Snakes and Ladders Using Weighted Graphs

Samar Ahmed 900222721

Youssef Aboushady 900223372

Tony Gerges 900222981

Department of Computer Science and Engineering, The American University in Cairo

CSCE 2211-01: Applied Data Structures

Dr. Amr Goneid

May 16, 2024

Abstract

Throughout history, humans have used games to combat boredom and keep themselves physically and intellectually stimulated. As technology improved, so did games. In addition to their increased complexity, games are constantly engineered to contain the perfect balance of uncertainty and player control. Randomness, or the unpredictability of the outcome of a game, is a key component of its entertainment value. This is illustrated in the classic board game "Snakes and Ladders." This report documents an endeavor to implement Snakes and Ladders digitally using the C++ programming language. The game's fundamental rules are explained, as well as the methodology chosen to translate it into a computer program. C++ contains a variety of data structures that can be tailored to different purposes; these are discussed, along with the relevant algorithms required to implement the game. Moreover, the development is explained in full detail. The process consisted of numerous phases, from planning and brainstorming to debugging and validating the output of the code. Results from the testing phase are included to give readers a good idea of the game's logic, as well as what to expect when playing the game. The constituent files of the program are appended to the report, and the code is deciphered thoroughly within the report itself. A complete analysis of the program is provided, justifying how and why the game works the way it does. Finally, the report reflects on the successes and failures of the program, mentioning its potential applications in the context of today's society.

Keywords: Snakes and Ladders, video games, C++ programming, game development, data structures

Outline

Snakes and Ladders Using Weighted Graphs

Introduction

Human beings have many essential needs, such as hunger, thirst, sleep, and hygiene. These requirements must be satisfied for humans to function optimally and live fulfilling lives (Schuppert, 2013). Among these basic needs is recreation; if it is not met, an individual may struggle to work and contribute to society.

Consequently, humans have constantly sought ways to "have fun." Several methods have been developed to achieve this, with one of the most popular being games. From a young age, people are exposed to games as a way to pass the time, stay entertained, and sometimes even to learn new concepts.

Early evidence of games dates back thousands of years. Simple games with minimal requirements, like "Tic Tac Toe" or "Rock Paper Scissors," were created to be playable anytime and anywhere (Monnens, 2013). These games have stood the test of time and are still played today. However, with the development of technology, humans have discovered novel methods of stimulation. Increasingly complex games like chess and checkers were developed to utilize more brainpower and involve more senses (Nicholson, 2013). The more complex the game, the more outcomes there are possible. This adds an element of randomness, making the game unpredictable and hence more exciting. By adding a specific set of rules and combining them with unpredictable outcomes, people were able to create dozens of unique games. These games can be organized into categories. One category of particular interest is board games.

Simply put, a board game is a game that is played in a small, confined space. The entire game takes place within this field, making it simple, self-contained, and easier to follow along (Noda et al., 2019). Similar to most games, board games utilize randomness as a way to keep players engaged. There are numerous ways to incorporate randomness, but this is usually done through the rolling of dice. The most common form is the six-sided die; it contains the first six positive integers and is used to generate a "random" value outside of the player's control.

Although randomness is essential in game design, there needs to be a balance between luck and skill. In other words, winning the game must depend on two things: random values generated in the player's favor, and the technical ability of the player to follow the game's rules (Mindell, 2018). One famous exception to this rule is "Snakes and Ladders," a classic board game that has been played for centuries. In the age of digitization, traditional board games are being replaced by more advanced video games. Many of these games are completely unique, as they are developed using tools that were not available in the past (Arsenault, 2009). On the other hand,

developers have chosen to revive classic board games, keeping them relevant by implementing them using this new technology.

For our project, we decided to recreate the Snakes and Ladders board game digitally using our knowledge of programming techniques, algorithms, and data structures. Specifically, we applied what we learned in our undergraduate studies to implement the game using the C++ programming language. This has been done before, but our task was different because we had to tailor the program to our own knowledge. Some external research was done, but the game logic and visual representation were planned, developed, and tested entirely by us.

Problem Definition

The premise of Snakes and Ladders is simple. There is a board containing positions from one to 100, and the objective is to start at the beginning and reach the end first. At least two players are required, and each player must reach 100 before the other player(s) (Ibam et al., 2018). The only way for a player to traverse the board is by rolling a six-sided die; the result of this roll determines how many places the player is allowed to advance.

However, it is not that simple. Numerous obstacles and shortcuts are placed strategically throughout the board to add a layer of suspense to the game. “Snakes” force the player to return to an earlier value, while “ladders” allow the player to skip a certain number of places to fast-track their way to 100. This feature is where the name of the game is derived, and it is what makes the game special (Ibam et al., 2018). Moreover, snakes and ladders are not of equivalent length. For example, one snake may push a player back only four spots, while another may force the player to go 51 places back, undoing roughly half of their progress toward the finish line.

These basic rules are what constitute the core functionality of the game, but players may add certain features of their own to make the game more interesting. For example, some people prefer to do an initial roll to determine the order in which the participants get to move (Reid & Van Niekerk, 2014). Consider a game with three players. The first player (player 1) rolls a two, the second (player 2) rolls a three, and the third (player 3) rolls a six. In this case, the order in which they would roll the die throughout the game would be player 3, followed by player 2, and then finally player 1.

Another twist that could be incorporated into Snakes and Ladders is moving players backward if they exceed the score limit of 100. For example, if a player is at 97, then they need to roll a value less than or equal to three to advance. If they roll exactly three, then they win the game. If they roll less than three (one or two), then they advance. But if they roll a four, five, or six, then they

must return that many places backward (Reid & Van Niekerk, 2014). This process is repeated until every player reaches the end, or in some iterations of the game, until a single player reaches 100.

For our purposes, we have decided to implement the most basic version of the game. Players roll in the order they are registered in the program, and they are not penalized for exceeding 100 points. However, in order for a player to win, they need to stop at exactly 100. For instance, if a player is at 99, then they will not be allowed to advance unless they roll a one.

When it comes to implementing Snakes and Ladders as a video game, the programmer is faced with a handful of difficulties. First, they need to think about the way they are going to represent the game board itself. This can be done using a variety of data structures, such as a vector, linked list, graph, or even a simple array. Additionally, the developer needs to consider a way to make the program detect when a player stops at the foot of a ladder or the head of a snake. There is also the issue of borderline cases; if a player approaches the end of the board and rolls a value greater than what they need to reach 100, this needs to be handled appropriately by the program. Finally, after putting the game together, the programmer must find a way to display the output of the program visually. This step is essential; without it, the game lacks the feeling of a simulation that keeps players engaged.

Methodology

While implementing Snakes and Ladders using C++, we had to consider two factors: the limitations of the language, and our own knowledge of its capabilities. We began by brainstorming the different variables and functions that would be required for the core functionality of the game. Then, as we developed the game, we added more features, increasing the complexity and versatility of the program. To represent the board, players, snakes, and ladders, we used a variety of simple and complex data structures; these will be explained in further detail later in the report.

To display the game visually, we had to manipulate several components of the console on which the program output was displayed. A typical output consists of alphanumeric text and symbols, displayed in a plain white font on a black background. This did not meet our needs, as a real snakes and ladders board is vibrant and colorful. Instead, we gave the user the option of choosing two different colors for the board to be displayed in. We also allowed customization of the colors in which players' names are displayed to differentiate them from the rest of the board.

The development process followed a simple timeline. After planning our game, we were ready to write the code. When developing a program, it is prudent to test each function as soon as it is created to ensure that it works as desired (Boehm, 1983). We made sure to do this to avoid difficulties when debugging at the end. After developing and testing each function of our program individually, we assembled the different components to create an integrated whole. The specifics and results of this process are detailed in the coming sections.

Specification of Algorithms to be Used

In this program, we used the weighted graph data structure to implement the snakes and ladders of the game. The program also utilizes an array of structures (struct), the implementation of which will be further explained later in this report. The overall program is composed of three header files.

The first header file is Graph.h, which contains four functions: addEdge(), getDest(), isLadder(), and isSnake(). The second file, AnimAndColour.h, has 3 different functions: changeColour(), clearScreen(), showLoadingScreen(), and showBlinkingLights(). Finally, the PlayerAndBoard.h file contains the struct “player” and the class “PlayerAndBoard.” This class has five functions that implement the game: die_roll(), create_players(), display_players(), move_player(), and display_board().

Data Specifications

This program was designed to be user-interactive. Similar to a “press play” button, the user will first type if they would like to play or not. If the answer is “yes,” they will then be prompted to enter the number of players, as well as the color of each respective player. This is needed to differentiate the players from each other, as in the real board game. Finally, the user will select the colors they want the checkered grid to be displayed in from a list of colors. At this point, the program will display the colored grid with the players, and the game will commence. Players will take turns, in the order they were registered, to press the “Enter” key to simulate rolling the die. With each turn, the game will display the value of the roll, the new score of the player, and the current state of the board.

Experimental Results

First, the loading screen will appear, where these strings will be displayed consecutively. Meanwhile, music will play in the background; it will continue to do so for the entire duration of the game.

Image 1: Loading screen

Once the screen is cleared, the following message will be displayed:

Image 2: Welcome message

Then, the user will be asked if they want to play:

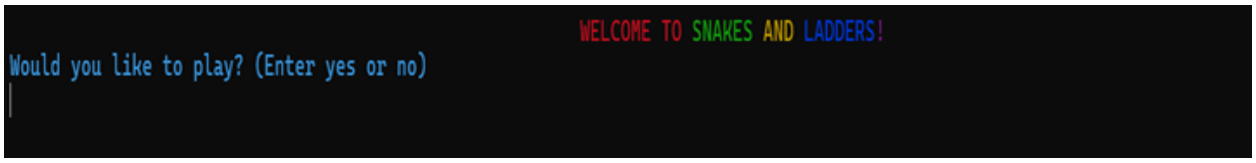

Image 3: Prompt to play

At this point, there are two possible cases. The first case is simple; if the user says “no,” then the program will reply “Hope you can play next time!” and then terminate.

Image 4: Declining to play

On the other hand, if the player answers “yes,” they will then be asked for the number, names, and colors of the players, as well as the colors of the board. This is displayed below:

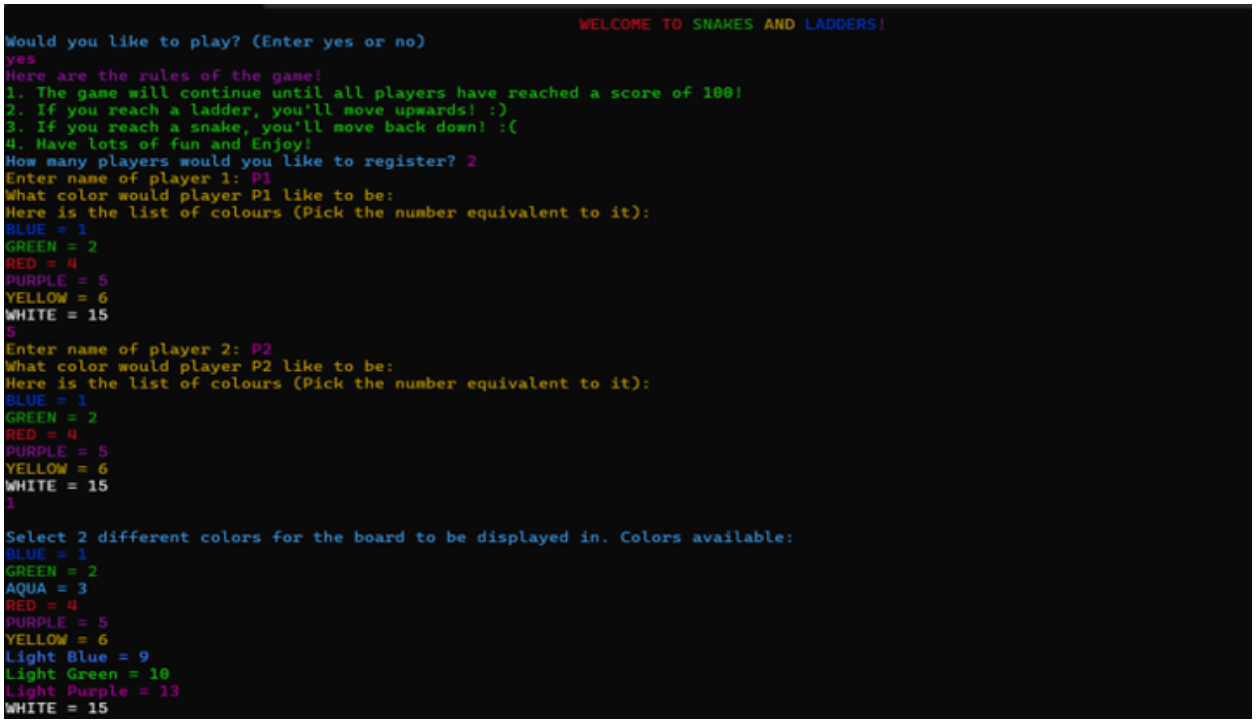

Image 5: Customization options

Once the visual aspects have been customized, the rules of the game will be explained to indicate how the program will function.

To demonstrate the output, we will simulate a typical game between two players. The first player, P1, will choose purple, and the second player, P2, will choose blue. The colors of the board are chosen to be red and green. The snakes, ladders, and initial game board are shown in the following image.

Image 6: Ladder and snake symbols, first turn

For the first turn, the die will be rolled automatically, and the first player will move accordingly. The updated scores will be displayed on the right-hand side of the screen. From this point onward, players will press the “Enter” key to roll the die.

After their first roll, P2 can now be seen on the board:

Image 7: Early game scenario

A few turns later, this is the state of the board:

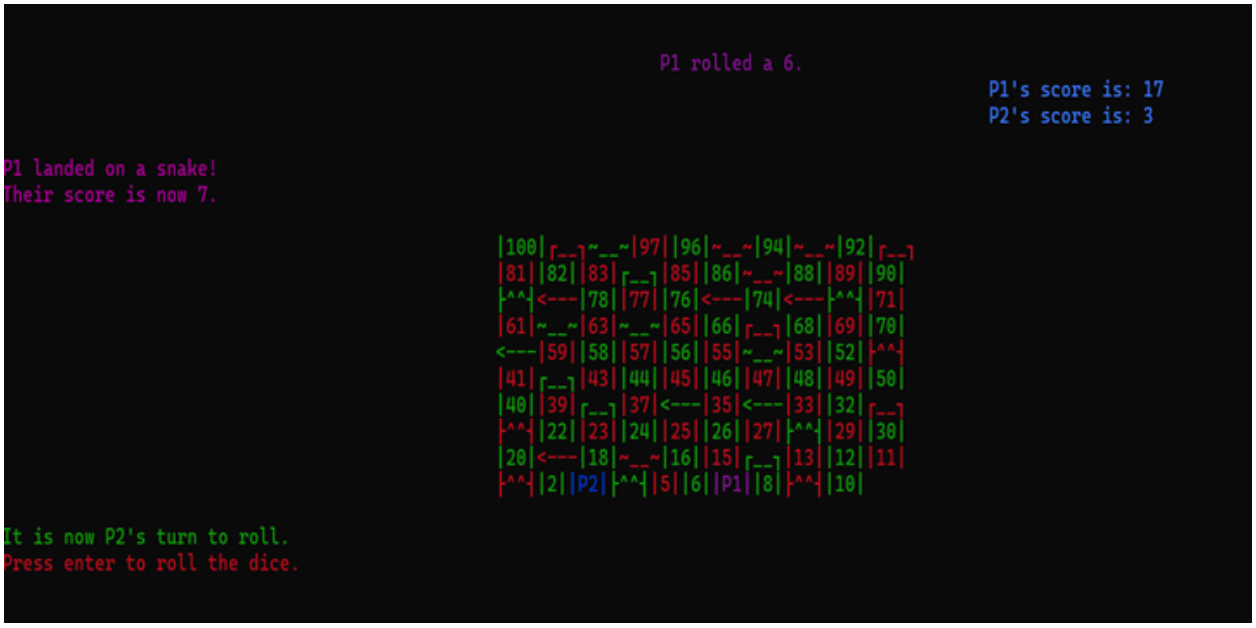

Image 8: Example of a snake

P1 was at a score of 11, but after rolling a six and moving to 17, they reached the head of a snake. This returned them to seven, where the tail of a snake was previously displayed.

Here is an example of a player reaching a ladder:

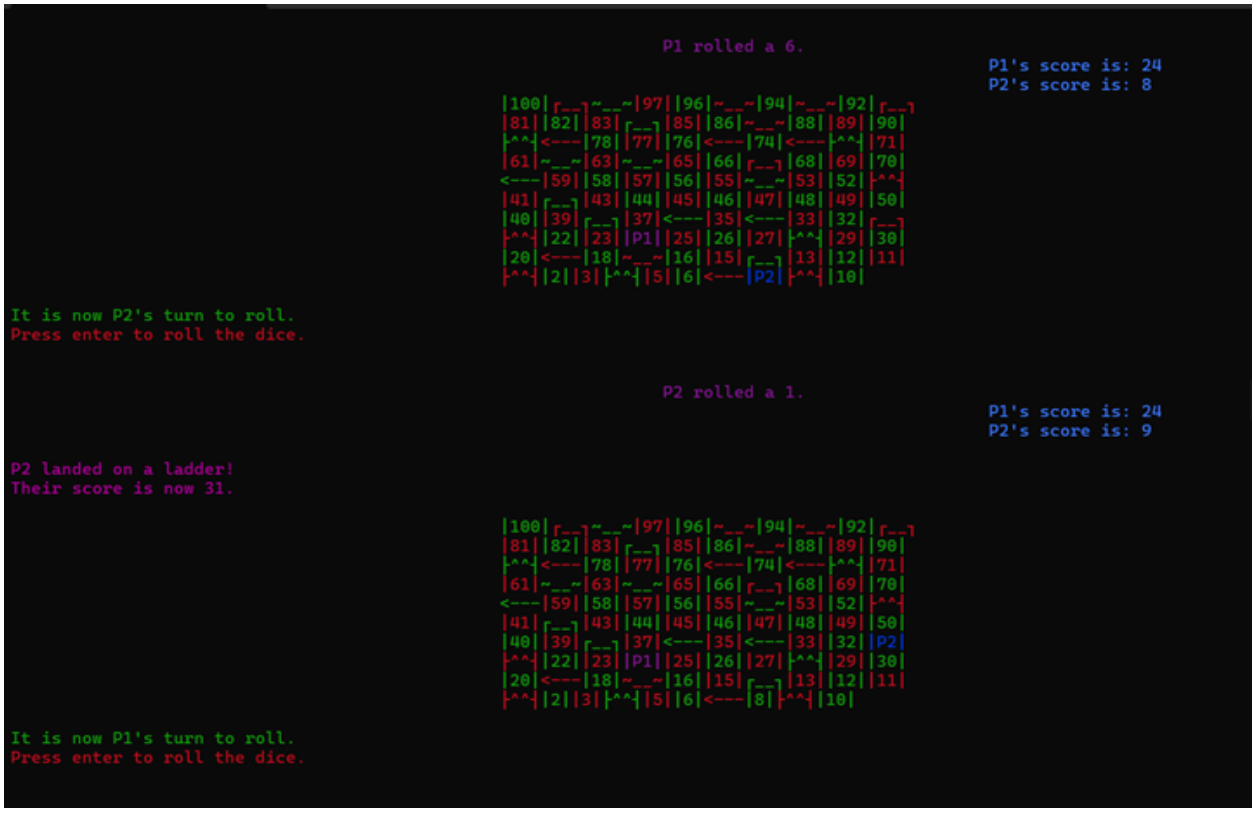

Image 9: Example of a ladder

P2 was initially at position eight. After rolling and moving one place forward, they stopped at the bottom of a ladder, allowing them to skip to position 31.

The players continue until one of them approaches the end, as shown in the following image.

Image 10: End game scenario

For any position greater than 94, players need to roll a value less than or equal to what is required for them to win. In other words, for them to move, they need to be able to stop at or before 100. In the previous image, P2 is at position 99, so they must roll a one to proceed and win the game. A winning scenario is shown below:

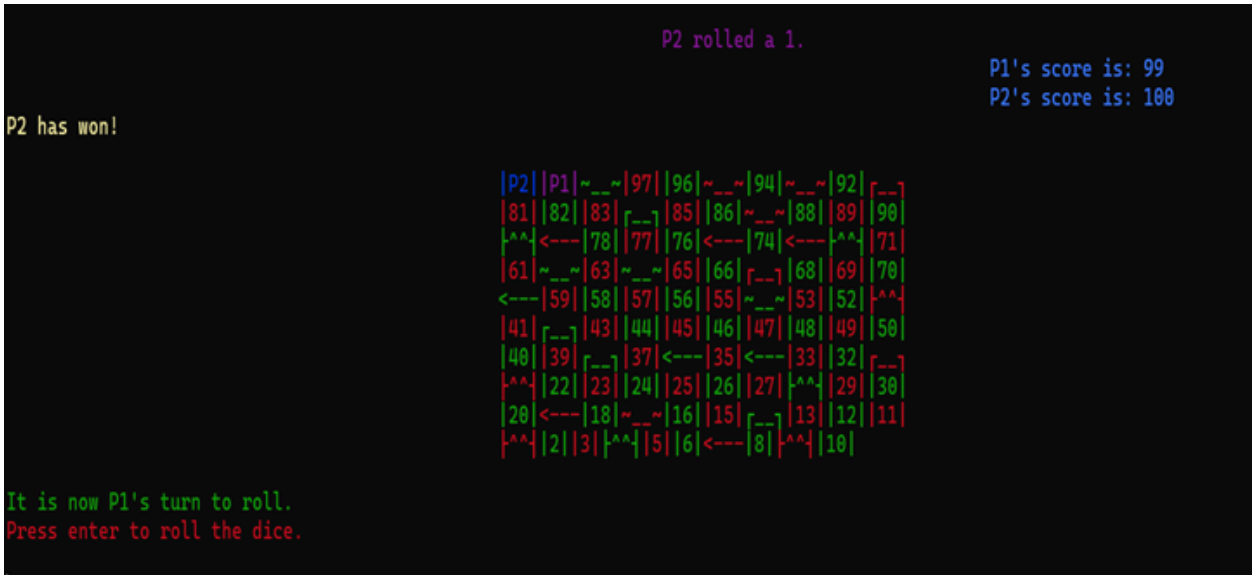

Image 11: Winning scenario

P2 has won, but the game is not over yet; P1 has to reach the end too. After P1 reaches 100 (and hence, all players have won), the board is displayed one last time.

Image 12: Final scoreboard

The final rankings are shown, and the user is then asked if they would like to play again. If they agree, the game will reset to the beginning, prompting the user to register the players. Otherwise, the program will terminate normally.

Algorithm Analysis

Graph.h and Graph.cpp:

The header file contains the "Graph" class, which consists of:

1. Vector<pair<int,int>> adj[101]: "Adj" is an array of 101 elements, with each element being a vector that holds a pair of integers. The first integer represents the adjacent node, which shares an edge with another node. The second integer is the weight of the edge connecting the node to the adjacent node. As indicated by the size of the array, up to 101 nodes in the graph can be created; these are used for the positions of the board.

2. Graph(): The constructor that contains the list of snakes and ladders. This is created using the addEdge() function.

3. addEdge(int u, int v, int wt): This function forms a graph by accepting two nodes, u and v, as parameters and connecting them with an edge of weight "wt" (the third parameter) (Geeks for Geeks, 2017). The purpose of this graph is to store the values associated with the snakes and ladders.

4. getDest(int score): This function receives the current score of a player. It then checks if this score is equivalent to the first node of any of the pairs in the graph. If it is, then it will return the second element, which is the new score of the player. This is what allows players to "skip" from one position to another when they stop at a snake or ladder. To execute this, the function must call two more functions: isLadder() and isSnake(). If the player's current score is not found in the graph, then the function will simply return the same score it took as a parameter.

5. isLadder(int node): This Boolean function checks if a player's score is in the list of ladders, i.e. whether they have reached the bottom of a ladder.

6. isSnake(int node): This Boolean function checks if a player's score is in the list of snakes, i.e. whether they have reached the head of a snake.

AnimAndColour.h and AnimAndColour.cpp

The colors used for the players and board are all declared in the class "AnimAndColour" at the beginning of the file (Geeks for Geeks, 2020).

1. changeColour(int colour): This function uses the Windows console function SetConsoleTextAttribute() to change the text color. It takes as a parameter the value of the color the text is to be displayed in (Ehiorobo, 2019).

2. clearScreen(int characterLength): This function clears a certain number of characters from the console. It outputs backspace characters (\b) a specific number of times determined by its parameter (Ehiorobo, 2019).

3. showLoadingScreen(): This function displays the loading screen shown at the start of the game. It outputs a string of underscores ("_ _ _"), before entering a loop where it clears the screen, changes the text color, and then outputs a string of asterisks ("* * *") (Ehiorobo, 2019).

4. showBlinkingLights(): This function calls showLoadingScreen(), then outputs the message "WELCOME TO SNAKES AND LADDERS!" Each word is displayed in a different color after a short interval of time.

PlayerAndBoard.h and PlayerAndBoard.cpp

The header file contains the struct "Player," which consists of the player's name, score, win state (whether or not the player has won), and color. In the class "PlayerAndBoard," there are five functions:

1. die_roll(): This is used to simulate the roll of a six-sided die. It returns an integer value between one and six, inclusive.

2. create_player(Player *players, int num, AnimAndColour effects): This function takes as parameters an array of players, the number of players, and the player color. It then asks the user for the name of the player; this is inserted into the "name" field in the "Player" struct. It also asks for the color the user wants; this is inserted into the "color" field. The function is called for each player registered in the game.

3. display_players(Player *players, int num): This function is a simple loop that displays each player's array index (the order they were registered in), name, and score.

4. move_player(Player *players, int turn, int roll, int &wins, Graph &g, int num, vector <player&winners, AnimAndColour effects): This function implements the game logic. The player's score is updated based on the result of the die roll. It first checks if the player's new score (current score + roll) exceeds 100; if it does, then the player's turn is skipped. This means that the player must try again on every turn until they reach exactly 100. However, if the new score is less than or equal to 100, then the player's score will be updated, and the scores of all players in the game will be displayed. The function will then check if the player's score has changed as a result of landing on a snake or ladder. If it has, then it will state that the player has landed on a snake or ladder, indicating their new score. Finally, if the player's updated score is exactly 100, then they are added to the list of winners, the number of total winners is incremented, and they are declared a winner on the screen.

5. display_board(Player *players, int num, int c1, int c2, AnimAndColour& effects): This function displays the game board. It begins by creating a 10x10 board and filling in the numbers from 1 to 100. At specific indices, the symbols for the snakes and ladders are added to the board. Then, it goes through each player and checks if their score is greater than 0. If it is, then the position of the player will display their name instead of a snake, ladder, or regular number. The position is calculated by finding the row and column.

   The row is calculated by the equation: row[i] = 9 - (score - 1)/10. The score is first decremented to adjust the row from 0 to 9, then it is divided by 10 to get the specific index required. The result is subtracted from 9 to flip the board such that the grid goes from 1 to 100, not 100 to 1. This is then added to the row array.

   The column is calculated as: col[i] = (score - 1)%10. The score is decremented to adjust it from 0-9, and then modulo-10 is taken to find the specific column the player should be at. If the row is even, then the column will be reversed because the game board goes from left to right for even-numbered rows and right to left for odd-numbered rows. As with the row, the column will be stored in an array.

   Finally, the function will print the board. For each square, the variables "isPlayerPosition" and "playerColor" are initialized. The boolean variable is used to check if the square is a player's position and store the player's color if it is at that position. A for loop will iterate over the player's position in the arrays "row" and "col." If the position on the grid matches the position of a player, then isPlayerPosition will be set to true, and playerColor will be set to the color of the player with whom a match was found. If a player's position is not found, then isPlayerPosition will remain false, and the

color of the grid will be set to either c1 or c2, depending on whether the sum of the row and column indices is odd or even. This produces the desired checkerboard pattern.

Any given square on the board can display one of three possible things. Precedence is given first to player names, then to a snake or ladder, and finally to the number of the position. In other words, the square a player stops on must display that player's name. If a square contains a ladder top/bottom or a snake head/tail, then it will be shown. Squares only display their position on the board if they have neither a snake/ladder nor a player standing on them.

SnakeAndLadder.cpp

This implementation file contains the main function of the program. Its contents can be summarized as follows:

1. srand(time(0) is used to ensure that every die roll is random and independent from the previous roll.

2. Objects are created to instantiate every class necessary.

3. A while loop is used to repeat the game song five times (O'Didily, 2023).

4. showBlinkingLights() is called to output the loading screen and welcome display.

5. After the user agrees to play, the rules of the game are displayed. The player then enters the number of players; this is the size of the array of type "Player." create_players() is called to initialize each player, and the colors of the grid are selected.

6. "l_top" and "l_bottom" are unique character arrays. They contain special characters from the ASCII (American Standard Code for Information Interchange) table and are used to visually represent a ladder top and bottom, respectively. Similarly, "s_head" and "s_tail" are strings used to represent the head and tail of a snake. These four symbols are labeled at the start of the game to avoid any confusion for the user.

7. The game takes place in a while loop that ends only after all the players have won. Before a player rolls, the loop checks to make sure that they have not won yet. If they have already won, their turn will be skipped. Otherwise, they will roll the die and move accordingly, and the result of this turn will be displayed on the screen.

8. Once all the players have won, the scoreboard is displayed to show their rank and name. Finally, the user is asked if they would like to play again. If they agree, the whole process will be repeated. Otherwise, the program will terminate successfully.

Critique

Our main goal for this game was to implement Snakes and Ladders in a way that is both user-friendly and visually appealing. We aimed to achieve this using the graph data structure and multiple features to facilitate user interaction. The game was designed to be as simple and clear as possible by explaining the rules, representing the snakes and ladders, labeling the board, coloring the grid and players, and displaying turn results regularly. After analyzing the output of our work, we were satisfied with the results. Nevertheless, we realize that there is plenty of room for improvement.

We wanted to further enhance our program such that all turns are displayed on one single board, as opposed to displaying a new grid with every roll. This could have been accomplished through the clearScreen() function. It could be argued that our version is advantageous, as it shows the full history of the game. However, a single grid would expand the capabilities of the program. For example, it could show the players move step-by-step, or simulate the entire game automatically. When it comes to video game development, the sky is the limit; a game's aspects can always be further refined.

Conclusion

Through the development of Snakes and Ladders as a video game, we gained invaluable experience. We were able to combine our individual strengths and collaborate as a team to achieve our main objective. The output was precise, satisfying all the requirements we had initially aimed for. Moreover, the final product was of high quality, meeting all criteria and fulfilling consumer expectations.

That is not to say, however, that the project did not have its drawbacks. We were faced with a number of limitations, ranging from the capabilities of C++ to our own programming experience. For example, the code could have been more efficient. From a space complexity perspective, the components of the game could have consumed less computer memory. From a time complexity perspective, the program could have been faster to run, requiring fewer operations for the processor to execute.

In addition, there are more features to be added. For example, the game could include the option to customize the background music; there could be a list of tracks for the player to choose from. Furthermore, there could be multiple modes for players to select from, each with its own set of rules. These modifications would make the program more adaptable, catering to a wider range of customers. Overall, there were many lessons for us to learn, and this project will undoubtedly benefit us in the future.

Snakes and Ladders is an all-time favorite; it brings individuals together, regardless of their cultural background. By implementing it using our knowledge of C++, we were able to create something that people can benefit from on a global scale. The game has stood the test of time, lasting centuries despite its incredible simplicity. Even with the advancement of technology and the evolution of video games, Snakes and Ladders is sure to remain a staple for generations to come.

Acknowledgments

The program and report were both produced by the same team of three members: Samar Ahmed, Tony Gerges, and Youssef Aboushady. We were inspired by external sources, but our work is entirely original.

First, we would like to thank our course instructor, Dr. Amr Goneid, for this opportunity. His lectures and teaching were necessary for us to acquire the knowledge needed to develop the program. Credit is also due to our teaching assistant, Eng. Mohamed Ibrahim; it was through his support, guidance, and feedback that we were able to carry this project out from start to finish. Finally, we extend our gratitude to the people responsible for the sources used to develop our program and write this report. These sources are cited fully in the references section.

References

Arsenault, D. (2009). Video game genre, evolution and innovation. Eludamos: Journal for Computer Game Culture, 3(2), 149-176. https://doi.org/10.7557/23.6003

Boehm, B. W. (1983). Seven basic principles of software engineering. Journal of Systems and Software, 3(1), 3-24. https://doi.org/10.1016/0164-1212(83)90003-1

Ehiorobo, E. (2019, October 9). *Creating a console animation with C++*. How-Tos. https://medium.com/building-a-simple-text-correction-tool/creating-a-console-animation-with-c-18bf9e8ca582

Geeks for Geeks. (2017, January 20). *Graph implementation using STL for competitive programming | Set 2 (Weighted graph)*. GeeksforGeeks. https://www.geeksforgeeks.org/graph-implementation-using-stl-for-competitive-programming-set-2-weighted-graph/?ref=lbp

Geeks for Geeks. (2020, May 29). *How to print Colored text in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/how-to-print-colored-text-in-c/

Ibam, E. O., Adekunle, T. A., & Agbonifo, O. C. (2018). A moral education learning system based on the snakes and ladders game. EAI Endorsed Transactions on e-Learning, 5(17). https://doi.org/10.4108/eai.25-9-2018.155641

Mindell, R. (2018). Context matters: Luck and the paradox of skill. Gaming Law Review, 22(5), 270-272. https://doi.org/10.1089/glr2.2018.2253

Monnens, D. (2013). Charles Babbage and the "First" Computer Game. Digital Games Research Association. efaidnbmnnnibpcajpcglclefindmkaj/http://digra.org/wp-content/uploads/digital-library/paper_436.pdf

Nicholson, S. (2013). Playing in the past: A history of games, toys, and puzzles in North American libraries. The Library Quarterly, 83(4), 341-361. https://doi.org/10.1086/671913

Noda, S., Shirotsuki, K., & Nakao, M. (2019). The effectiveness of intervention with board games: A systematic review. BioPsychoSocial Medicine, 13(22). https://doi.org/10.1186/s13030-019-0164-1

O'Didily, M. (2023). *How to Stop and Play Music Using C++ (Simple) (PlaySound)*. Www.youtube.com. https://www.youtube.com/watch?v=zOljIjBJvRI&ab_channel=MaxO%27Didily

Reid, R., & Van Niekerk, J. (2014). Snakes and ladders for digital natives: Information security education for the youth. Information Management & Computer Security, 22(2), 179-190. https://doi.org/10.1108/imcs-09-2013-0063

Schuppert, F. (2013). Distinguishing basic needs and fundamental interests. Critical Review of International Social and Political Philosophy, 16(1), 24-44. https://doi.org/10.1080/13698230.2011.583532

SNAKES AND LADDERS 1-100. (2023). Thermmark. https://www.thermmark.co.uk/product/snakes-and-ladders-1-100/

Appendix

Graph.h

```
#pragma once
#include <cstdlib>
#include <vector>
#include <iterator>
#include <iostream>

using namespace std;

class Graph {
     vector<pair<int, int> > adj[101];
public:

     Graph();

     void addEdge(int u, int v, int wt);
     int getDest(int score);

     bool isLadder(int node);
     bool isSnake(int node);

private:
};
```

Graph.cpp

```
#include "Graph.h"

Graph::Graph() {
     // Ladders
     addEdge(1, 38, 1);
     addEdge(4, 14, 1);
     addEdge(9, 31, 1);
     addEdge(21, 42, 1);
     addEdge(28, 84, 1);
     addEdge(51, 67, 1);
     addEdge(72, 91, 1);
     addEdge(80, 99, 1);

     // Snakes
     addEdge(17, 7, 1);
     addEdge(54, 34, 1);
     addEdge(62, 19, 1);
     addEdge(64, 60, 1);
     addEdge(87, 36, 1);
     addEdge(93, 73, 1);
     addEdge(95, 75, 1);
     addEdge(98, 79, 1);
}

void Graph :: addEdge(int u, int v, int wt) {
     adj[u].push_back(make_pair(v, wt));
     adj[v].push_back(make_pair(u,        wt));        //reference
GeeksforGeeks
}
int Graph:: getDest(int score) {
     int v, w;
     for     (vector<pair<int,     int>     >::iterator     it     =
adj[score].begin(); it != adj[score].end(); it++) {
          v = it->first;
          w = it->second;
          if (isLadder(score) || isSnake(score)) {
               return v; // Return the destination of the second
edge
          }
     }
     return score;
}

bool Graph:: isLadder(int node) {
     // Check if the node is the starting node of a ladder
```

```
     return (node == 1 || node == 4 || node == 9 || node == 21
|| node == 28 || node == 51 || node == 72 || node == 80);
}
bool Graph:: isSnake(int node) {
     return (node == 17 || node == 54 || node == 62 || node ==
64 || node == 87 || node == 93 || node == 95 || node == 98);
}
```

AnimAndColour.h

```
#pragma once

#include <iostream>
#include <string>
#include <Windows.h>
#include <MMSystem.h>

using namespace std;

class AnimAndColour {
public:
	const int BLUE = 1;
	const int GREEN = 2;
	const int AQUA = 3;
	const int RED = 4;
	const int PURPLE = 5;
	const int YELLOW = 6;
	const int LightBlue = 9;
	const int LightGreen = 10;
	const int LightPurple = 13;
	const int LightYellow = 14;
	const int WHITE = 15;

	void changeColour(int colour);

	void clearScreen(int characterLength);

	void showLoadingScreen();

	void showBlinkingLights();

private:

};
```

AnimAndColour.cpp

```
#include "AnimAndColour.h"

void AnimAndColour::changeColour(int colour) {
	HANDLE hConsole;
	hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
	SetConsoleTextAttribute(hConsole, colour);
}
void AnimAndColour::clearScreen(int characterLength) {
	for (int i = 0; i < characterLength; i++) {
		cout << "\b";
	}
}


void AnimAndColour::showLoadingScreen() {
	cout << "\t\t\t\t\t\t\t";
	int i;
	string closed = "_ _ _", open = "* * *";
	int colour[] = { RED, GREEN, BLUE };
	cout << closed;
	for (i = 0; i < 3; i++) {
		Sleep(500);
		clearScreen(5);
		changeColour(colour[i]);
		cout << open;
		Sleep(500);
		clearScreen(5);
		changeColour(WHITE);
		cout << closed;
	}
	clearScreen(5);
	changeColour(WHITE);
}
void AnimAndColour::showBlinkingLights() {
	showLoadingScreen();
	changeColour(RED);
	cout << "WELCOME TO";
	Sleep(500);

	changeColour(GREEN);
	cout << " SNAKES";
	Sleep(500);
	changeColour(YELLOW);
	cout << " AND";
	Sleep(500);
```

```
        changeColour(BLUE);
        cout << " LADDERS";
        Sleep(500);

        changeColour(PURPLE);
        cout << "!\t\t\t\t\t" << endl;
        Sleep(500);
        changeColour(WHITE);
}
```

PlayerAndBoard.h

```
#pragma once
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
#include "Graph.cpp"
#include "AnimAndColour.cpp"

using namespace std;

const int WHITE = 15;

struct player {
     string name = "";
     int score = 0;
     bool win = 0;
     int color = WHITE;
};

class PlayerAndBoard {
public:

     int die_roll();

     void create_players(player* players, int num, AnimAndColour
effects);

     void display_players(player* players, int num);

     void move_player(player* players, int turn, int roll, int&
wins, Graph& g, int num, vector<player>& winners, AnimAndColour
effects);

     void display_board(player* players, int num, int c1, int
c2, AnimAndColour& effects);

private:
};
```

PlayerAndBoard.cpp

```
#include "PlayerAndBoard.h"

int PlayerAndBoard::die_roll() {
     return 1 + (rand() % 6);
}


void  PlayerAndBoard::create_players(player*  players,  int  num,
AnimAndColour effects) {
     for (int i = 0; i < num; i++) {
          effects.changeColour(effects.YELLOW);
          cout << "Enter name of player " << i + 1 << ": ";
          effects.changeColour(effects.LightPurple);
          cin >> players[i].name;
          effects.changeColour(effects.YELLOW);
          cout << "What color would player " << players[i].name
<< " like to be:" << endl;
          cout << "Here is the list of colours (Pick the number
equivalent to it): " << endl;
          effects.changeColour(effects.BLUE);
          cout << "BLUE = 1" << endl;
          effects.changeColour(effects.GREEN);
          cout << "GREEN = 2" << endl;
          effects.changeColour(effects.RED);
          cout << "RED = 4" << endl;
          effects.changeColour(effects.PURPLE);
          cout << "PURPLE = 5" << endl;
          effects.changeColour(effects.YELLOW);
          cout << "YELLOW = 6" << endl;
          effects.changeColour(effects.WHITE);
          cout << "WHITE = 15" << endl;
          effects.changeColour(effects.LightPurple);
          cin >> players[i].color;

     }
     effects.changeColour(effects.WHITE);
}

void PlayerAndBoard::display_players(player* players, int num) {
     for (int i = 0; i < num; i++) {
          cout << endl;
          cout << "PLAYER " << i << endl;
          cout << "Name: " << players[i].name << endl;
          cout << "Score: " << players[i].score << endl << endl;
     }
```

```
}

void PlayerAndBoard::move_player(player* players, int turn, int
roll, int& wins, Graph& g, int num, vector<player>& winners,
AnimAndColour effects) {
     effects.changeColour(effects.WHITE);
     if (players[turn].score + roll > 100) {
          effects.changeColour(effects.RED);
          cout << players[turn].name << "'s score has exceeded
100." << endl << "Their turn will be skipped." << endl;
     }
     else {
          effects.changeColour(effects.LightBlue);
          players[turn].score += roll;
          for (int i = 0; i < num; i++) {
               cout     <<     "\t\t\t\t\t\t\t\t\t\t\t\t"     <<
players[i].name << "'s score is: " << players[i].score << endl;
          }
          int prev = players[turn].score; // Store the previous
score
          players[turn].score = g.getDest(players[turn].score);
          effects.changeColour(effects.WHITE);

          if (prev != players[turn].score) {
               effects.changeColour(effects.LightPurple);
               string   s   =   (prev   <   players[turn].score)   ?
"ladder" : "snake";

               cout << endl << players[turn].name << " landed on
a " << s << "! \nTheir score is now " << players[turn].score <<
".\n\n";
          }
     }


     if (players[turn].score == 100) {
          players[turn].win = 1;
          wins++;
          winners.push_back(players[turn]);
          effects.changeColour(effects.LightYellow);
          cout << players[turn].name << " has won!" << endl <<
endl;
     }
     effects.changeColour(effects.WHITE);
}
void PlayerAndBoard::display_board(player* players, int num, int
c1, int c2, AnimAndColour& effects) {
```

```
     char l_top[5] = { char(218),'_','_',char(191) };
     char l_bottom[5] = { char(195),'^','^',char(180) };


     string s_head, s_tail;

     s_head = "~__~";
     s_tail = "<---";


     string board[10][10];
     int row[10]; // Array to store the rows of all players
     int col[10]; // Array to store the columns of all players
     effects.changeColour(effects.WHITE);
     for (int i = 0; i < 10; i++) {
          for (int j = 0; j < 10; j++) {
               int pos;
               ostringstream temp;
               string square;

               if (i % 2 == 0) {
                    pos = 100 - 10 * i - j;
                    temp << pos;
                    square = temp.str();
               }
               else {
                    pos = 100 - 10 * i - (9 - j);
                    temp << pos;
                    square = temp.str();
               }

               board[i][j] = "|" + square + "|";
          }
     }

     board[8][6] = board[6][9] = board[6][2] = board[5][1] =
board[3][6] = board[1][3] = board[0][9] = board[0][1] = l_top;
// Ladder tops
     board[9][0] = board[9][3] = board[9][8] = board[7][0] =
board[7][7] = board[4][9] = board[2][0] = board[2][8] =
l_bottom; // Ladder bottoms

     board[8][3] = board[4][6] = board[3][1] = board[3][3] =
board[1][6] = board[0][7] = board[0][5] = board[0][2] = s_head;
// Snake heads
```

```
     board[9][6] = board[8][1] = board[6][4] = board[6][6] =
board[4][0] = board[2][1] = board[2][5] = board[2][7] = s_tail;
// Snake tails

     for (int i = 0; i < num; i++) {
          int score = players[i].score;
          if (score > 0) {
               row[i] = 9 - (score - 1) / 10;
               col[i] = (score - 1) % 10;
               if (row[i] % 2 == 0) {
                    col[i] = 9 - col[i];
               }
               board[row[i]][col[i]] = "|" + players[i].name +
"|";
          }
     }

     for (int i = 0; i < 10; i++) {
          cout << "\t\t\t\t\t\t";
          for (int j = 0; j < 10; j++) {
               bool isPlayerPosition = false;
               int playerColor = effects.WHITE; // Default color
               for (int k = 0; k < num; k++) {
                    if (i == row[k] && j == col[k]) {
                         isPlayerPosition = true;
                         playerColor = players[k].color; // Use
the player's color
                         break;
                    }
               }
               if (isPlayerPosition) {
                    effects.changeColour(playerColor);
               }
               else if ((i + j) % 2 == 1) {
                    effects.changeColour(c1);
               }
               else {
                    effects.changeColour(c2);
               }
               cout << board[i][j];
          }
          cout << endl;
     }
     effects.changeColour(effects.WHITE);
}
```

SnakesAndLadder.cpp

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <Windows.h>
#include <MMSystem.h>
#include <string>
#include <iomanip>
#include <dos.h>
#include "PlayerAndBoard.cpp"

using namespace std;

int main() {
     srand(static_cast<unsigned int>(time(0)));
     string Answer;
     string plays;
     AnimAndColour effects;
     Graph g;
     PlayerAndBoard Design;
     do {
          for (int i = 0; i < 5; i++) {
//             bool                    music                    =
PlaySound("game-music-teste-204327.wav", NULL, SND_ASYNC);
          }
          effects.showBlinkingLights();
          int num = 0;
          effects.changeColour(effects.AQUA);
          cout << "Would you like to play? (Enter yes or no) "
<< endl;
          effects.changeColour(effects.LightPurple);
          cin >> plays;
          Sleep(1000);
          if (plays == "Yes" || plays == "yes") {
               effects.changeColour(effects.PURPLE);
               cout << "Here are the rules of the game!" <<
endl;
               effects.changeColour(effects.LightGreen);
               cout << "1. The game will continue until all
players have reached a score of 100!" << endl;
               cout << "2. If you reach a ladder, you'll move
upwards! :)" << endl;
               cout << "3. If you reach a snake, you'll move
back down! :(" << endl;
               cout << "4. Have lots of fun and Enjoy!" << endl;
```

```
            effects.changeColour(effects.AQUA);
            cout << "How many players would you like to
register? ";
            Sleep(500);
            effects.changeColour(effects.LightPurple);
            cin >> num;
            player *pls = new player[num];
            vector<player> winners;
            Design.create_players(pls, num, effects);

            int turn = 0;
            int roll = 0;
            int wins = 0;
            string c;


            int c1, c2;

            effects.changeColour(effects.AQUA);

            cout << "\nSelect 2 different colors for the
board to be displayed in. Colors available: " << endl;
            effects.changeColour(effects.BLUE);
            cout << "BLUE = 1" << endl;
            effects.changeColour(effects.GREEN);
            cout << "GREEN = 2" << endl;
            effects.changeColour(effects.AQUA);
            cout << "AQUA = 3" << endl;
            effects.changeColour(effects.RED);
            cout << "RED = 4" << endl;
            effects.changeColour(effects.PURPLE);
            cout << "PURPLE = 5" << endl;
            effects.changeColour(effects.YELLOW);
            cout << "YELLOW = 6" << endl;
            effects.changeColour(effects.LightBlue);
            cout << "Light Blue = 9" << endl;
            effects.changeColour(effects.LightGreen);
            cout << "Light Green = 10 " << endl;
            effects.changeColour(effects.LightPurple);
            cout << "Light Purple = 13" << endl;
            effects.changeColour(effects.WHITE);
            cout << "WHITE = 15" << endl;
            effects.changeColour(effects.LightPurple);
            cin >> c1 >> c2;
            cout << endl;

            char l_top[5] = { char(218),'_','_',char(191) };
```

```
                char l_bottom[5] = { char(195),'^','^',char(180)
};


                effects.changeColour(effects.WHITE);

                string s_head, s_tail;

                s_head = "~__~";
                s_tail = "<---";

                cout << "\t\t\t\t\t\t\t" << "Ladder top: " <<
l_top << endl << endl;
                cout << "\t\t\t\t\t\t\t" << "Ladder bottom: " <<
l_bottom << endl << endl;

                cout << "\t\t\t\t\t\t\t" << "Snake head: " <<
s_head << endl << endl;
                cout << "\t\t\t\t\t\t\t" << "Snake tail: " <<
s_tail << endl << endl;
                while (wins < num) {
                     if (pls[turn].win == 1) {
                          turn = ++turn % num;
                          continue;
                     }

                     else {
                          effects.changeColour(effects.GREEN);
                          cout << endl << "It is now " <<
pls[turn].name << "'s turn to roll." << endl;
                          effects.changeColour(effects.RED);
                               cout << "Press enter to roll the
dice. \n\n";
                               cin.ignore();
                          roll = Design.die_roll();
                          effects.changeColour(effects.PURPLE);
                          cout    <<    "\t\t\t\t\t\t\t\t"    <<
pls[turn].name << " rolled a " << roll << "." << endl;
                          Design.move_player(pls,   turn,   roll,
wins, g, num, winners, effects);
                          Design.display_board(pls, num, c1, c2,
effects);
                          turn = ++turn % num;
                     }

                }
                effects.changeColour(effects.LightYellow);
```

```
                cout << "Scoreboard:" << endl;
                for (int i = 0; i < winners.size(); i++) {
                     cout << "Place " << i + 1 << ": " <<
winners[i].name << endl;
                }
                effects.changeColour(effects.RED);
                cout << "Would you like to play again? (Enter Yes
or No): " << endl;
                cin >> Answer;
           }
           else {

                effects.changeColour(effects.PURPLE);
                cout << "Hope you can play next time! " << endl;
                Sleep(1000);
                return 0;
           }
           effects.changeColour(effects.WHITE);
     } while (((Answer == "Yes" && Answer != "No") || (Answer ==
"yes" && Answer != "no")));
     return 0;
}
```