Interactive Casino

Krishoy Dey

Advisor: **Mr. Sahabzada Betab Badar**

Department of Computer Science and Information Technology
Jain (Deemed-to-be) University

This report is submitted for the course of
Programming in C (24BCA1C04)

Jain University, Bengaluru   December 2024

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this report entitled "Interactive Casino" are original and have not been submitted in whole or in part for consideration for any other degree or qualification or course in this or any other university. This report is the collective work of me and my team.

Krishoy Dey
USN: 24BCAR0101
Department of Computer Science and Information Technology,
Jain (Deemed-to-be) University, Bengaluru

Acknowledgements

I acknowledge my advisor and teacher Mr. Sahabzada Betab Badar's invaluable guidance and support throughout this project. My gratitude for his mentorship extends to all the Department of Computer Science and Information Technology faculty at Jain (Deemed-to-be) University, Bengaluru. Additionally, I am grateful to my friends, peers and team members for their constructive feedback and collaborative spirit during the brainstorming sessions. Their inputs have helped refine my ideas and improve the overall quality of this work. Finally, I also acknowledge the support of my family, friends, peers and team members whose encouragement and understanding has enabled me to focus on this project with dedication. This project would not have been possible without their support. Thank you to everyone who contributed to the success of this project.

Abstract

This project implements three Casino Games using C programming language, which provides an interactive platform for users to play the casino games: **Roulette**, **Blackjack**, **Slots** and **Coin Flip**. The C program is designed to simulate some essence of a casino environment, where players can place bets to win or lose based on the outcomes of random "luck-based" events and manage their balance throughout the session. Each game has its own set of rules and payout conditions. The first game is Roulette where players guess a number between 0 and 18, with the winnings determined by matching the randomly generated number. The second game is Blackjack in which players are dealt a card value and they compete against a dealer's card, aiming for the higher value with 21 as the max. The third game is Slots where players spin three virtual slots, with the rewards based on matching patterns. The fourth game is Coin Flip and as the name implies, players flip a virtual coin after betting on either Heads or Tails and win take double as rewards if they win or nothing if they lose. The project uses core and fundamental C programming concepts such as functions, pointers, loops, conditional statements, and random number generation to create an engaging experience for the user. The randomness in each of the games is achieved by seeding **the rand() function with srand(time(NULL)),** which ensures variability in results. This program also includes error handling for invalid inputs and ensures that bets do not exceed the player's current available balance. This project demonstrates how fundamental programming techniques can be used to create a functional and entertaining application for anyone interested in gaming.

Table of Contents

List of Figures

List of Tables

1. Introduction

The **Interactive Casino** is a C-based project designed to experience the thrill and unpredictability of a casino environment in a virtual setting. By integrating various classic casino games into a single platform, this program provides users with a unique and engaging experience that combines entertainment with elements of strategy and decision-making. The program features four distinct games: Roulette, Blackjack, Slots, and Coin Flip, each offering different rules and outcomes. Players start with an initial balance and participate in the games of their choice, betting their virtual money and testing their luck and decision-making skills. The core idea behind the Casino Game Simulator is to recreate the essence of real-world gambling within the confines of a safe, risk-free environment. This is achieved through the use of a command-line interface, which allows players to interact with the program in a straightforward and user-friendly manner. Each game is designed with specific winning and losing conditions, which are handled through logical decision-making and random number generation [1]. The unpredictability of outcomes is a key feature, made possible by seeding the random number generator using the system clock. This ensures fairness and prevents predictability in the gameplay. Beyond its entertainment value, the Casino Game Simulator serves as an educational tool for developers and programming enthusiasts. It showcases several fundamental programming concepts, including the use of functions to organize code, pointers for managing data dynamically, and control structures like loops and conditionals to guide program flow [2]. Input validation mechanisms have been implemented to handle incorrect or invalid user inputs gracefully, ensuring the program remains robust and error-free even under unexpected scenarios. The project also highlights the importance of dynamic resource management. Players can place bets on games, and their balance is updated based on the results of their decisions. This not only adds a layer of realism to the simulation but also introduces players to basic financial management concepts. For example, they need to strategize their betting amounts based on their current balance and the risk associated with each game. Moreover, this project demonstrates the potential of programming in creating real-world applications for entertainment. By simulating a safe gambling environment, it encourages responsible gaming while highlighting and blending creativity, functionality, logic and the practical applications of computer science concepts in the gambling industry.

1.1 Objectives

The primary objective of the Interactive Casino project is to recreate the excitement and strategic challenges of a real-world casino in a virtual, risk-free environment, combining entertainment with practical applications of programming concepts. This project aims to offer users an immersive experience by integrating multiple games, including Roulette, Blackjack, Slots, and Coin Flip, each with its own set of rules, mechanics, and payout structures. By allowing players to manage an initial balance and place bets on games of their choice, the simulator encourages strategic decision-making and introduces basic financial management skills. At its core, the project seeks to enhance the understanding of fundamental programming principles such as modularity, dynamic data handling, and random number generation, which is crucial for simulating the unpredictability of casino games [3]. Through the implementation of control structures like loops, conditions, and function calls, the program emphasizes structured problem-solving and logical thinking. Input validation is another critical aspect of the project, ensuring that the program remains robust and user-friendly by gracefully handling invalid inputs, such as incorrect game choices or betting amounts that exceed the player's balance. The simulator is designed to provide continuous gameplay, where users can repeatedly participate in different games until they decide to exit. This feature not only ensures replayability but also serves to maintain user engagement. By demonstrating the practical application of randomization through system time seeding, the program introduces an essential technique for creating fairness and unpredictability in games, mirroring the essence of real-world gambling. Additionally, the simulator aims to showcase the entertainment potential of programming, inspiring creativity and innovation in software development. Ultimately, the project aspires to combine learning and fun, illustrating how coding skills can be applied to develop interactive and meaningful applications. It provides a practical platform for reinforcing theoretical knowledge while demonstrating the versatility of programming in the gaming industry. Through the Casino Game Simulator, users can enjoy a compelling mix of luck, strategy, and education, making the project a good exercise for entertainment and the understanding of programming basics.

1.2 Features

1.2.1 Roulette

A guessing game where players bet on a number between 0 and 18. A random winning number is then generated and players with correct guesses are rewarded with a payout of 18 times their bet amount. It demonstrates luck and strategy with high stakes and high reward.

1.2.2 Blackjack

A card comparison game where players compete against a dealer to achieve a higher card value without exceeding 21. Players win their bet if their card value is higher than the dealer, lose if it's lower or a draw if it's equal. It blends luck and decision-making.

1.2.3 Slots

A slot machine simulation with three virtual reels, offering payouts based on matching patterns. Players win a jackpot for 3 matching numbers, half of their bet for 2 matching numbers or lose otherwise. It's a Bonafide high-risk high-reward gambling game.

1.2.4 Coin Flip

A coin flip is a simple but thrilling game where players bet either on "Heads" or "Tails". A virtual coin is flipped and the outcome is determined randomly. Players win double their bet if guessed correctly but lose otherwise. It is a very straightforward game for risk-takers.

This project not only serves as an exercise in programming but also highlights how simple logic, randomness, and user interaction can be combined to create a fun and engaging application. By using concepts of C, such as its standard library functions (*rand()*, *srand()*, etc.) and control structures, this project demonstrates how these tools can be utilized to build a complete system that is both engaging and fun for the user.

1.3 Contribution

As a team member of the Interactive Casino project, my main responsibility was designing and implementing the Slots game. This involved developing the logic for simulating a three-reel slot machine, ensuring an engaging and realistic user experience.

I started by generating three random slot values using *rand()* seeded with *srand(time(NULL))*, replicating the mechanics of a traditional slot machine. The game logic was structured to check for jackpots (three matching numbers), partial matches (two matching numbers), or losses (no matching numbers). These outcomes were evaluated using conditional statements, with corresponding balance updates for each scenario.

To ensure user clarity, I added detailed game rules and displayed the results of each spin, including the slot values and whether the player won or lost. Input validation was implemented to handle invalid bets, ensuring that players could not bet more than their available balance or use non-positive values. The dynamic balance updates were managed using a pointer, enabling real-time reflection of wins or losses.

By coding the Slots module, I contributed to the project's modular design and functionality. This game highlights key C programming concepts such as randomization, conditional logic, and user interaction, aligning with the educational and entertainment objectives of the Casino Game Simulator.

2. Why the Chosen Approach

2.1 Justification for Game Selection

The selection of the four games Roulette, Blackjack, Slots, and Coin Flip for the Casino Game Simulator project was primarily because of their popularity, simplicity, and ability to demonstrate diverse gameplay mechanics and programming concepts.

Roulette was chosen for its iconic status in casinos and its blend of luck and strategy. The game's mechanics of betting on numbers and generating a random winning number allowed us to showcase the use of random number generation (*rand()*) and conditional logic. Roulette also provides players with high-risk, high-reward scenarios, adding excitement and engagement to the simulation.

Blackjack was selected due to its balance of chance and skill, making it one of the most popular card games globally. Implementing Blackjack required the use of randomization for card values, conditional comparisons to determine outcomes, and dynamic balance updates. The game's rules, including the possibility of ties, provided an opportunity to demonstrate programming concepts like flow control and decision-making.

Slots offers a simple yet thrilling experience, making it a perfect addition to the simulator. The random generation of three reels simulates the essence of a traditional slot machine, while jackpot payouts and partial wins demonstrate probability and reward logic. Slots' fast-paced nature complements the other games, providing players with an enjoyable option.

Coin Flip was included for its simplicity and accessibility, making it ideal for beginners or players seeking a quick round. The binary outcome of Heads or Tails allowed us to implement straightforward randomization and betting logic while highlighting the fairness and unpredictability of chance-based games.

By combining these four games, the project offers a well-rounded casino experience that appeals to different player preferences. Each game contributes uniquely to the simulator, demonstrating a variety of programming techniques while ensuring entertainment and replayability.

2.2 Programming Techniques and Tools

I. **Random Number Generation**
The *rand()* function, seeded with *srand(time(NULL))*, is used to generate unpredictable outcomes [1], ensuring fairness in games like Roulette, Blackjack, Slots, and Coin Flip.

II. **Functions**
The program follows a modular design by defining separate functions for each game (e.g., *roulette*, *blackjack*, *slots*, *cf*) and other tasks like displaying the balance (*show_bal*). This improves code readability, reusability, and maintainability.

III. **Error Handling**
Input validation is extensively used to handle invalid bets (e.g., bets exceeding the balance or non-positive bets) and incorrect choices. This ensures the program remains robust and user-friendly.

IV. **Dynamic Memory Management**
The player's balance is managed dynamically, reflecting changes based on wins or losses, simulating real-world financial transactions.

V. **Control Structures**
Loops (*while* loop) are used to maintain continuous gameplay until the user chooses to exit. Conditional statements (*if*, *else if*, *switch*) are employed for decision-making and flow control based on user input and game logic.

VI. **User Input and Output**
Interactive features are implemented using *scanf* and *printf* to read user inputs and display game results, instructions, and balance updates, providing a seamless user experience.

VII. **Menu-Driven Interface**
A simple yet effective menu system is implemented to allow users to choose games and navigate the program. This structure enhances usability and engagement.

VIII.   **Game-Specific Rules Implementation**

Each game has its unique logic, such as card comparisons in Blackjack, number matching in Roulette, symbol alignment in Slots, and binary outcomes in Coin Flip. These are coded as per the rules of each game, demonstrating detailed and accurate programming. These techniques were chosen for their efficiency, simplicity, and alignment with the project's educational goals of demonstrating core programming concepts.

IX.   **Use of Pointers**

Pointers are used to pass the player's balance dynamically [4], enabling direct modification of the balance without returning values, showcasing an essential concept in C programming.

X.   **Scalability and Extensibility**

The modular nature of the code allows for easy addition of new games or features, ensuring the project is scalable and adaptable for future enhancements.

XI.   **Commenting and Documentation**

The code includes comments explaining key sections and logic, making it easier to understand, debug, and maintain, which is a best practice in software development.

XII.   **Logical Comparison for Outcomes**

Logical comparisons (e.g., *if (guess == winnumber), if (player_card > dealer_card)*) are used to determine game outcomes, helping to replicate real-life scenarios accurately.

XIII.   **Efficient Resource Management**

The program ensures resources like the player's balance and game states are efficiently managed, reflecting realistic gaming principles such as betting and payouts.

XIV.   **Educational Application**

By implementing a variety of games, the project demonstrates diverse applications of programming concepts, serving as an educational tool for understanding modular programming, control structures, and user interaction.

3. Code Components

The Interactive Casino is a thoughtfully designed program that showcases the principles of modular programming, user interaction, and dynamic data management in C while simulating a virtual casino experience. This chapter provides a detailed breakdown of the program's key components, offering insights into how various parts of the code work together to deliver an engaging gaming experience. By leveraging fundamental programming techniques such as random number generation, pointer manipulation, and structured control flow, the project effectively simulates four classic casino games: Roulette, Blackjack, Slots, and Coin Flip.

Each section of the code is purposefully crafted to handle specific aspects of the simulator, ranging from managing the user's balance to executing the unique rules of each game. The modular design ensures that every game is encapsulated in its own function, improving readability and reusability while minimizing code redundancy. The use of a menu-driven interface facilitates intuitive navigation, allowing users to seamlessly switch between games or exit the simulation. Additionally, robust error handling mechanisms are implemented throughout the program to validate inputs and prevent crashes, ensuring a smooth user experience.

This section of Code Components not only explains the logic behind individual components but also highlights their contributions to the overall functionality of the program. Code snippets are provided to illustrate the implementation of critical features, making it easier to understand how concepts such as randomization, dynamic memory management, and conditional logic are applied in practice. Through this detailed analysis, the Casino Game Simulator emerges as a compelling example of how programming skills can be harnessed to create interactive and meaningful applications.

3.1 Header Section
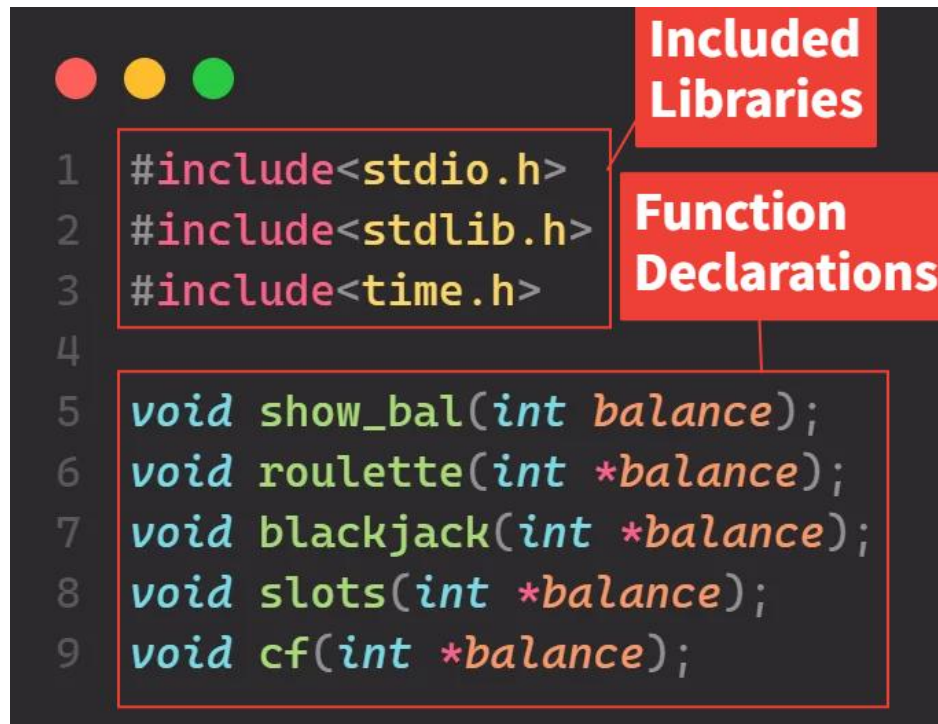
Figure 1: Header Section

3.1.1 Explanation of Header Section

Included Libraries

Three libraries were included in the C program to utilize functions like *printf* and *scanf* from stdio.h, *rand* and *srand* from stdlib.h and *time(NULL)* from time.h libraries respectively.

Function Declarations

Functions for displaying the player's balance (*show_bal*) and each of the games (*roulette*, *blackjack*, *slots* and *cf*) were declared so it increases reusability, readability and is easier to maintain and debug the C program.

3.2 Main Function


```c
int main()
{
    int balance, choice;

    srand(time(NULL));

    printf("Welcome to Casino!\n\n");
    printf("Enter your initial balance: ");
    scanf("%d", &balance);

    while (1)
    {
        printf("Current balance: %d\n", balance);
        printf("Choose a game!\n");
        printf("1. Roulette.\n");
        printf("2. Blackjack.\n");
        printf("3. Slots.\n");
        printf("4. Coin Flip\n");
        printf("5. Exit.\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            roulette(&balance);
            break;

        case 2:
            blackjack(&balance);
            break;

        case 3:
            slots(&balance);
            break;

        case 4:
            cf(&balance);
            break;

        case 5:
            printf("Thank you for playing, Your balance is: %d\n", balance);
            return 0;

        default:
            printf("Invalid choice, Please try again. \n\n");
        }
    }
}
```

Figure 2: Main Function

3.2.1 Explanation of Main Function

Main Function
The main function of integer datatype acts as an entry-point and defines the game menu (main body) of the C program [2].

Variable Declaration
Declares the balance and choice variables for storing the player's balance and game of choice from input.

Random Number Generator
srand(time(NULL)) seeds the random number generator. It basically means that it provides a changing seed value based on current time across different executions of the program ensuring randomness for each game.

Initial Setup
Displays welcome message to user using *printf* function and takes input for player's balance then stores it in balance variable's address using *scanf* function.

Game Menu Loop
Creates a loop for continuous gameplay using while loop. It shows the current balance and displays the game options using *printf* function then captures the user's game choice using *scanf* function in choice variable's address.

Game Selection
Based on the user's game choice, it calls the corresponding game function using *switch* statement and also passes balance by reference to update it after each game using the address of balance variable.

Exit Handling: *case 5* (Exit) handles program exit and displays the final balance using *printf* function while default case handles invalid inputs which is useful for error handling.

3.3 Roulette Function


```c
void roulette(int *balance)                          Function Header
{
    int guess, winnumber, bet;                       Variable Declaration

    printf("<---Roulette--->\n\n");
    printf("Rules for Roulette:\n");
    printf("1. Guessing the Right Number: The player chooses a number between 0 and 18. If the randomly gene
    printf("2. Betting and Losing: If the player's guess does not match the randomly generated number, they
    printf("3. Range of Possible Outcomes: The random number generated is between 0 and 18, inclusive, repre

    printf("Enter your bet: ");                       Player's Bet
    scanf("%d", &bet);

    if(bet > *balance || bet <= 0 )                  Input Validation for Bet
    {
        printf("Invalid bet amount. Please try again.\n\n");
        return;
    }

    printf("Choose a number from the range 0-18 : ");    Player's Guess
    scanf("%d", &guess);

    if (guess < 0 || guess > 18)                     Input Validation for Guess
    {
        printf("Invalid number, please try again.\n\n");
        return;
    }

    winnumber = rand() % 19;
    printf("Winning number is: %d\n", winnumber);

    if (guess == winnumber)
    {
        printf("Congratulations!! You win! \n");
        *balance = *balance + bet*18;                Winning Logic
    }

    else
    {
        printf("You lose! Better luck next time!\n");
        *balance = *balance - bet;
    }

    show_bal(*balance);                              show_bal function call
}
```
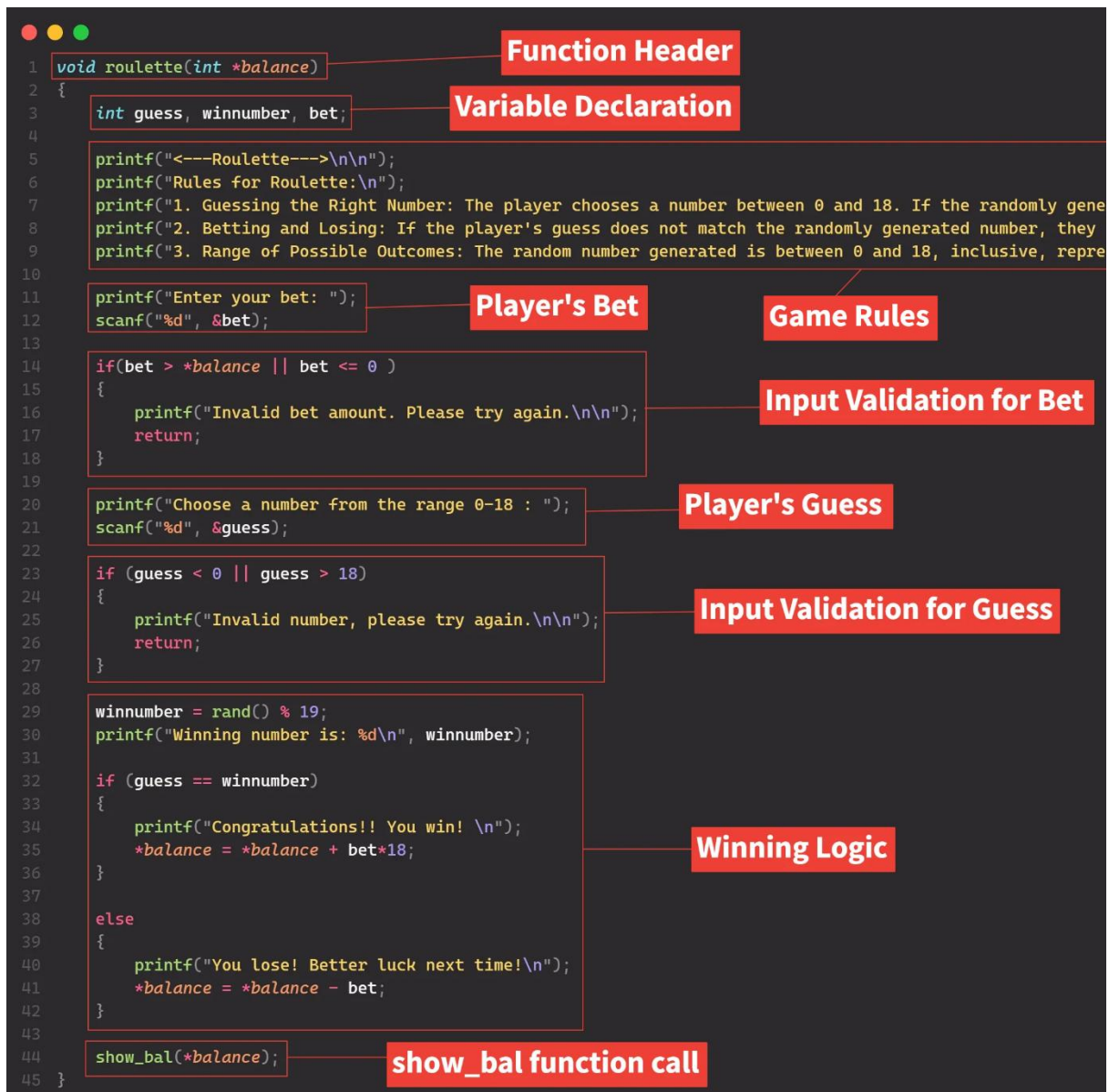
Figure 3: Roulette Function

3.3.1 Explanation of Roulette Function

Function Header

Defines roulette game function with *void* return type and a parameter of a *pointer* to the balance variable.

Variable Declaration

Declares the guess, winnumber and bet variables of integer datatype for storing the player's guessed number, winning number and bet amount respectively.

Game Rules

Prints the rules of the Roulette game using *printf* function.

Player's Bet

Takes input from user for bet amount from user and stores value in the bet variable memory address using *scanf* function.

Player's Guess

Takes input from user for the number to bet on and stores value in the guess variable's address using *scanf* function.

Input Validations for Bet and Guess

Handles invalid inputs by validating whether the bet amount is not zero or higher than player's available balance using *pointer* to balance variable and whether the guessed number is within the 0-18 range using *if* conditional statements and logical operators.

Winning Logic

Checks whether the guessed number matches the winning number using *if else* conditional statements. If guess matches winning number, player wins 18x the bet amount. If guess is wrong, player loses the bet amount. It uses *rand() % 19* to generate random winning number. It also updates balance accordingly using *pointer* arithmetic.

Function Call

Calls the *show_bal* function which displays player's updated balance once game is over.

3.4 Blackjack Function

```
1   void blackjack(int *balance)                          Function Header
2   {
3       int player_card, dealer_card, bet;                Variable Declaration
4
5       printf("<---BlackJack--->\n\n");
6       printf("Rules for BlackJack:\n");
7       printf("1. Betting Rule: The player must place a bet before each round, and t
8       printf("2. Card Values: Both the player and the dealer are randomly assigned
9       printf("3. Winning Rule: If the player's card value is greater than the deale
10      printf("4. Losing Rule: If the dealer's card value is greater than the player
11      printf("5. Draw Rule: If the player and dealer have the same card value, the
12                                                         Game Rules
13      printf("Enter your bet: ");            Player's Bet
14      scanf("%d", &bet);
15
16      if (bet > *balance || bet <= 0)
17      {
18          printf("Invalid bet amount. Please try again.\n\n");
19          return;                                        Input
20      }                                                  Validation
21                                                         for Bet
22      player_card = rand() % 21+1;
23      dealer_card = rand() % 21+1;
24
25      printf("Your card: %d\n", player_card);
26      printf("Dealer's card: %d\n", dealer_card);
27
28      if (player_card > dealer_card)
29      {
30          printf("Congratulations! You win!\n");         Winning Logic
31          *balance = *balance + bet;
32      }
33      else if (player_card < dealer_card)
34      {
35          printf("You Lose. Better luck next time!\n");
36          *balance = *balance - bet;
37      }
38      else
39      {
40          printf("It's a draw! no gain or loss\n");
41      }
42
43      show_bal(*balance);               show_bal Function Call
44  }
```

Figure 4: Blackjack Function

3.4.1 Explanation of Blackjack Function

Function Header
Defines blackjack game function with a parameter of a *pointer* to the balance variable.

Variable Declaration
Declares the player_card, dealer_card and bet variables of integer datatype for storing the player's cards' value, dealer's cards' value and player's bet amount respectively.

Game Rules
Prints the rules of the Blackjack game using *printf* function so player has a basic understanding of the game.

Player's Bet
Takes input from user for bet amount from user and stores value in the bet variable's address using *scanf* function.

Input Validation for Bet
Handles invalid inputs by checking whether the bet amount is not zero or higher than player's available balance using *if* conditional statement with logical operators and by using *pointer* to balance variable.

Winning Logic
Generates random numbers for both player's and dealer's cards between 1 to 21 using *rand()* function seeded with *time(NULL)* then compares the two values using *if, else if and else* conditional statements and rational operators and updates the player's balance accordingly using *pointer* arithmetic.

Function Call
Calls the *show_bal* function to display player's updated balance once game is over.

3.5   Slots Function


```c
void slots(int *balance)                          Function Header
{
    int bet, s1, s2, s3;                          Variable Declaration

    printf("<---SLOTS--->\n\n");
    printf("Rules for Slots.\n");
    printf("1. Betting Rule: The player must place a bet before spinning the slot
    printf("2. Slot Spin: Three random values are generated for the three slots
    printf("3. Jackpot Rule: If all three slots match (e.g., 3 | 3 | 3), the pla
    printf("4. Partial Match Rule: If any two slots match (e.g., 3 | 3 | 1), the
    printf("5. No Match Rule: If none of the slots match (e.g., 3 | 1 | 2), the p

    printf("Enter your bet: ");                    Player's Bet      Game Rules
    scanf("%d", &bet);

    if (bet > *balance || bet <= 0)
    {
        printf("Invalid bet amount. Please try again.\n\n");
        return;                                    Input
    }                                              Validation
                                                   for Bet
    s1 = rand() % 11;
    s2 = rand() % 11;
    s3 = rand() % 11;

    printf("Slot Results: %d | %d | %d\n", s1,s2,s3);

    if (s1 == s2 && s2 == s3)
    {
        printf("Congratulations! You won a JACKPOT!!!\n");
        *balance = *balance + bet * 10;
    }
    else if (s1 == s2 || s2 == s3 || s1 == s3)     Winning Logic
    {
        printf("Congrats! You win half your bet!\n");
        *balance = *balance + bet / 2;
    }
    else
    {
        printf("You lose. Better luck next time!\n");
        *balance = *balance - bet;
    }

    show_bal(*balance);                            show_bal Function Call
}
```

Figure 5: Slots Function

3.5.1 Explanation of Slots Function

Function Header

Defines slots game function with a parameter of a *pointer* to the balance variable.

Variable Declaration

Declares the bet, s1, s2 and s3 variables of integer datatype for storing the player's bet amount and number generated after spin in each slot (slot 1, slot 2, slot 3) of the slot machine respectively.

Game Rules

Prints the rules of the Slots game using *printf* function so player has a basic understanding of the game.

Player's Bet

Takes input from user for bet amount from user and stores value in the bet variable's address using *scanf* function.

Input Validation for Bet

Handles invalid inputs by checking whether the bet amount is not zero or higher than player's available balance using *if* conditional statement with logical operators and by using *pointer* to balance variable.

Winning Logic

Generates random numbers for each of the three slots reels between 0 to 10 using *rand()* function seeded with *time(NULL)* then checks whether it's a jackpot (all 3 slots have same number), a partial win (2 slots have same number) or a loss (none of the slots have same number) using *if, else if and else* conditional statements and rational operators and updates the player's balance accordingly using *pointer* arithmetic.

Function Call

Calls the *show_bal* function to display player's updated balance once game is over.

3.6 Coin Flip Function


```c
void cf(int *balance)
{
    int bet, choice, result;

    printf("<---Coin Flip--->\n\n");
    printf("Rules for Coin Flip:\n");
    printf("1. The player bets on either 'Heads' or 'Tails'. \n");
    printf("2. A coin is flipped, and the outcome is randomly determined. \n");
    printf("3. If the player's guess matches the outcome, they win ddouble their bet.\n");
    printf("4. If the player's guess does not match, they lose the bet.\n\n");

    printf("Enter your bet: ");
    scanf("%d", &bet);

    if (bet > *balance || bet <= 0)
    {
        printf("Invalid bet amount. Please try again. \n\n");
        return;
    }

    printf("Choose your bet:\n");
    printf("1. Heads\n");
    printf("2. Tails\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);


    if(choice != 1 && choice != 2)
    {
        printf("Invalid choice! Please slect 1 for Heads or 2 for Tails. \n\n");
        return;
    }

    result = rand() % 2+1;

    if(result == 1)
    {
        printf("Coin Flip result: Heads\n");
    }
    else
    {
        printf("Coin Flip result: Tails\n");
    }

    if(choice == result)
    {
        printf("Congratulations! You win!\n");
        *balance = *balance + bet;
    }
    else
    {
        printf("You lose! Better luck next time.\n");
        *balance = *balance - bet;
    }

    show_bal(*balance);

}
```

Figure 6: Coin Flip Function

3.6.1 Explanation of Coin Flip Function

Function Header
Defines cf (coin flip) game function with *void* return type and a parameter of a *pointer* to the balance variable.

Variable Declaration
Declares the bet, choice and result variables of integer datatype for storing the player's bet amount, player's choice of heads or tails and game result respectively.

Game Rules
Prints the rules of the Coin Flip game using *printf* function.

Player's Bet
Takes input from user for bet amount from user and stores value in the bet variable memory address using *scanf* function.

Player's Choice
Takes input from user for the coin side to bet on then stores value 1 for heads and 2 for tails in the choice variable's address using *scanf* function.

Input Validations for Bet and Choice
Handles invalid inputs by validating whether the bet amount is not zero or higher than player's available balance using *pointer* to balance variable and whether the chosen value is either 1 or 2 using *if* conditional statements and logical operators.
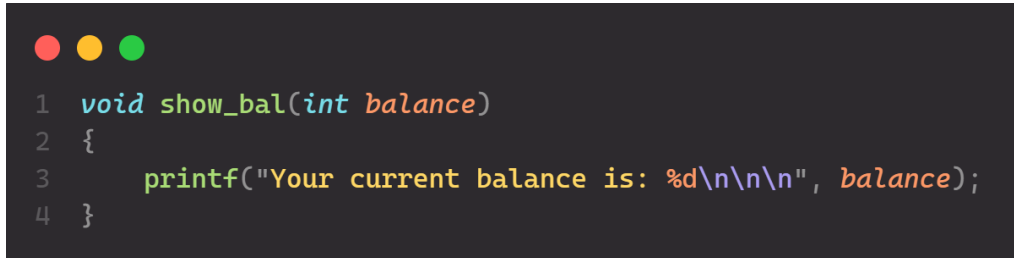
Winning Logic
Checks whether the chosen coin side matches the winning side using *if else* conditional statements. If player choice matches winning side, player wins double the bet amount. If choice is wrong, player loses the bet amount. It uses *rand()* to generate random winning side (0 or 1). It also updates balance accordingly using *pointer* arithmetic.

Function Call
Calls the *show_bal* function which displays player's updated balance once game is over.

3.7   Show Balance Function

```c
void show_bal(int balance)
{
    printf("Your current balance is: %d\n\n\n", balance);
}
```

Figure 7: Show Balance Function

3.7.1   Explanation of Show Balance Function

The player's balance is passed as a *pointer* to each game function and this allows modifying the balance directly without returning values. Using pointers eliminates the need for global variables and the dynamic balance updates reflect real-time gains or losses.

3.8   Summary

The Interactive Casino's code components are built with a modular and structured approach. Essential libraries (stdio.h, stdlib.h, time.h) include library functions for input/output, randomization, and time-based seeding for unpredictable outcomes. Function declarations define a clear structure, with each game—Roulette, Blackjack, Slots, and Coin Flip encapsulated in its own function. The program also uses pointers for dynamic balance management and a menu-driven interface for user interaction. Randomization ensures fairness, while input validation handles errors or invalid inputs. This design highlights key C programming concepts like modularity, control structures, and dynamic memory management, creating a functional, user-friendly, and interesting project.

4. Comparison of Techniques

Feature	Current Implementation	Alternatives	Why Current Implementation is Better
Random Number Generation	`rand()` with `srand(time(NULL))`	Use `random()` or `drand48()` for better randomness and uniformity.	`rand()` is part of the standard C library, ensuring portability across platforms. Alternatives like `random()` or `drand48()` are POSIX-specific, limiting compatibility on non-POSIX systems such as Windows. Additionally, `rand()` was sufficient for this project.
Game Logics	Modular functions for each game	Use function pointers or arrays of function pointers for dynamic game selection.	Modular functions improve readability and maintainability by keeping each game's logic separate. Function pointers add complexity and are unnecessary for a fixed set of games, also making it easier to debug.
Input Validation	Manual checks using if condition	Use macros or custom validation functions for reusable and centralized error handling.	Manual checks are simple to implement and directly integrated into the game logic. While macros or centralized validation functions could reduce redundancy, they may overcomplicate the code for a small scale project like this one.

Table 1: Comparison of Techniques

5. Data Description and Analysis

5.1   Types of Data Used

The project primarily uses the following types of data:

❖ **Player Balance:** An integer variable (balance) tracks the player's virtual currency, dynamically updated after each game based on wins or losses. It represents the central resource managed by the player throughout the simulation.

❖ **Bet Amount:** An integer variable (bet) is input by the player for each game. This value is validated to ensure it is positive and does not exceed the current balance, preventing invalid transactions.

❖ **Game-Specific Data:**

➢ **Roulette:** Player's guessed number (guess) and a randomly generated winning number (winnumber).

➢ **Blackjack:** Randomly generated card values for the player (player_card) and dealer (dealer_card).

➢ **Slots:** Three random slot values (s1, s2, s3) simulate a traditional slot machine and determine matches.

➢ **Coin Flip:** The player's choice (choice) of heads or tails is compared with the randomly determined outcome of the coin flip (result).

The Interactive Casino relies on a variety of data types to handle user interactions, gameplay mechanics and outcomes. And some of these data types listed above work together to create an interactive and realistic simulation of a casino environment, handling user input, game logic, and probabilistic outcomes.

5.2   Insights Derived from Data and Statistical Outcomes

The data used in the project helps analyze probabilities and statistical patterns across the games, offering insights into their risk-reward dynamics:

❖ **Roulette:** The probability of winning is $\frac{1}{19}$ (5.26%), that is high risk but high reward (18x payout).

❖ **Blackjack:** The outcomes depend on card values, with a roughly equal chance of winning, losing, or drawing due to uniform random distribution between 1 and 21.

❖ **Slots:** The probabilities vary:

  ➢ Jackpot (all three slots match): $\frac{1}{11^2}$ = 0.826% making it a rare but highly rewarding event with a 10x payout.
  ➢ Partial match (two slots match): This outcome is more probable due to multiple combinations, offering a smaller but consistent reward (half the bet).
  ➢ No match: The most frequent outcome aligns with the house advantage in traditional slot machines.

❖ **Coin Flip:** The probability of winning in Coin Flip is $\frac{1}{2}$ (50%), making it the simplest and most balanced game in the simulator. With even odds and a 2x payout, it appeals to players seeking quick results with moderate risk.

These insights provide similarity to real-world casino dynamics, where games are designed to balance excitement and fairness while maintaining a statistical edge for the house to win more frequently in comparison with the player. The data highlights the importance of probability in shaping player decisions and game design, demonstrating the connection between risk and reward in a controlled and risk-free environment.

6. Technical Description

6.1   Random Number Generation: Concept and Usage

The project utilizes the *rand()* function from the <stdlib.h> library to generate random numbers [5], a critical component for ensuring unpredictability in game outcomes. To prevent repeated patterns of random numbers across multiple executions, the generator is seeded with the system clock using *srand(time(NULL))*. The *time(NULL)* function, sourced from the <time.h> library, returns the number of seconds elapsed since the Unix epoch (January 1, 1970). This seeding ensures that each game session starts with a unique sequence of random numbers, simulating the unpredictability found in real casino scenarios.

Examples of Usage:

- ❖ **Roulette:** The winning number is generated using *rand() % 19*, which produces a random integer in the range [0, 18], simulating the outcomes of a Roulette spin.
- ❖ **Blackjack:** Card values for both the player and the dealer are determined using *rand() % 21 + 1*, generating values in the range [1, 21] to replicate the randomness of drawing cards.
- ❖ **Slots:** The three slot values are generated independently using *rand() % 11*, producing random integers in the range [0, 10], mimicking mechanics from traditional slot machines.
- ❖ **Coin Flip:** The coin's result is simulated with *rand() % 2 + 1*, outputting a binary outcome (1 for Heads, 2 for Tails).

The use of *srand(time(NULL))* ensures that the sequence of numbers generated by *rand()* varies each time the program is executed. Without this, the random number generator would produce the same sequence on every run, reducing unpredictability.

This implementation of random number generation forms the technical foundation of the simulator, enhancing both the gameplay experience and the educational value of the project by demonstrating real-world applications of randomization in programming.

6.2 Input Validation and Error Handling

To enhance user experience and maintain program robustness, input validation and error handling are implemented throughout the simulator.

6.2.1 Key Features of Input Validation

Bet Validation: The program ensures that bets are positive integers and do not exceed the player's current balance. Invalid bets trigger error messages, prompting the user to try again.

Game-Specific Validation: Inputs are checked against the acceptable ranges for each game.

- ❖ In Roulette, the guessed number must be between 0 and 18.
- ❖ In Coin Flip, the choice must be either 1 (Heads) or 2 (Tails).
- ❖ Other games follow similar input validation rules to ensure fairness.

6.2.2 Error Handling Mechanism

When an invalid input is detected, the program displays a clear error message without disrupting the overall gameplay. This ensures the program remains user-friendly and prevents crashes or unintended behaviour. For example, if the user attempts to bet more than their current balance, the program displays:

```
Invalid bet amount. Please try again.
```

Figure 8: Balance Error (Example)

By implementing these mechanisms, the program ensures smooth gameplay and an error-free experience for the user, even with incorrect user inputs.

7. Time and Space Complexity

7.1 Roulette

- ❖ **Time Complexity**: $O(1)$ (constant operations per round, such as generating a random number and comparing it)
- ❖ **Space Complexity**: $O(1)$ (no additional memory usage except local variables)

7.2 Blackjack

- ❖ **Time Complexity**: $O(1)$ (constant operations, including random number generation and comparison)
- ❖ **Space Complexity**: $O(1)$ (no additional memory usage except local variables)

7.3 Slots

- ❖ **Time Complexity**: $O(1)$ (constant operations to generate three random numbers and compare them)
- ❖ **Space Complexity**: $O(1)$ (no additional memory usage except local variables)

7.4 Coin Flip

- ❖ **Time Complexity**: $O(1)$ (constant operations to generate three random numbers and compare them)
- ❖ **Space Complexity**: $O(1)$ (no additional memory usage except local variables)
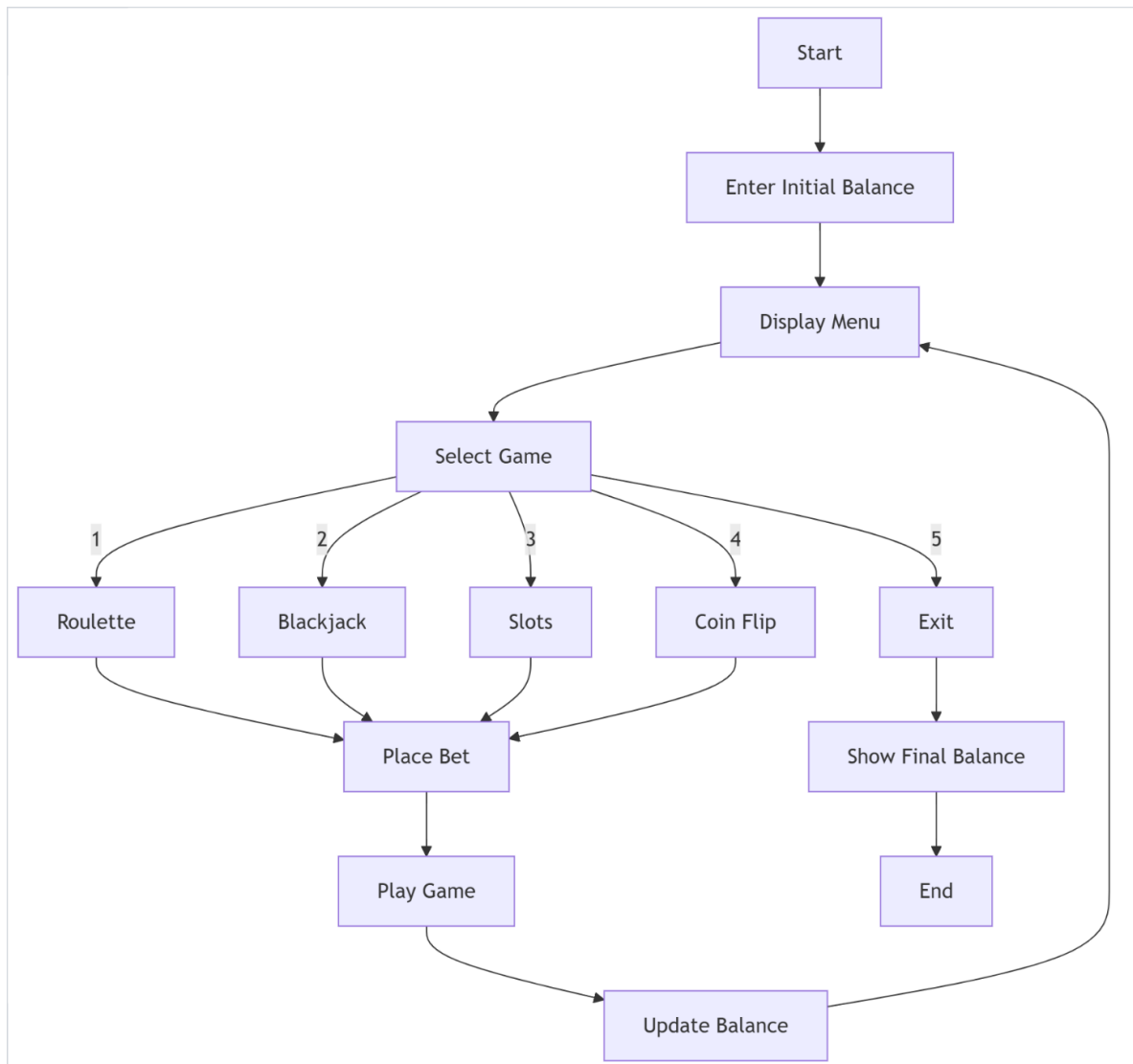
8. Workflow

Figure 9: Workflow of the C Program

A simplified flowchart of the C program that focuses on the core program flow instead of expanding it and making it more complex by including the working of each function. This makes it easier to understand the basic structure of the Interactive Casino. It includes program initialization, game selection menu, common betting process, game execution, balance update and return to menu or exit the program.

9. Glossary

Balance: The virtual currency amount available to the player for betting across different games.

Bet: The amount of virtual currency wagered by a player in any given game.

Blackjack: A card game where players aim to achieve a higher card value than the dealer without exceeding 21.

Coin Flip: A simple betting game based on guessing whether a virtual coin will land on heads or tails.

Function: A reusable block of code that performs a specific task in the program.

Jackpot: The highest possible payout in the Slots game, achieved by matching all three slot numbers.

Pointer: A variable that stores the memory address of another variable, used for direct memory manipulation.

rand(): A C library function that generates pseudorandom numbers used for game outcomes.

Roulette: A game where players bet on a number between 0 and 18, with payouts based on correct guesses.

Slots: A game simulating a slot machine with three reels, offering various payout combinations.

srand(): A C library function used to seed the random number generator.

time(): A function that returns the current system time, used to seed the random number generator

10. Conclusion

The Interactive Casino project successfully demonstrates the practical application of fundamental C programming concepts in creating an engaging and functional gaming simulation [2]. Through the implementation of four distinct casino games - Roulette, Blackjack, Slots, and Coin Flip - this project showcases how basic programming elements can be combined to create an interactive entertainment system that effectively implements random number generation using rand() and srand(time(NULL)) to ensure unpredictable and fair game outcomes, while maintaining robust error handling and input validation mechanisms for program stability. The efficient memory management through pointers for balance updates and modular code structure with separate functions for each game enhances maintainability, while the implementation of specific game mechanics - from Roulette's high-risk, high-reward system with a 5.26% win probability to Blackjack's player versus dealer comparison logic, Slots' multiple winning conditions, and Coin Flip's straightforward 50-50 chance gameplay demonstrates the versatility of programming concepts in creating diverse gaming experiences. The project serves as a practical example of how programming concepts can be applied to create real-world applications, demonstrating the importance of structured programming approaches, random number generation techniques, user input handling, dynamic memory management, and modular code organization. This casino simulator not only provides an entertaining gaming experience but also stands as a testament to how fundamental programming principles can be effectively utilized to create functional and engaging applications that combine learning with entertainment, illustrating the practical applications of coding skills in the gaming industry.

11. References

1. Kernighan, B.W. and Ritchie, D.M., 2002. The C programming language.

2. Deitel, P., & Deitel, H. (2020). C How to Program (8th ed.). Pearson Education.

3. Reese, R.M., 2013. Understanding and using C pointers: Core techniques for memory management. " O'Reilly Media, Inc.".

4. Gustedt, J., 2019. Modern C. Manning Publications.

5. IEEE. (2018). IEEE Standard for Information Technology--Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7.

6. Harbison, S. P., & Steele, G. L. (2002). C: A Reference Manual (5th ed.). Prentice Hall.

7. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.

8. Jain University. (2024). Programming in C Course Materials (24BCA1C04). Department of Computer Science and Information Technology.