'s HAND \n";
    cout << "---------------------------------------\n";
    for (size_t i = 0; i < hand.size(); ++i) {
        cout << "| " << hand[i] << " |\n";
    }
    cout << "---------------------------------------\n";
    cout << "Total: " << getVal() << "\n";
    cout << "Funds: $" << getFund() << "\n";  // Changed to getFund
    if (bet > 0) {
        cout << "Current Bet: $" << bet << "\n";
    }
    cout << "---------------------------------------\n";
}

/** Dealer Class Implementation */

/**
* @brief Dealer constructor
* Initializes dealer with default values
*/
Dealer::Dealer() {
  name = "Dealer";
  funds = 0;
  bet = 0;
  stats = PStats(100);
}

/**
* @brief Get dealer's hit decision based on rules
* @return bool True if dealer must hit (< 17)
*/
bool Dealer::wantHit() {
  return getVal() < 17;
```

```
}

/**
* @brief Display dealer's initial hand (one card hidden)
*/
void Dealer::dispHand() const {
    cout << "\n-----------------------------------------\n";
    cout << "DEALER'S HAND\n";
    cout << "-----------------------------------------\n";
    if (!hand.empty()) {
        cout << " " << hand[0] << " \n";
        cout << " [Hidden Card] \n";
    }
    cout << "-----------------------------------------\n";
}

/**
 * @brief Display dealer's complete hand
 * Shows all cards and total value
 */
void Dealer::showAll() const {
    cout << "\n-----------------------------------------\n";
    cout << "DEALER'S FULL HAND\n";
    cout << "-----------------------------------------\n";
    for (size_t i = 0; i < hand.size(); ++i) {
        cout << hand[i]<< " \n";
    }
    cout << "-----------------------------------------\n";
    cout << "Total: " << getVal() << "\n";
    cout << "-----------------------------------------\n";
}
```

PLAYERSTATS.CPP

```
/**
* @file PlayerStats.cpp
* @author Ireoluwa Dairo
* @brief Implementation of player statistics tracking
* @date 2024-11-27
*/

#include "PlayerStats.h"
#include <stdexcept>

/** Initialize static counter for statistics objects */
int PStats::num = 0;

/**
* @brief Constructor
* @param maxSz Maximum number of game results to store
* @throws runtime_error if memory allocation fails
*/
PStats::PStats(int maxSz) {
   try {
       res = new GameRes[maxSz];
       max = maxSz;
       size = 0;
       rate = 0;
       games = 0;
       wins = 0;
       num++;
   } catch (bad_alloc& ba) {
       throw runtime_error("Memory fail");
   }
}

/**
* @brief Copy constructor
* @param old PStats object to copy from
* @throws runtime_error if memory allocation fails
*/
PStats::PStats(const PStats& old) {
   try {
       res = new GameRes[old.max];
       max = old.max;
       size = old.size;
       rate = old.rate;
       games = old.games;
       wins = old.wins;

       for(int i = 0; i < size; i++) {
```

```
            res[i] = old.res[i];
        }
        num++;
    } catch (bad_alloc& ba) {
        throw runtime_error("Copy fail");
    }
}

/**
* @brief Destructor
* Frees allocated memory and decrements counter
*/
PStats::~PStats() {
    delete[] res;
    num--;
}

/**
* @brief Assignment operator
* @param rhs PStats to assign from
* @return PStats& Reference to this object
* @throws runtime_error if memory allocation fails
*/
PStats& PStats::operator=(const PStats& rhs) {
    if(this != &rhs) {
        try {
            delete[] res;
            res = new GameRes[rhs.max];
            max = rhs.max;
            size = rhs.size;
            rate = rhs.rate;
            games = rhs.games;
            wins = rhs.wins;

            for(int i = 0; i < size; i++) {
                res[i] = rhs.res[i];
            }
        } catch (bad_alloc& ba) {
            throw runtime_error("Set fail");
        }
    }
    return *this;
}

/**
* @brief Set total games and wins
* @param gms Total games
* @param wns Total wins
*/
void PStats::setAll(int gms, int wns) {
    games = gms;
    wins = wns;
    if(games > 0) {
        rate = (wins * 100) / games;
    } else {
        rate = 0;
    }
}

/**
* @brief Get games and wins stats
* @return Stats Structure containing games and wins
*/
Stats PStats::getAll() const {
    return {games, wins};
}

/**
* @brief Add new game result
* @param newRes Game result to add
* @throws runtime_error if results array is full
*/
void PStats::add(const GameRes& newRes) {
    if(size >= max) {
        throw runtime_error("Stats full");
    }

    res[size++] = newRes;
    games++;
    if(newRes.isWin()) {
```

```
        wins++;
    }

    if(games > 0) {
        rate = (wins * 100) / games;
    }
}

/**
* @brief Get win rate percentage
* @return int Win rate as percentage
*/
int PStats::getRate() const {
    return rate;
}

/**
* @brief Less than comparison operator
* @param rhs PStats to compare with
* @return bool True if this win rate is less than rhs
*/
bool PStats::operator<(const PStats& rhs) const {
    return rate < rhs.rate;
}

/**
* @brief Addition operator
* @param rhs PStats to add
* @return PStats Combined statistics
*/
PStats PStats::operator+(const PStats& rhs) const {
    PStats tmp(max + rhs.max);
    tmp.size = 0;
    tmp.rate = (rate + rhs.rate) / 2;

    for(int i = 0; i < size; i++) {
        tmp.res[tmp.size++] = res[i];
    }
    for(int i = 0; i < rhs.size; i++) {
        tmp.res[tmp.size++] = rhs.res[i];
    }

    return tmp;
}
```

SAVEGAME.CPP

```
/**
* @file SaveGame.cpp
* @author Ireoluwa Dairo
* @brief Implementation of game state persistence
* @date 2024-11-27
*/

#include "SaveGame.h"
#include <cctype>
using namespace std;

/**
* @brief Load saved game data for user
* @param data Save structure to load data into
* @param user Username to load data for
* @return bool True if load successful
*/
bool SaveGame::load(Save& data, const string& user) {
    ifstream file(fname(user), ios::binary);
    if (!file) return false;

    file.read(reinterpret_cast<char*>(&data), sizeof(Save));
    file.close();
    return true;
}

/**
* @brief Save current game state for user
* @param data Save structure containing data
* @param user Username to save data for
* @return bool True if save successful
*/
bool SaveGame::save(const Save& data, const string& user) {
    ofstream file(fname(user), ios::binary);
```

```
  if (!file) return false;

  file.write(reinterpret_cast<const char*>(&data), sizeof(Save));
  file.close();
  return true;
}

/**
* @brief Validate username format
* @param user Username to validate
* @return bool True if username is valid
* @details Username must be 3-30 characters, alphanumeric or underscore
*/
bool SaveGame::valid(const string& user) {
  if (user.length() < 3 || user.length() > 30) {
     return false;
  }

  for (char c : user) {
     if (!isalnum(c) && c != '_') {
        return false;
     }
  }
  return true;
}

/**
* @brief Generate filename for user's save data
* @param user Username
* @return string Filename with .dat extension
*/
string SaveGame::fname(const string& user) {
  return user + ".dat";
}
```

MAIN.CPP

```
/**
* @file main.cpp
* @author Ireoluwa Dairo
* @brief Main entry point for Blackjack game
* @date 2024-11-27
*/

#include <cstdlib>
#include "BlackjackGame.h"
#include <iostream>
using namespace std;

/**
* @brief Test function demonstrating friend functionality and exception handling
* @details Creates a game instance and runs a test session
*/
void runBlackjackTest() {
  try {
     Game& game = Game::get();
     game.start();
  }
  catch (const exception& e) {
     cerr << "Test failed: " << e.what() << endl;
  }
}

/**
* @brief Main program entry point
* @return int Exit status (0 for success, 1 for error)
*/
int main() {
  try {
     Game& game = Game::get();
     game.start();
  }
  catch (const exception& e) {
     cerr << "Game encountered an error: " << e.what() << endl;
     return 1;
  }

  return 0;
}
```