

**Abdulhaq Zulfiqar   
Zubair Ali  
Zunnon Waheed  
SE-G  
Operating System   
Project Report**

**Contents**

[**Introduction: 5**](#_Toc184440308)

[**Modules and Features 5**](#_Toc184440309)

[**Board Management (board.cpp & board.h): 5**](#_Toc184440310)

[**Gameplay Utilities (gameplay\_utils.cpp & gameplay\_utils.h): 5**](#_Toc184440311)

[**User Interface (gui\_helper.cpp & gui\_helper.h): 6**](#_Toc184440312)

[**Display (display.cpp & display.h): 6**](#_Toc184440313)

[**Main Game Logic (main.cpp): 6**](#_Toc184440314)

[**Player and Team Management (ludo.h): 6**](#_Toc184440315)

[**Code Implementation & Explanation 6**](#_Toc184440316)

[**board.cpp: 6**](#_Toc184440317)

[**display.cpp: 7**](#_Toc184440318)

[**gameplay\_utils.cpp: 7**](#_Toc184440319)

[**gui\_helper.cpp: 7**](#_Toc184440320)

[**main.cpp: 7**](#_Toc184440321)

[**ludo.h: 7**](#_Toc184440322)

[**Phase I and II Pseudocode: 8**](#_Toc184440323)

[**Phase I: 8**](#_Toc184440324)

[**Phase II: 8**](#_Toc184440325)

[**Pseudocode: 8**](#_Toc184440326)

[**Illustrations of Operating System Concepts in Pseudo Code: 9**](#_Toc184440327)

[**Threads and Synchronization: 9**](#_Toc184440328)

[**Mutexes/Locks: 9**](#_Toc184440329)

[**Input/Output Handling: 9**](#_Toc184440330)

[**Memory Management: 9**](#_Toc184440331)

[**Process Management: 9**](#_Toc184440332)

[**Implemented Code: 9**](#_Toc184440333)

[**Board.cpp: 10**](#_Toc184440334)

[**Display.cpp: 24**](#_Toc184440335)

[**gameplay\_utils.cpp: 29**](#_Toc184440336)

[**Main.cpp: 33**](#_Toc184440337)

[**Menu.cpp: 39**](#_Toc184440338)

[**Play.cpp: 46**](#_Toc184440339)

[**Utils.cpp: 52**](#_Toc184440340)

[**gui\_helper.cpp: 56**](#_Toc184440341)

[**Individual File Pseudocode: 57**](#_Toc184440342)

[**Board.cpp: 57**](#_Toc184440343)

[**Purpose: 57**](#_Toc184440344)

[**Pseudocode for display() 57**](#_Toc184440345)

[**Pseudocode for displayTeam() 58**](#_Toc184440346)

[**Pseudocode for initBoard() 59**](#_Toc184440347)

[**Key Operating System (OS) Concepts in the File 59**](#_Toc184440348)

[**Explanation of Code Sections 59**](#_Toc184440349)

[**Display.cpp: 60**](#_Toc184440350)

[**Pseudocode for displayCurrent 60**](#_Toc184440351)

[**Function: displayCurrent 60**](#_Toc184440352)

[**Key Features Explained 61**](#_Toc184440353)

[**Gameplay\_utils.cpp: 62**](#_Toc184440354)

[**Pseudocode for gameplay\_utils.cpp 62**](#_Toc184440355)

[**1. Global Variables and OS Constructs 62**](#_Toc184440356)

[**2. Thread for Player Actions (PlayerThread) 62**](#_Toc184440357)

[**3. Master Thread for Game Logic 63**](#_Toc184440358)

[**4. Helper Functions 64**](#_Toc184440359)

[**OS Concepts Demonstrated 66**](#_Toc184440360)

[**Code Description 66**](#_Toc184440361)

[**Main.cpp: 67**](#_Toc184440362)

[**Key Functions 67**](#_Toc184440363)

[**Pseudocode 69**](#_Toc184440364)

[**main() 70**](#_Toc184440365)

[**Flow Summary 71**](#_Toc184440366)

[**Menu.cpp: 71**](#_Toc184440367)

[**1. Main Function and Menu Display 72**](#_Toc184440368)

[**2. Configuring Players and Tokens 72**](#_Toc184440369)

[**3. Team Configuration (for Team Play) 73**](#_Toc184440370)

[**4. Flashing Text for Menu Options 73**](#_Toc184440371)

[**5. Center Text Display 74**](#_Toc184440372)

[**6. Drawing Horizontal Lines 74**](#_Toc184440373)

[**7. User Input Validation 74**](#_Toc184440374)

[**Operating System Concepts Highlighted 75**](#_Toc184440375)

[**Play.cpp: 75**](#_Toc184440376)

[**Description of Code: 77**](#_Toc184440377)

[**play function: 77**](#_Toc184440378)

[**makeTeams function: 78**](#_Toc184440379)

[**Teamplay function: 78**](#_Toc184440380)

[**Utils.cpp: 79**](#_Toc184440381)

[**Pseudocode 79**](#_Toc184440382)

[**Description 80**](#_Toc184440383)

[**gui\_helper.cpp: 81**](#_Toc184440384)

[**Pseudocode 81**](#_Toc184440385)

[**Description 82**](#_Toc184440386)

[**Use Case and Flow: 83**](#_Toc184440387)

[**Illustrations of Operating System Concepts in Pseudocode: 83**](#_Toc184440388)

[** Threading: 83**](#_Toc184440389)

[** Semaphores for Synchronization: 83**](#_Toc184440390)

[** Process Management: 84**](#_Toc184440391)

[**System Specifications: 84**](#_Toc184440392)

[** Hardware/Software Requirements: 84**](#_Toc184440393)

[** Game Rules/Logic: 84**](#_Toc184440394)

[** Game Flow: 84**](#_Toc184440395)

[**Short Paragraph on Implementation in Another Scenario: 84**](#_Toc184440396)

[** Threading: 84**](#_Toc184440397)

[** Semaphores: 84**](#_Toc184440398)

[**Another Example: 84**](#_Toc184440399)

[**Teamwork Division 85**](#_Toc184440400)

[**Conclusion 85**](#_Toc184440401)

Ludo Game

# Introduction:

The **Ludo Game** is a multiplayer board game that simulates the classic board game in a C++ console application. The game supports up to four players, with each player having four tokens. The game board has defined paths, homes, and safe points, with players competing to move their tokens to the center of the board. The project is structured into several modules, each handling different aspects of the game, such as display, gameplay, team management, and user interface.

The goal of this documentation is to provide a comprehensive overview of the system's architecture, functionality, and code implementation, including the roles of each team member involved in the development of this project.

# Modules and Features

This Ludo game consists of several core modules, each responsible for specific functionality. These modules are:

## Board Management (board.cpp & board.h):

* + Handles the initialization of the game board and manages the movement of tokens across the board.
  + Contains functions like initBoard(), initGame(), and display().
  + Handles the game state, displaying the game board, paths, and player tokens.

## Gameplay Utilities (gameplay\_utils.cpp & gameplay\_utils.h):

* + Responsible for game logic such as player turns, dice rolls, checking for valid moves, and determining safe points.
  + Functions such as moveToken(), isBlock(), and isAnyMovePossible() are included.

## User Interface (gui\_helper.cpp & gui\_helper.h):

* + Manages the terminal user interface, including screen clearing, delayed text printing, and printing centered or horizontal lines with colors.

## Display (display.cpp & display.h):

* + Responsible for rendering the current state of the game on the console, including board layout, player tokens, and game status.
  + Implements colorful and dynamic board rendering based on the player and their tokens.

## Main Game Logic (main.cpp):

* + Initializes the game, resets the state, and sets up the environment for gameplay.
  + Controls the flow of the game by handling player interactions and updating the game state.
  + Contains functions like resetAll() and splashScreen().

## Player and Team Management (ludo.h):

* + Contains the definitions for Player and Cell classes.
  + Manages the teams, players, and their tokens on the board.
  + Includes global variables and function prototypes for player-related operations.

# Code Implementation & Explanation

Each file in the project plays an important role in making the game interactive and functional. Here is an explanation of the key files and their respective functionalities:

## board.cpp:

* + Implements the logic for displaying and managing the board layout. It includes functions like initBoard() to initialize the board, initGame() to reset the game state, and display() to render the board state with updated player tokens.

## display.cpp:

* + Contains the displayCurrent() function, which renders the game board along with player tokens on the console. The board is dynamically updated, and players are represented with colorful tokens based on their team colors.
  + Utilizes a time delay to enhance the user experience and make the game look smoother.

## gameplay\_utils.cpp:

* + Includes core gameplay functionalities, such as moving tokens, checking for valid moves, and determining whether a player can proceed.
  + The PlayerThread() and Masterthread() functions control the flow of the game in a multi-threaded environment.

## gui\_helper.cpp:

* + Helps in handling user interface elements, such as printing text with a delay, centering text on the screen, and drawing horizontal lines for better aesthetics.

## main.cpp:

* + Manages the overall game flow by resetting game states, initiating threads, and controlling the game loop.
  + Displays the splash screen and calls relevant functions from other modules to maintain game logic.

## ludo.h:

* + Defines the constants, global variables, player, and cell structures used throughout the game.
  + Provides function prototypes for handling gameplay operations and player interactions.

# Phase I and II Pseudocode:

Phase I:

Phase I typically involves initializing and setting up the game (e.g., initializing the board, players, and game state). This can be outlined in pseudocode as steps to initialize objects, variables, and display the game board.  
Pseudocode:

1. Initialize the board with required cells and player positions.

2. Setup semaphore for synchronization between threads.

3. Display a splash screen with game instructions.

4. Input player names and setup teams.

5. Initialize the game state (tokens, houses, dice rolls).

6. Display the initial board setup.

Phase II:

Phase II involves gameplay mechanics (e.g., dice rolls, token movements, turns, etc.). You can create pseudocode for the logic behind player turns, rolling dice, validating moves, and updating the game state.

Pseudocode:

1. Display the current game state on the board.

2. Roll the dice for each player and update the token positions.

3. Check if the player can move based on the dice roll.

4. Move the player's token to the new position.

5. If the player lands on an opponent's token, 'kill' that token and move it back to the starting point.

6. Check if the player has reached the winning position.

7. Repeat the cycle until a player wins.

8. Display the game results.

# Illustrations of Operating System Concepts in Pseudo Code:

The following are some OS concepts used in your code and their illustration:

## Threads and Synchronization:

* + Use of threads to simulate multiple players: PlayerThread and MasterThread.
  + **Semaphore (sem)** is used to synchronize access to shared resources like the board to prevent race conditions during simultaneous player actions.

## Mutexes/Locks:

* + Semaphore acts as a lock mechanism to prevent multiple threads from accessing the shared game state simultaneously, ensuring thread-safe operations.

## Input/Output Handling:

* + Dynamic board display and user inputs (such as dice rolls and token moves) are managed interactively through the terminal, utilizing threading and synchronization.

## Memory Management:

* + Memory is dynamically allocated and deallocated (e.g., isfinished, diceroll).

## Process Management:

* + Player actions are managed using threads, with the main game loop being controlled by a master thread (MasterThread), which coordinates the gameplay.

Implemented Code:

The code of various components (e.g., board.cpp, gameplay\_utils.cpp, etc.) demonstrates the functionality of the Ludo game, with multi-threading for handling player interactions and game synchronization.

## Board.cpp:

#include "board.h" // Include the header for board functions

#include <iostream>

//#include <SFML/Graphics.hpp>

using namespace std;

void display() {

// Clear screen for a clean display

cout << "\033[2J\033[1;1H";

// Decorative header

cout << WHITE << string(50, '=') << RESET << endl;

cout << YELLOW <<

R"(

" GAME RESULTS "

)" << RESET << endl;

cout << WHITE << string(60, '=') << RESET << endl;

// Game Statistics

cout << WHITE << "🎲 Game Statistics:" << RESET << endl;

cout << "Total Tokens per Player: " << GREEN << n\_tokens << RESET << endl;

cout << "Total Players: " << GREEN << n\_players << RESET << endl;

// Detailed Player Performance

cout << "\n" << WHITE << "📊 Player Performance Breakdown:" << RESET << endl;

// Sort players by position for more meaningful display

vector<pair<int, int>> player\_ranks;

for (int i = 0; i < n\_players; i++) {

player\_ranks.push\_back({players[i][0].position, i});

}

sort(player\_ranks.begin(), player\_ranks.end());

// Color codes for different ranks

const string rank\_colors[] = {

YELLOW, // 1st

WHITE, // 2nd

"\033[38;5;208m" // 3rd (bronze-like)

};

// Detailed player statistics

for (auto& rank : player\_ranks) {

int i = rank.second;

string rank\_display;

string color = RESET;

if (players[i][0].position == 1) {

rank\_display = "🥇 1st Place";

color = rank\_colors[0];

} else if (players[i][0].position == 2) {

rank\_display = "🥈 2nd Place";

color = rank\_colors[1];

} else if (players[i][0].position == 3) {

rank\_display = "🥉 3rd Place";

color = rank\_colors[2];

} else if (players[i][0].position == -1) {

rank\_display = "❌ Eliminated";

color = RED;

} else {

rank\_display = "🏁 Finished";

color = GREEN;

}

cout << color

<< "Player " << i + 1 << " (" << players[i][0].name << "):" << endl;

cout << " 🏆 Final Position: " << rank\_display << endl;

// Performance metrics

cout << " 💥 Tokens Eliminated: "

<< RED << players[i][0].removed << RESET << endl;

cout << " 🔥 Hits: " << CYAN << players[i][0].hits << RESET << endl; // Added hits display

// Completion status

if (players[i][0].position != -1) {

cout << " 🚩 Game Completion Status: "

<< (players[i][0].position <= n\_players ? "Completed" : "Ongoing")

<< RESET << endl;

}

cout << RESET << endl;

}

// Cancelled Threads Section

cout << WHITE << "🚫 Cancelled Threads:" << RESET << endl;

bool has\_cancelled\_threads = false;

for (int i = 0; i < n\_players; i++) {

if (players[i][0].position == -1) {

cout << RED

<< " 🔴 Player " << i + 1 << " (" << players[i][0].name

<< "): Thread Cancelled" << RESET << endl;

has\_cancelled\_threads = true;

}

}

if (!has\_cancelled\_threads) {

cout << GREEN << " ✅ No Threads Cancelled" << RESET << endl;

}

// Final Game Over Display

cout << "\n" << WHITE << string(60, '=') << RESET << endl;

cout << YELLOW <<

R"(

)" << RESET << endl;

cout << WHITE << string(60, '=') << RESET << endl;

// Optional: Sound or visual alert (platform-dependent)

cout << "\a"; // Audible alert (console bell)

// Wait for user input to restart the game

cout << endl << GREEN << "Press Enter to restart the game..." << RESET;

cin.ignore(); // Ignore any remaining input in the buffer

cin.get();

}

void displayTeam() {

// Clear screen for a clean display

cout << "\033[2J\033[1;1H";

cout << WHITE << string(50, '=') << RESET << endl;

cout << YELLOW <<

)" << RESET << endl;

cout << WHITE << string(50, '=') << RESET << endl;

// Game Summary

cout << WHITE << "🎲 Game Summary:" << RESET << endl;

// Display winning team details

for (int team = 0; team < n\_teams; team++) {

int teamScore = 0;

for (int i = 0; i < n\_players; i++) {

if (players[i][0].team\_member == team) {

teamScore += players[i][0].position;

}

}

// Show the winning team and players

if (teamScore == winning\_position) {

cout << GREEN << "🏆 Winning Team " << team + 1 << " 🏆" << RESET << endl;

cout << " Players in Team " << team + 1 << ":" << endl;

for (int i = 0; i < n\_players; i++) {

if (players[i][0].team\_member == team) {

cout << " - Player " << i + 1 << ": " << players[i][0].name << endl;

}

}

}

}

// Show cancelled threads (if any)

cout << WHITE << "🚫 Cancelled Threads:" << RESET << endl;

bool hasCancelled = false;

for (int i = 0; i < n\_players; i++) {

if (players[i][0].position == -1) {

cout << RED << " 🔴 Player " << i + 1 << " (" << players[i][0].name << "): Thread Cancelled" << RESET << endl;

hasCancelled = true;

}

}

if (!hasCancelled) {

cout << GREEN << " ✅ No Threads Cancelled" << RESET << endl;

}

// Show token count and hit rates

cout << WHITE << "🎯 Tokens and Hit Rates:" << RESET << endl;

for (int team = 0; team < n\_teams; team++) {

int teamTokens = 0;

int teamHits = 0;

for (int i = 0; i < n\_players; i++) {

if (players[i][0].team\_member == team) {

teamTokens += players[i][0].removed;

teamHits += players[i][0].hits;

}

}

cout << " Team " << team + 1 << ":" << endl;

cout << " Total Tokens: " << teamTokens << endl;

cout << " Total Hits: " << teamHits << endl;

}

// Final Game Over Display

cout << "\n" << WHITE << string(60, '=') << RESET << endl;

cout << YELLOW <<

R"(

███████╗███╗ ██╗██████╗

)" << RESET << endl;

cout << WHITE << string(60, '=') << RESET << endl;

// Optional: Sound or visual alert (platform-dependent)

cout << "\a"; // Audible alert (console bell)

// Wait for user input to restart the game

cout << endl << GREEN << "Press Enter to restart the game..." << RESET;

cin.ignore(); // Ignore any remaining input in the buffer

cin.get();

}

void initBoard()

{

char newBoard[SIZE][SIZE] = {

{'r', 'r', 'r', 'r', 'r', 'r', 'O', 'O', 'O', 'g', 'g', 'g', 'g', 'g', 'g'},

{'r', ' ', ' ', ' ', ' ', 'r', 'O', 'g', 'g', 'g', ' ', ' ', ' ', ' ', 'g'},

{'r', ' ', 'r', 'r', ' ', 'r', 'g', 'g', 'O', 'g', ' ', 'g', 'g', ' ', 'g'},

{'r', ' ', 'r', 'r', ' ', 'r', 'O', 'g', 'O', 'g', ' ', 'g', 'g', ' ', 'g'},

{'r', ' ', ' ', ' ', ' ', 'r', 'O', 'g', 'O', 'g', ' ', ' ', ' ', ' ', 'g'},

{'r', 'r', 'r', 'r', 'r', 'r', 'O', 'g', 'O', 'g', 'g', 'g', 'g', 'g', 'g'},

{'O', 'r', 'O', 'O', 'O', 'O', ' ', ' ', ' ', 'O', 'O', 'O', 'b', 'O', 'O'},

{'O', 'r', 'r', 'r', 'r', 'r', ' ', 'H', ' ', 'b', 'b', 'b', 'b', 'b', 'O'},

{'O', 'O', 'r', 'O', 'O', 'O', ' ', ' ', ' ', 'O', 'O', 'O', 'O', 'b', 'O'},

{'y', 'y', 'y', 'y', 'y', 'y', 'O', 'y', 'O', 'b', 'b', 'b', 'b', 'b', 'b'},

{'y', ' ', ' ', ' ', ' ', 'y', 'O', 'y', 'O', 'b', ' ', ' ', ' ', ' ', 'b'},

{'y', ' ', 'y', 'y', ' ', 'y', 'O', 'y', 'O', 'b', ' ', 'b', 'b', ' ', 'b'},

{'y', ' ', 'y', 'y', ' ', 'y', 'O', 'y', 'y', 'b', ' ', 'b', 'b', ' ', 'b'},

{'y', ' ', ' ', ' ', ' ', 'y', 'y', 'y', 'O', 'b', ' ', ' ', ' ', ' ', 'b'},

{'y', 'y', 'y', 'y', 'y', 'y', 'O', 'O', 'O', 'b', 'b', 'b', 'b', 'b', 'b'},

};

for (int i = 0; i < SIZE; i++)

{

for (int j = 0; j < SIZE; j++)

{

board[i][j] = newBoard[i][j];

}

}

}

void initGame()

{

int newCells[CELL\_NUMBER][2] = {

{6, 1}, {6, 2}, {6, 3}, {6, 4}, {6, 5},

{5, 6}, {4, 6}, {3, 6}, {2, 6}, {1, 6}, {0, 6}, {0, 7},

{0, 8}, {1, 8}, {2, 8}, {3, 8}, {4, 8}, {5, 8},

{6, 9}, {6, 10}, {6, 11}, {6, 12}, {6, 13}, {6, 14}, {7, 14},

{8, 14}, {8, 13}, {8, 12}, {8, 11}, {8, 10}, {8, 9},

{9, 8}, {10, 8}, {11, 8}, {12, 8}, {13, 8}, {14, 8}, {14, 7},

{14, 6}, {13, 6}, {12, 6}, {11, 6}, {10, 6}, {9, 6},

{8, 5}, {8, 4}, {8, 3}, {8, 2}, {8, 1}, {8, 0}, {7, 0}, {6, 0}};

int newHouses[MAX\_PLAYER][4][2] = {

{{2, 2}, {2, 3}, {3, 3}, {3, 2}},

{{2, 11}, {2, 12}, {3, 12}, {3, 11}},

{{11, 11}, {11, 12}, {12, 12}, {12, 11}},

{{11, 2}, {11, 3}, {12, 3}, {12, 2}},

};

for (int i = 0; i < CELL\_NUMBER; i++)

{

cells[i].y = newCells[i][0];

cells[i].x = newCells[i][1];

cells[i].value = 0;

}

for (int i = 0; i < MAX\_PLAYER; i++)

{

for (int j = 0; j < 4; j++)

{

houses[i][j].y = newHouses[i][j][0];

houses[i][j].x = newHouses[i][j][1];

houses[i][j].value = (j + 1) + (i)\*10;

}

}

for (int i = 0; i < MAX\_PLAYER; i++)

{

for (int j = 0; j < 4; j++)

{

players[i][j].x = houses[i][j].x;

players[i][j].y = houses[i][j].y;

players[i][j].index = -1;

players[i][j].team = i;

players[i][j].team\_member = i / 2; // Assuming 2 players per team

players[i][j].id = j + 1;

players[i][j].removed = 0;

players[i][j].position = n\_players;

players[i][j].counter = 0;

players[i][j].ingame = (i < n\_players);

players[i][j].hits = 0;

}

}

}

/\*

void renderBoard(sf::RenderWindow &window, const char board[SIZE][SIZE], const Player players[MAX\_PLAYER][4]) {

// Create a background rectangle to give the board a defined background

sf::RectangleShape background(sf::Vector2f(SIZE \* CELL\_SIZE, SIZE \* CELL\_SIZE));

background.setFillColor(sf::Color(255, 255, 255)); // White background for clean appearance

window.draw(background);

// Highlight path and home for each player

for (int row = 0; row < SIZE; row++) {

for (int col = 0; col < SIZE; col++) {

sf::RectangleShape cell(sf::Vector2f(CELL\_SIZE, CELL\_SIZE));

cell.setPosition(col \* CELL\_SIZE, row \* CELL\_SIZE);

// Set cell color based on board content

switch (board[row][col]) {

case 'O':

cell.setFillColor(sf::Color(230, 230, 230)); // Light gray for open spaces

break;

case 'r':

cell.setFillColor(sf::Color(255, 0, 0)); // Red for one type of cell (houses)

break;

case 'g':

cell.setFillColor(sf::Color(0, 255, 0)); // Green for another cell type

break;

case 'b':

cell.setFillColor(sf::Color(0, 0, 255)); // Blue for another type

break;

case 'y':

cell.setFillColor(sf::Color(255, 255, 0)); // Yellow for different cell

break;

default:

cell.setFillColor(sf::Color(255, 255, 255)); // Default to white for uninitialized or empty spaces

break;

}

// Highlight home sections with a colored border

if ((row == 7 && col >= 6 && col <= 10) || (row == 6 && col == 6)) { // Home section for player 1

cell.setOutlineThickness(3);

cell.setOutlineColor(sf::Color(255, 0, 0)); // Red border for player 1's home area

} else if ((row == 7 && col >= 11 && col <= 14) || (row == 6 && col == 11)) { // Home section for player 2

cell.setOutlineThickness(3);

cell.setOutlineColor(sf::Color(0, 255, 0)); // Green border for player 2's home area

} else if ((row == 12 && col >= 11 && col <= 14) || (row == 13 && col == 11)) { // Home section for player 3

cell.setOutlineThickness(3);

cell.setOutlineColor(sf::Color(0, 0, 255)); // Blue border for player 3's home area

} else if ((row == 12 && col >= 6 && col <= 10) || (row == 13 && col == 6)) { // Home section for player 4

cell.setOutlineThickness(3);

cell.setOutlineColor(sf::Color(255, 255, 0)); // Yellow border for player 4's home area

}

// Add a subtle outline to all cells

cell.setOutlineThickness(1);

cell.setOutlineColor(sf::Color(200, 100, 250)); // Light gray border for all cells

window.draw(cell);

}

}

}

void renderTokens(sf::RenderWindow &window, const Player players[MAX\_PLAYER][4]) {

for (int i = 0; i < MAX\_PLAYER; i++) {

for (int j = 0; j < 4; j++) {

if (players[i][j].ingame) {

sf::CircleShape token(CELL\_SIZE / 2.5f); // Smaller token size for better fit

token.setPosition(players[i][j].x \* CELL\_SIZE + CELL\_SIZE / 4,

players[i][j].y \* CELL\_SIZE + CELL\_SIZE / 4); // Center the token in cell

// Set color of the token based on player team

switch (i) {

case 0:

token.setFillColor(sf::Color(255, 0, 0)); // Red player

break;

case 1:

token.setFillColor(sf::Color(0, 255, 0)); // Green player

break;

case 2:

token.setFillColor(sf::Color(0, 0, 255)); // Blue player

break;

case 3:

token.setFillColor(sf::Color(255, 255, 0)); // Yellow player

break;

}

// Add some effects to the token for realism

token.setOutlineThickness(2);

token.setOutlineColor(sf::Color(0, 0, 0)); // Black border for visibility

window.draw(token); // Draw the token

}

}

}

// Render path highlights for each player (adjust logic as per game path)

for (int i = 0; i < MAX\_PLAYER; i++) {

for (int j = 0; j < 4; j++) {

if (players[i][j].ingame) {

sf::RectangleShape pathHighlight(sf::Vector2f(CELL\_SIZE, CELL\_SIZE));

pathHighlight.setPosition(players[i][j].x \* CELL\_SIZE, players[i][j].y \* CELL\_SIZE);

// Highlight the path with the color matching the player's team

switch (i) {

case 0:

pathHighlight.setFillColor(sf::Color(255, 0, 0, 100)); // Red path with transparency

break;

case 1:

pathHighlight.setFillColor(sf::Color(0, 255, 0, 100)); // Green path with transparency

break;

case 2:

pathHighlight.setFillColor(sf::Color(0, 0, 255, 100)); // Blue path with transparency

break;

case 3:

pathHighlight.setFillColor(sf::Color(255, 255, 0, 100)); // Yellow path with transparency

break;

}

window.draw(pathHighlight); // Draw the path highlight

}

}

}

}

\*/

## Display.cpp:

#include "display.h"

#include <iostream>

#include <string>

#include <thread> // For this\_thread

#include <chrono> // For chrono

using namespace std;

void displayCurrent() {

// Clear screen for a fresh display

cout << "\033[2J\033[1;1H";

// Add a decorative border

cout << WHITE << string(50, '=') << RESET << endl;

cout << WHITE << " 🎲 LUDO GAME ROUND 🎲" << RESET << endl;

cout << WHITE << string(50, '=') << RESET << endl;

char currentBoard[SIZE][SIZE];

// Copy the board

for (int i = 0; i < SIZE; i++) {

for (int j = 0; j < SIZE; j++) {

currentBoard[i][j] = board[i][j];

}

}

// Place player tokens

for (int i = 0; i < MAX\_PLAYER; i++) {

for (int j = 0; j < 4; j++) {

currentBoard[players[i][j].y][players[i][j].x] = '0' + players[i][j].id;

}

}

// Enhanced board display

for (int i = 0; i < SIZE; i++) {

cout << " ";

for (int j = 0; j < SIZE; j++) {

string s = "";

s = s + currentBoard[i][j] + " ";

// Colorful and dynamic board rendering

switch (currentBoard[i][j]) {

case 'r': // Red home

s = "■ ";

s = RED + s + RESET;

break;

case 'g': // Green home

s = "■ ";

s = GREEN + s + RESET;

break;

case 'b': // Blue home

s = "■ ";

s = BLUE + s + RESET;

break;

case 'y': // Yellow home

s = "■ ";

s = YELLOW + s + RESET;

break;

case 'O': // Neutral zones

s = "◊ ";

s = WHITE + s + RESET;

break;

case 'H': // Neutral zones

s = "H ";

s = DARK\_ORANGE + s + RESET;

break;

case '1':

case '2':

case '3':

case '4':

s = "";

s = s + currentBoard[i][j] + " ";

for (int a = 0; a < MAX\_PLAYER; a++) {

for (int b = 0; b < 4; b++) {

if (i == players[a][b].y && j == players[a][b].x) {

// More vibrant token colors

switch (players[a][b].team) {

case 0:

s = RED + s + RESET;

break;

case 1:

s = GREEN + s + RESET;

break;

case 2:

s = BLUE + s + RESET;

break;

case 3:

s = YELLOW + s + RESET;

break;

}

}

}

}

break;

default:

s = ". ";

s = MAGENTA + s + RESET;

}

cout << s;

}

cout << endl;

}

// Add some game stats or current player information

cout << endl;

cout << WHITE << "Current Game Status:" << RESET << endl;

for (int i = 0; i < MAX\_PLAYER; i++) {

string playerColor;

switch (i) {

case 0: playerColor = RED; break;

case 1: playerColor = GREEN; break;

case 2: playerColor = BLUE; break;

case 3: playerColor = YELLOW; break;

}

cout << playerColor << "Player " << i+1 << ": "

<< players[i][0].name << RESET << endl;

}

// Optional: Add a subtle flashing effect

this\_thread::sleep\_for(chrono::seconds(1));

}

## gameplay\_utils.cpp:

#include "display.h"

#include <iostream>

#include <string>

#include <thread> // For this\_thread

#include <chrono> // For chrono

using namespace std;

void displayCurrent() {

// Clear screen for a fresh display

cout << "\033[2J\033[1;1H";

// Add a decorative border

cout << WHITE << string(50, '=') << RESET << endl;

cout << WHITE << " 🎲 LUDO GAME ROUND 🎲" << RESET << endl;

cout << WHITE << string(50, '=') << RESET << endl;

char currentBoard[SIZE][SIZE];

// Copy the board

for (int i = 0; i < SIZE; i++) {

for (int j = 0; j < SIZE; j++) {

currentBoard[i][j] = board[i][j];

}

}

// Place player tokens

for (int i = 0; i < MAX\_PLAYER; i++) {

for (int j = 0; j < 4; j++) {

currentBoard[players[i][j].y][players[i][j].x] = '0' + players[i][j].id;

}

}

// Enhanced board display

for (int i = 0; i < SIZE; i++) {

cout << " ";

for (int j = 0; j < SIZE; j++) {

string s = "";

s = s + currentBoard[i][j] + " ";

// Colorful and dynamic board rendering

switch (currentBoard[i][j]) {

case 'r': // Red home

s = "■ ";

s = RED + s + RESET;

break;

case 'g': // Green home

s = "■ ";

s = GREEN + s + RESET;

break;

case 'b': // Blue home

s = "■ ";

s = BLUE + s + RESET;

break;

case 'y': // Yellow home

s = "■ ";

s = YELLOW + s + RESET;

break;

case 'O': // Neutral zones

s = "◊ ";

s = WHITE + s + RESET;

break;

case 'H': // Neutral zones

s = "H ";

s = DARK\_ORANGE + s + RESET;

break;

case '1':

case '2':

case '3':

case '4':

s = "";

s = s + currentBoard[i][j] + " ";

for (int a = 0; a < MAX\_PLAYER; a++) {

for (int b = 0; b < 4; b++) {

if (i == players[a][b].y && j == players[a][b].x) {

// More vibrant token colors

switch (players[a][b].team) {

case 0:

s = RED + s + RESET;

break;

case 1:

s = GREEN + s + RESET;

break;

case 2:

s = BLUE + s + RESET;

break;

case 3:

s = YELLOW + s + RESET;

break;

}

}

}

}

break;

default:

s = ". ";

s = MAGENTA + s + RESET;

}

cout << s;

}

cout << endl;

}

// Add some game stats or current player information

cout << endl;

cout << WHITE << "Current Game Status:" << RESET << endl;

for (int i = 0; i < MAX\_PLAYER; i++) {

string playerColor;

switch (i) {

case 0: playerColor = RED; break;

case 1: playerColor = GREEN; break;

case 2: playerColor = BLUE; break;

case 3: playerColor = YELLOW; break;

}

cout << playerColor << "Player " << i+1 << ": "

<< players[i][0].name << RESET << endl;

}

// Optional: Add a subtle flashing effect

this\_thread::sleep\_for(chrono::seconds(1));

}

## Main.cpp:

#include "ludo.h"

#include "board.h"

#include "utils.h"

#include "menu.h"

#include "play.h"

#include <iostream>

#include <thread>

#include <chrono>

#include <iomanip>

#include <vector>

#include <cstdlib>

#include <ctime>

#include "gui\_helper.h"

using namespace std;

// Function to reset the game state

void resetAll() {

// Reset global variables

sem\_destroy(&sem);

sem\_init(&sem, 0, 1);

position = 1;

winning\_position = 1;

// Reset board and game state arrays

for (int i = 0; i < CELL\_NUMBER; i++) {

cells[i] = Cell();

}

for (int i = 0; i < MAX\_PLAYER; i++) {

for (int j = 0; j < 4; j++) {

players[i][j] = Player();

houses[i][j] = Cell();

}

}

for (int i = 0; i < SIZE; i++) {

for (int j = 0; j < SIZE; j++) {

board[i][j] = ' ';

}

}

// Reset dynamic arrays if allocated

if (isfinished != nullptr) {

delete[] isfinished;

isfinished = new bool[MAX\_PLAYER]();

}

if (diceroll != nullptr) {

delete[] diceroll;

diceroll = new int[3]();

}

cout << GREEN << "Game state reset successfully." << RESET << endl;

}

// Splash Screen Implementation

void splashScreen() {

clearScreen();

// Colorful border

horizontalLine('\*', 80, MAGENTA);

// Enhanced Ludo ASCII Art

string ludoArt = R"(

)";

string gameArt = R"(

)";

// Center and color the ASCII arts with safe color application

centerText(ludoArt);

centerText(gameArt);

// Fancy welcome banner

horizontalLine('=', 80, CYAN);

centerText("🎲 WELCOME TO THE ULTIMATE LUDO ADVENTURE! 🎲", RED);

horizontalLine('=', 80, CYAN);

// Loading sequence with creative messages

vector<pair<string, string>> loadingMessages = {

{YELLOW, "🎲 Preparing Epic Battlefield... 🎲"},

{GREEN, "🏁 Assembling Legendary Game Tokens... 🏁"},

{BLUE, "🧩 Generating Mystical Player Zones... 🧩"},

{MAGENTA, "🌟 Charging Cosmic Dice Power... 🌟"}

};

for (const auto& [color, msg] : loadingMessages) {

centerText(msg);

this\_thread::sleep\_for(chrono::milliseconds(600));

}

// Final activation message

horizontalLine('-', 80, GREEN);

centerText("✨ LUDO ARENA ACTIVATED! BATTLE COMMENCES! ✨", RED);

horizontalLine('-', 80, GREEN);

// Wait before clearing screen

this\_thread::sleep\_for(chrono::seconds(3));

clearScreen();

}

// Global Variable Initialization

sem\_t sem;

Cell cells[CELL\_NUMBER];

Cell houses[MAX\_PLAYER][4];

char board[SIZE][SIZE];

int n\_players = -1;

int n\_teams = -1;

Player players[MAX\_PLAYER][4];

int position = 1;

int winning\_position = 1;

int n\_tokens = -1;

bool \*isfinished = nullptr;

int \*diceroll = nullptr;

int teams[MAX\_TEAM][2];

bool gameOver = false;

int main() {

// Memory allocation with error checking

try {

isfinished = new bool[MAX\_PLAYER](); // Zero-initialize

diceroll = new int[3](); // Zero-initialize

} catch (const std::bad\_alloc& e) {

cerr << "Memory allocation failed: " << e.what() << endl;

return 1;

}

// Semaphore and random seed initialization

sem\_init(&sem, 0, 1);

srand(time(NULL));

// Splash Screen

splashScreen();

while (!gameOver) {

resetAll();

// Initialize the board

initBoard();

int choice = -1;

menu(choice);

if (choice == 1) {

play();

display();

} else if (choice == 2) {

Teamplay();

} else if (choice == 3) {

cout << endl << RED << "Stopping the game..." << RESET << endl;

break;

}

if (gameOver) {

break; // Exit the game loop

}

}

// Safe memory cleanup

delete[] isfinished;

delete[] diceroll;

return 0;

}

## Menu.cpp:

#include "ludo.h"

#include "utils.h"

#include "menu.h"

#include "gui\_helper.h"

void drawLudoLogo() {

string ludoLogo = R"(

)";

centerText(ludoLogo, MAGENTA);

}

void TeamdrawLudoLogo() {

string ludoLogo = R"(

)";

centerText(ludoLogo, MAGENTA);

}

void flashingText(const string& text, int flashes = 3, int delay = 300) {

for (int i = 0; i < flashes; ++i) {

// Alternate between colors

if (i % 2 == 0) {

cout << "\r" << YELLOW << text << RESET << flush;

} else {

cout << "\r" << GREEN << text << RESET << flush;

}

this\_thread::sleep\_for(chrono::milliseconds(delay));

cout << "\r" << string(text.length(), ' ') << "\r" << flush;

this\_thread::sleep\_for(chrono::milliseconds(delay));

}

// Final display

cout << "\r" << BRIGHT\_YELLOW << text << RESET << endl;

}

void menu(int &choice) {

while (true) {

// Clear screen and draw decorative elements

clearScreen();

// Draw Ludo Logo

drawLudoLogo();

// Decorative lines

horizontalLine('=', 80, CYAN);

// Menu Title with Flashing Effect

flashingText("🎲 LUDO GAME MENU 🎲", 3, 300);

// Decorative lines

horizontalLine('-', 80, GREEN);

// Menu Options with Colorful Design

vector<string> menuOptions = {

"1. START GAME 🚀",

"2. MAKE TEAMS 👥",

"3. QUIT GAME 🏳️"

};

// Display Menu Options

for (const auto& option : menuOptions) {

centerText(option, YELLOW);

}

// Decorative lines

horizontalLine('-', 80, RED);

// Prompt for Choice

cout << BRIGHT\_YELLOW;

centerText("Enter your choice (1-2): ");

cout << RESET;

// Input with error handling

cout << GREEN << "> " << RESET;

cin >> choice;

verifyInput();

// Validate main menu choice

if (choice >= 1 && choice <= 2) {

break;

}

// Invalid choice handling with flashing error

centerText("INVALID CHOICE! Please try again.", RED);

this\_thread::sleep\_for(chrono::seconds(1));

}

// Player and Token Configuration

if (choice == 1) {

// Player Number Selection

while (true) {

clearScreen();

drawLudoLogo();

horizontalLine('=', 80, CYAN);

centerText("PLAYER CONFIGURATION", MAGENTA);

horizontalLine('-', 80, GREEN);

cout << YELLOW;

centerText("Select Number of Players (2-4)");

cout << RESET;

cout << GREEN << "> " << RESET;

cin >> n\_players;

verifyInput();

if (n\_players >= 2 && n\_players <= 4) {

break;

}

centerText("INVALID PLAYERS! Choose between 2-4.", RED);

this\_thread::sleep\_for(chrono::seconds(1));

}

// Token Number Selection

while (true) {

clearScreen();

drawLudoLogo();

horizontalLine('=', 80, CYAN);

centerText("TOKEN CONFIGURATION", MAGENTA);

horizontalLine('-', 80, GREEN);

cout << YELLOW;

centerText("Select Number of Tokens (1-4)");

cout << RESET;

cout << GREEN << "> " << RESET;

cin >> n\_tokens;

verifyInput();

if (n\_tokens >= 1 && n\_tokens <= 4) {

break;

}

centerText("INVALID TOKENS! Choose between 1-4.", RED);

this\_thread::sleep\_for(chrono::seconds(1));

}

}

else if (choice == 2) {

n\_players = 4; // Fixed for team play

// Token Number Selection

while (true) {

clearScreen();

TeamdrawLudoLogo();

horizontalLine('=', 80, CYAN);

centerText("TOKEN CONFIGURATION", MAGENTA);

horizontalLine('-', 80, GREEN);

cout << YELLOW;

centerText("Select Number of Tokens (1-4)");

cout << RESET;

cout << GREEN << "> " << RESET;

cin >> n\_tokens;

verifyInput();

if (n\_tokens >= 1 && n\_tokens <= 4) {

break;

}

centerText("INVALID TOKENS! Choose between 1-4.", RED);

this\_thread::sleep\_for(chrono::seconds(1));

} }

}

## Play.cpp:

#include "ludo.h"

#include "board.h" // Assuming you have board logic in a separate file

#include "utils.h"

#include "play.h"

#include <algorithm>

#include <set>

#include "gameplay\_utils.h"

#include "gui\_helper.h"

int play() {

initGame();

inputnames();

bool \*visited = new bool[n\_players];

int counter = 0; // Sequential turn counter

int \*turn = new int;

srand(time(NULL)); // Seed the random number generator

// Initial setup

for (int i = 0; i < n\_players; i++) {

isfinished[i] = 0;

visited[i] = 0;

if (i < 3) {

diceroll[i] = 0;

}

}

for (int i = 0; i < n\_players; i++) {

players[i][0].team\_member = i; // Assign each player to their own team

cout << "Player " << i + 1 << " is their own team in single-player mode.\n";

}

cout << "Game Started" << endl;

pthread\_t Playthreads[MAX\_PLAYER];

while (position != n\_players) {

// Ensure every player completes their turn in sequence

for (int i = 0; i < n\_players; i++) {

\*turn = i; // Ensure players take turns sequentially

// Create a thread for each player's turn

pthread\_create(&Playthreads[i], NULL, PlayerThread, turn);

// Check if it's time to reset turn flags (after all players completed their turns)

if (counter == n\_players) {

counter = 0;

for (int j = 0; j < n\_players; j++) {

visited[j] = 0; // Reset turn visited flags

}

}

// Wait for the current player's turn to complete before moving on to the next

pthread\_join(Playthreads[i], NULL);

counter++;

}

}

return 0;

}

void makeTeams() {

cout << "Enter the number of players per team (2-4): ";

cin >> n\_players;

verifyInput();

if (n\_players < 2 || n\_players > MAX\_PLAYER) {

cout << "Invalid number of players! Must be between 2 and 4." << endl;

return;

}

cout << "Configuring tokens for each player in teams...\n";

for (int team = 0; team < MAX\_TEAM; ++team) {

cout << "Team " << (team + 1) << ":\n";

for (int player = 0; player < n\_players; ++player) {

cout << "Enter Player ID for Team " << (team + 1) << ", Player " << (player + 1) << ": ";

cin >> teams[team][player];

verifyInput();

players[teams[team][player]][player].team = team + 1;

players[teams[team][player]][player].counter = n\_tokens;

players[teams[team][player]][player].ingame = true;

}

}

cout << "Teams and tokens configured successfully!" << endl;

}

void Teamplay() {

cout << "\nStarting Team Play Mode\n";

// Initialize game for team play

initGame();

cout << "Enter the number of teams (2-4): ";

cin >> n\_teams;

while (n\_teams < 2 || n\_teams > MAX\_TEAM) {

cout << "Invalid number of teams. Enter a number between 2 and 4: ";

cin >> n\_teams;

}

n\_players = n\_teams \* 2; // Assume each team has 2 players for simplicity

string Name;

for (int i = 0; i < n\_players; i++) {

printWithDelay("Enter Player " + to\_string(i + 1) + "'s Name (Without spaces): ", 50);

cin >> Name;

players[i][0].name = Name; // Assign name to player

}

// Displaying players in teams

for (int i = 0; i < n\_teams; i++) {

cout << "\nTeam " << i + 1 << " Players:\n";

for (int j = 0; j < 2; j++) { // Assuming 2 players per team

int playerIndex = i \* 2 + j;

teams[i][j] = playerIndex; // Assign player to team

cout << "Player " << playerIndex + 1 << ": " << players[playerIndex][0].name << endl;

players[playerIndex][0].team\_member = i;

cout << " - Team: " << i + 1 << ", Name: " << players[playerIndex][0].name << "temmember" << players[playerIndex][0].team\_member << endl;

}

// Optional: Increase the pause time to 1 second

usleep(5000000); // 1 second pause

}

// Create a list for player order

vector<int> playerOrder(n\_players);

for (int i = 0; i < n\_players; i++) {

playerOrder[i] = i;

}

srand(time(NULL)); // Seed the random number generator

// Initialize game state

for (int i = 0; i < n\_players; i++) {

isfinished[i] = false;

if (i < 3) {

diceroll[i] = 0;

}

}

cout << "Team Play Game Started\n";

pthread\_t Playthreads[MAX\_PLAYER];

while (position != n\_teams) { // Game loop until all positions are filled

// Shuffle player order for this round

random\_shuffle(playerOrder.begin(), playerOrder.end());

// Each player takes their turn sequentially in this round

for (int roundPlayerIndex = 0; roundPlayerIndex < n\_players; roundPlayerIndex++) {

int turn = playerOrder[roundPlayerIndex];

// Skip if player has already finished

if (isfinished[turn]) {

continue;

}

// Create and join threads for the current player's turn

pthread\_create(&Playthreads[turn], NULL, PlayerThread, &turn);

pthread\_join(Playthreads[turn], NULL); // Wait for the player's turn to complete

// Optional: Add a small delay between turns for readability

usleep(500000); // 500 milliseconds

}

}

cout << "Team Play Mode Ended\n";

}

## Utils.cpp:

#include "ludo.h"

#include <limits>

#include <iostream>

#include "utils.h"

#include "gui\_helper.h"

void inputnames()

{

string Name;

for (int i = 0; i < n\_players; i++) {

// Typewriter-style delayed text prompt

printWithDelay("Enter Player " + to\_string(i + 1) + "'s Name (Without spaces): ", 50);

cin >> Name; // Capture player name input

players[i][0].name = Name; // Assign the name to the appropriate player object

}

}

void verifyInput()

{

if (!cin)

{

cout << "ERROR - Enter a valid number";

cin.clear();

cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

}

}

string idToColor(int id)

{

switch (id)

{

case 0:

return RED;

case 1:

return GREEN;

case 2:

return CYAN;

case 3:

return YELLOW;

default:

return "";

}

return "";

}

int getIndexByTurn(int turn)

{

return turn \* 13;

}

bool isFinalWay(int turn,int choice, int index)

{

switch (turn)

{

case 0:

if (index == 50 && players[turn][choice-1].removed >0)

return true;

return false;

case 1:

if (index == 11 && players[turn][choice-1].removed >0)

return true;

return false;

case 2:

if (index == 24 && players[turn][choice-1].removed >0)

return true;

return false;

case 3:

if (index == 37 && players[turn][choice-1].removed >0)

return true;

return false;

}

return false;

}

void moveOnFinalWay(Player players[MAX\_PLAYER][4], int turn, int choice)

{

switch (turn)

{

case 0:

players[turn][choice - 1].x += 1;

break;

case 2:

players[turn][choice - 1].x -= 1;

break;

case 1:

players[turn][choice - 1].y += 1;

break;

case 3:

players[turn][choice - 1].y -= 1;

break;

}

}

## gui\_helper.cpp:

#include "gui\_helper.h"

// Screen Clearing Function

void clearScreen() {

#ifdef \_WIN32

system("cls");

#else

system("clear");

#endif

}

// Delayed Text Printing

void printWithDelay(const std::string& text, int delay) {

for (char c : text) {

std::cout << c << std::flush;

std::this\_thread::sleep\_for(std::chrono::milliseconds(delay));

}

std::cout << std::endl;

}

// Centered Text Printing with Color

void centerText(const std::string& text, const std::string& color) {

int terminalWidth = 80;

int textLength = text.length();

int padding = (terminalWidth - textLength) / 2;

// Ensure padding is not negative

padding = std::max(0, padding);

std::cout << std::string(padding, ' ') << color << text << RESET << std::endl;

}

// Horizontal Line with Color

void horizontalLine(char character, int length, const std::string& color) {

std::cout << color << std::string(length, character) << RESET << std::endl;

}

# Individual File Pseudocode:

## Board.cpp:

### Purpose:

This file implements the display() and displayTeam() functions, which are responsible for presenting the game's results and team-related statistics in a formatted and visually engaging way. Additionally, the file includes logic for initializing the game board with initBoard().

### Pseudocode for display()

Function display:

1. Clear the screen for a clean display.

2. Print decorative headers with ASCII art for visual appeal.

3. Show game statistics:

a. Total tokens per player.

b. Total number of players.

4. Sort players by their final positions for ranking.

5. Iterate through sorted players:

a. Display the player's position (e.g., 1st, 2nd) using color coding.

b. Show detailed statistics such as tokens eliminated, hits, and game completion status.

6. List cancelled threads, if any.

7. Print a "Game Over" banner.

8. Wait for user input to restart the game.

### Pseudocode for displayTeam()

Function displayTeam:

1. Clear the screen.

2. Print a decorative team banner with ASCII art.

3. Iterate through teams to:

a. Calculate total team scores.

b. Identify the winning team.

c. Display players in the winning team.

4. Display statistics for each team:

a. Total tokens removed by the team.

b. Total hits scored by the team.

5. Show cancelled threads for players, if any.

6. Print a "Game Over" banner.

7. Wait for user input to restart the game.

### Pseudocode for initBoard()

Function initBoard:

1. Define a 2D array `newBoard` representing the game board layout.

2. Populate the board with characters representing:

a. Player positions (`r`, `g`, `b` for different teams).

b. Obstacles (`O`).

c. Home zones (`H`).

### Key Operating System (OS) Concepts in the File

#### **Multithreading:**

* + References to "cancelled threads" indicate the game's use of threads for players, showcasing synchronization and thread management.

#### **Concurrency:**

* + Display functions handle concurrent updates from game states, requiring careful data access and thread-safe design.

#### **Input/Output Operations:**

* + Uses console-based I/O (cin, cout) to interact with users, a fundamental OS concept.

#### **Memory Management:**

* + Initialization of arrays like newBoard represents static memory allocation.

#### **Process Control:**

* + The functions handle resetting or restarting the game loop, similar to process lifecycle management.

### Explanation of Code Sections

#### **Game Results Display:**

* + Displays player statistics and positions using clear formatting and ASCII art.
  + Implements ranking logic and status breakdown.

#### **Team Statistics:**

* + Aggregates team performance metrics, supporting collaborative game modes.
  + Highlights the winning team and individual contributions.

#### **Board Initialization:**

* + Prepares the game board layout, ensuring consistency in the gameplay environment.

## Display.cpp:

The display.cpp file implements the function displayCurrent responsible for rendering the Ludo game board in a visually appealing and dynamic way. It uses ANSI escape codes for color and formatting, and dynamically updates the board to display player tokens and home areas in distinct colors. The function also includes a pause to simulate a refreshing or blinking effect during gameplay.

## Pseudocode for displayCurrent

### Function: displayCurrent

1. Clear the console screen using ANSI escape codes.

2. Print a decorative border and title for the game board using:

- A sequence of equal signs for the border.

- A centered title "🎲 LUDO GAME ROUND 🎲".

3. Create a local 2D array `currentBoard` to store the current state of the board.

- Loop through each row and column of `board` and copy its contents to `currentBoard`.

4. Loop through each player and their tokens:

- Place the player's token ID (`0` to `4`) at the respective position in `currentBoard`.

5. Render the board on the console:

- Loop through each row (`i`) in `currentBoard`:

a. Print an initial spacing for alignment.

b. Loop through each column (`j`) in the row:

- Prepare a string `s` for the cell value.

- Apply specific colors and symbols based on the cell content:

i. 'r', 'g', 'b', 'y' -> Red, Green, Blue, Yellow homes (colored squares).

ii. 'O' -> Neutral zones (white diamond symbol).

iii. 'H' -> Highlight zones (orange "H").

iv. '1', '2', '3', '4' -> Player tokens:

- Loop through all players to match the token's position.

- Assign the player's team color to the token.

v. Default -> Empty cells (magenta dot).

- Print the cell with the assigned color and formatting.

6. After the board is displayed, print game status:

- Loop through all players and display their current information:

a. Assign a color to each player (Red, Green, Blue, Yellow).

b. Print the player's team number and name with the assigned color.

7. Add a one-second delay to simulate a blinking or refreshing effect:

- Use `this\_thread::sleep\_for` with a one-second duration.

### Key Features Explained

#### **Dynamic Board Update:**

* + The board is updated with token positions for each player based on their coordinates.

#### **Color-coded Visuals:**

* + Different colors for homes, tokens, and neutral zones enhance the readability and immersion.

#### **Game Status Display:**

* + Displays each player's name and color-coded information at the end of the board rendering.

#### **Refresh Effect:**

* + Adds a delay to create a smooth transition or blinking effect, mimicking live updates during gameplay.

This pseudocode captures the high-level logic of the displayCurrent function, focusing on board rendering, player token placement, and aesthetic enhancements.

## Gameplay\_utils.cpp:

### Pseudocode for gameplay\_utils.cpp

### 1. Global Variables and OS Constructs

Declare shared variables:

- sem: Semaphore for thread synchronization

- diceroll[3]: Array to store dice rolls

- players[][]: 2D array to store player and token information

- isfinished[]: Tracks if a player has finished the game

- n\_players, n\_tokens: Number of players and tokens per player

- position, winning\_position: Tracks rankings

Initialize sem with value 1 (binary semaphore for mutual exclusion)

### 2. Thread for Player Actions (PlayerThread)

Function PlayerThread(arg):

Wait on semaphore (sem\_wait)

turn <- Extract player's turn from arg

Initialize dice roll array to zeros

If current player has not finished:

Print "Player X's turn"

Repeat until dice roll != 6 or turn is over:

result <- Random dice roll (1-6)

Store result in diceroll array

If three consecutive 6s:

Print "Three 6s! Turn skipped"

Reset diceroll and terminate turn

If dice roll == 6 and all tokens of current player are finished:

Find teammate

Share dice roll with teammate

Terminate turn

Check if any move is possible for player

If no moves possible:

Increment "no-progress" counter for the player

Else:

Reset counter

For each dice roll:

Prompt player to select a token to move

Validate choice (check if move is valid and doesn't overshoot)

If valid, move token and update position

If invalid, prompt again

Post to semaphore (sem\_post)

### 3. Master Thread for Game Logic

Function Masterthread(args):

turn, choice <- Extract player's turn and choice from args

Check if player has finished all tokens:

If true, check if their teammate has also finished:

If both finished:

Declare the team as winners

Mark game as over

Exit thread

Check for inactivity:

If player hasn't moved in 20 turns:

Print "Player X inactive for too long"

Mark player as finished

Terminate their participation

Check for blocks:

If player token is blocked by opponent:

Print "Move blocked!"

Terminate turn

If single-player mode or no team wins yet:

Check for individual win conditions

If player finishes:

Assign ranking and print results

Exit thread

### 4. Helper Functions

Function shareDiceRollWithPartner(turn, partner, diceroll):

If three consecutive 6s:

Print "Partner's turn skipped"

Exit function

For each dice roll:

Check if partner can open new tokens with a 6

If yes, prompt partner to choose token to open

Else, check if any token can move

Prompt partner to choose token to move

Perform move

Function isblock(turn, choice):

For all other players:

Check if tokens of other players block the given move

If two or more blocks exist, return true

Return false

Function isanymovepossible(turn, diceroll):

If player is finished, return false

Check if any token can be moved using dice rolls

If valid move exists, return true

Else, return false

Function movetoken(turn, choice, result):

If token is at start (-1) and dice roll is 6:

Move token to the board

Else:

Move token forward by the dice result

If token reaches the final position, mark it as finished

### OS Concepts Demonstrated

#### **Threading:**

* + Multiple threads (PlayerThread and Masterthread) handle concurrent player actions and game logic.

#### **Synchronization:**

* + Semaphores (sem\_wait, sem\_post) ensure only one thread modifies shared resources like the game board or dice rolls at a time.

#### **Inter-Thread Communication:**

* + Shared variables (diceroll, players) allow threads to exchange game state information.

#### **Resource Sharing:**

* + All threads access shared data structures (e.g., player positions, dice rolls) with proper synchronization.

#### **Deadlock Prevention:**

* + Semaphores ensure no two threads wait indefinitely for each other.

#### **Timers:**

* + Functions like usleep simulate real-world delays, enhancing gameplay realism.

#### **Inactivity Handling:**

* + Logic for terminating inactive players after 20 turns prevents resource hogging.

#### **Error Handling:**

* + Validations (e.g., dice roll limits, token movements) ensure smooth gameplay and prevent undefined behaviors.

### Code Description

The provided code manages a multiplayer game where players roll dice and move tokens. It integrates advanced OS concepts for concurrency and synchronization:

* Player Actions (PlayerThread): Each player’s actions (rolling dice, selecting tokens) are handled in separate threads, allowing parallelism.
* Game Logic (Masterthread): Oversees win conditions, inactivity checks, and team coordination.
* Dice Sharing: Players can share dice rolls with teammates if they finish early, enhancing team-based gameplay.
* Block Checks: Prevents players from moving into positions blocked by opponents.
* Validation and Safety: Ensures no invalid moves, preventing undefined behaviors.

This code is highly modular, with clear separation of logic for threads, player actions, and game rules. It uses OS constructs effectively to ensure safe and fair gameplay in a concurrent environment.

## Main.cpp:

This code implements the core functionalities of a Ludo game, including game initialization, splash screen display, memory management, and game state reset. Below is a detailed breakdown:

**Modules and Libraries**

1. **Included Headers:**
   * "ludo.h", "board.h", "utils.h", "menu.h", "play.h", "gui\_helper.h": Game-specific modules that encapsulate core logic and utility functions.
   * <iostream>: For input/output operations.
   * <thread> and <chrono>: For handling time delays.
   * <iomanip>: For formatted output.
   * <vector>: For dynamic array operations.
   * <cstdlib> and <ctime>: For random number generation and time-based seed initialization.
2. **Namespaces:**
   * std is used for standard library features.

### Key Functions

1. **resetAll()**
   * **Purpose:** Resets the entire game state for a new round.
   * **Steps:**
     + Reinitializes semaphore sem.
     + Resets global variables like position and winning\_position.
     + Reinitializes board cells, player tokens, and houses.
     + Clears or reallocates dynamic arrays like isfinished and diceroll.
     + Prints a confirmation message.
2. **splashScreen()**
   * **Purpose:** Displays an introductory splash screen for the game.
   * **Features:**
     + Displays ASCII art for the game title.
     + Shows colorful banners and animated loading messages.
     + Includes a time delay to enhance the visual experience.
3. **Global Variables Initialization**
   * sem: A semaphore for managing game state synchronization.
   * cells, houses, board: Arrays representing the game board and player positions.
   * players: Stores player-specific data (e.g., positions and tokens).
   * isfinished, diceroll: Dynamically allocated arrays for game progress tracking.
   * gameOver: Boolean flag to terminate the game.
4. **main() Function**
   * **Purpose:** Entry point of the program, orchestrating game setup and gameplay.
   * **Steps:**
     + Allocates memory for isfinished and diceroll with error handling.
     + Initializes semaphore and random number generator seed.
     + Displays the splash screen.
     + Implements a game loop for menu-driven gameplay.
       - Resets the game state.
       - Calls menu() for user input and navigates to respective game modes (play, Teamplay, etc.).
       - Handles memory cleanup and game termination.

### Pseudocode Of Phase I & II

**resetAll()**

Function resetAll():

Reinitialize semaphore: sem\_destroy(sem) and sem\_init(sem)

Set position and winning\_position to 1

For each cell in board:

Reset cells[i] to default Cell()

For each player and token:

Reset players[i][j] and houses[i][j] to default

For each board position:

Set board[i][j] to ' '

If isfinished array exists:

Free its memory and reallocate

If diceroll array exists:

Free its memory and reallocate

Print "Game state reset successfully."

End Function

**splashScreen()**

Function splashScreen():

Clear the screen

Print ASCII art for Ludo and game title

Print colorful welcome banners

Define loading messages with themes

For each loading message:

Center and print the message with a delay

Print final activation message

Delay and clear the screen

End Function

### main()

Function main():

Try to allocate memory for isfinished and diceroll

If allocation fails:

Print error and exit

Initialize semaphore and seed for randomness

Call splashScreen()

While gameOver is false:

Call resetAll()

Initialize the board using initBoard()

Display menu and get user choice

If choice == 1:

Call play() and display()

Else if choice == 2:

Call Teamplay()

Else if choice == 3:

Print "Stopping the game"

Break the loop

If gameOver is true:

Break the loop

Free dynamically allocated memory

Return 0

End Function

### Flow Summary

1. Display a splash screen with animations.
2. Enter a game loop:
   * Reset the game state.
   * Show menu options and execute user-selected functionality.
3. Safely clean up resources and terminate the program.

This modular design ensures reusability, scalability, and efficient memory management for the Ludo game.

## Menu.cpp:

The pseudocode represents the structure of a Ludo game menu system that allows the user to select game settings, configure players and teams, and start or quit the game. Here's a breakdown of the code components and the OS concepts involved.

### 1. Main Function and Menu Display

START

FUNCTION main():

DISPLAY Ludo logo

SHOW Main Menu

GET user choice

IF user choice is 1 (Start Game):

CALL configurePlayers()

CALL configureTokens()

START the game

ELSE IF user choice is 2 (Make Teams):

CALL configureTeams()

START the game

ELSE IF user choice is 3 (Quit):

EXIT program

#### Explanation:

* **Menu System**: The main function handles the game logic and shows the menu, which prompts the user to choose between three options: start the game, configure teams, or quit the game. The game proceeds based on the user's choice.

### 2. Configuring Players and Tokens

FUNCTION configurePlayers():

DISPLAY "Select number of players (2-4)"

GET number of players from user

IF number of players is not between 2 and 4:

SHOW error message and prompt again

END IF

FUNCTION configureTokens():

DISPLAY "Select number of tokens (1-4)"

GET number of tokens from user

IF number of tokens is not between 1 and 4:

SHOW error message and prompt again

END IF

#### Explanation:

* **Player and Token Configuration**: These functions gather user inputs for configuring the number of players and tokens, enforcing limits for valid input. If the input is invalid, an error message is displayed, and the user is prompted to re-enter valid values.

OS Concept: Input/output handling, validation, and user interaction are key aspects of operating system functionality. In the context of this code, the user input and validation process is analogous to handling user requests in an interactive system.

### 3. Team Configuration (for Team Play)

FUNCTION configureTeams():

DISPLAY "Select team configuration"

GET number of teams from user

GET token configuration for each team

VALIDATE and prompt again if invalid

#### Explanation:

* **Team Configuration**: This function allows the user to set up teams for gameplay, including selecting how many teams there will be and how many tokens each team has. This functionality is triggered when the user selects the "Make Teams" option.

### 4. Flashing Text for Menu Options

FUNCTION flashingText(text, flashes, delay):

FOR each flash:

ALTERNATE between two colors (Yellow and Green)

SLEEP for delay duration

CLEAR screen briefly

END FOR

DISPLAY final text with one color

#### Explanation:

* **Flashing Text Effect**: The flashingText function alternates between two colors and flashes a message for a given number of times, creating a visual effect to capture the user's attention. This can be useful for notifications or alerts in the user interface.

OS Concept: This function deals with screen management (e.g., clearing and updating the console) and simulating visual effects. OS-level concepts like screen refresh rates and control of hardware output are indirectly referenced.

### 5. Center Text Display

FUNCTION centerText(text, color):

CALCULATE screen width

CENTER the text in the screen width

DISPLAY the text with specified color

#### Explanation:

* **Text Centering**: This function calculates the center position of the text relative to the console screen width and displays it. This helps in making the user interface more visually appealing by aligning text centrally.

### 6. Drawing Horizontal Lines

FUNCTION horizontalLine(character, length, color):

DISPLAY a line of 'length' using 'character' in the specified color

#### Explanation:

* **Drawing Lines**: The horizontalLine function is used to draw decorative horizontal lines in the menu using a specified character (e.g., '=', '-', etc.) and color. This helps in separating different sections of the menu for better readability.

### 7. User Input Validation

IF user choice is not valid:

SHOW error message and prompt again

#### Explanation:

* **Input Validation**: The input entered by the user is validated. If the input doesn't meet the required criteria (e.g., valid choice for number of players), the system re-prompts the user to enter a valid input.

### Operating System Concepts Highlighted

#### Input/Output Handling:

* + **User Interaction**: The game uses user input to configure game settings. Handling input/output is a core concept of operating systems, as they manage how programs interact with users.

#### Process Control:

* + **Input Validation and Looping**: The program uses loops to repeatedly ask the user for valid input until the correct value is provided, which simulates control over the flow of processes in OS-level user applications.

#### Screen Management:

* + **Text and Graphics Display**: The program clears and updates the screen regularly (e.g., using clearScreen(), flashingText()), which simulates the OS's handling of output devices like monitors.

#### Concurrency (implicitly):

* + **Sleeping and Delaying**: The game uses this\_thread::sleep\_for() to create delays in the flashing text and input prompts. This introduces an element of concurrency, where the program waits for user input or time intervals before moving to the next operation.

#### Error Handling:

* + **Input Error Handling**: The OS concept of handling invalid input is replicated in the way the program checks for valid values (e.g., checking the number of players or tokens) and displays error messages accordingly.

This pseudocode and explanation provide a high-level view of the Ludo game menu system, integrating essential operating system concepts related to user input, output, process control, and screen management.

## Play.cpp:

**Pseudocode for play.cpp:**

pseudo

Copy code

Start play function

Initialize the game

Get player names

Initialize visited array for players and turn counter

Set up random number generator

For each player (0 to n\_players-1):

Set team member

Initialize dice roll for the first three players

Create a thread array for players

While the game is not finished:

For each player:

Set the player's turn

Create a thread for the player's turn

If all players completed their turn, reset the visited array

Wait for the current player to finish before moving to the next turn

Return 0

End play function

Start makeTeams function

Ask for the number of players per team

Validate the input (should be between 2 and 4 players)

Configure tokens for each player in the teams

For each team (0 to MAX\_TEAM):

For each player in the team:

Get player ID and configure team details

Set up the player's token counter and game status

Display success message

End makeTeams function

Start Teamplay function

Initialize the game for team play

Ask for the number of teams (between 2 and 4)

Set the number of players based on the number of teams

Get player names and assign to teams

Display team members and their details

Shuffle the order of players for each round

While the game is not finished:

For each player (in the shuffled order):

Skip if the player has finished

Create and join a thread for the player's turn

Add delay between turns for readability

End Team Play mode

End Teamplay function

### Description of Code:

This C++ program defines several functions for handling a multiplayer game with team play and sequential turns. The main logic of the game is implemented in the play, makeTeams, and Teamplay functions. Here's a breakdown of the functionality:

### play function:

* + **Game Initialization**: The game is initialized, and player names are inputted. Arrays like visited, turn, and counter are used for managing players' turns and ensuring sequential play.
  + **Thread Creation**: The program uses threads (pthread\_create) to handle each player's turn concurrently. The main game loop continues until all players have finished their turns.
  + **Synchronization**: The pthread\_join function ensures that the main thread waits for each player's turn to complete before proceeding. The visited flags are reset after every round to ensure players take turns in sequence.

### makeTeams function:

* + **Team Setup**: The number of players per team is defined, and players are assigned to teams. The game allows up to four teams, and each player is assigned a unique team number.
  + **Input Validation**: The function checks that the number of players per team is within the valid range (2 to 4 players).
  + **Token Setup**: Each player gets a token counter and game status is updated (whether the player is still in the game or not).

### Teamplay function:

* + **Team Play Mode**: This function configures the game for team play, where each team is formed, and player names are inputted. The program uses a shuffled order for player turns within each round.
  + **Thread Management**: Similar to the play function, each player's turn is handled by a separate thread. The program waits for each player's turn to complete using pthread\_join.
  + **Synchronization and Delays**: To make the game more readable and to prevent players from acting too quickly, the program includes small delays between turns using usleep.

**OS Concepts Used in the Code:**

#### Multithreading (using pthread\_create and pthread\_join):

* + The code utilizes threads to allow each player to take their turn concurrently. This involves creating threads to run the PlayerThread function, which simulates a player's actions.
  + Threads enable parallel execution, allowing the game to run smoothly without waiting for each player's turn to be completed before starting the next player's turn.

#### Synchronization (using pthread\_join):

* + The pthread\_join function ensures that the main game loop waits for a player’s thread to finish before moving on to the next player. This synchronization mechanism prevents race conditions and ensures that players take turns in the correct order.

#### Memory Management (dynamic memory allocation):

* + The code uses new to dynamically allocate memory for arrays like visited and turn, which track player states. This shows an understanding of dynamic memory management in C++.

#### Randomization (using srand and rand):

* + The use of rand() for random number generation is key for game logic, such as dice rolls. srand initializes the random seed based on the current time, ensuring that the random numbers are different each time the game is played.

#### Input Validation:

* + Input validation checks are implemented in both the makeTeams and Teamplay functions, ensuring that the user enters valid values for the number of players, teams, and player IDs. This involves ensuring that input is within the defined ranges.

## Utils.cpp:

### Pseudocode

Function inputnames:

Loop through each player (0 to n\_players):

Prompt user to input Player Name with typewriter delay

Store the player's name in the corresponding player object

Function verifyInput:

If input is invalid (cin fails):

Display an error message

Clear input buffer and ignore invalid input

Function idToColor:

Switch (id):

Case 0: Return RED

Case 1: Return GREEN

Case 2: Return CYAN

Case 3: Return YELLOW

Default: Return empty string

Function getIndexByTurn:

Return turn multiplied by 13 (get player index)

Function isFinalWay:

Switch (turn):

Case 0: Return true if index is 50 and player is removed

Case 1: Return true if index is 11 and player is removed

Case 2: Return true if index is 24 and player is removed

Case 3: Return true if index is 37 and player is removed

Default: Return false

Function moveOnFinalWay:

Switch (turn):

Case 0: Move player along x-axis by 1

Case 1: Move player along y-axis by 1

Case 2: Move player along x-axis by -1

Case 3: Move player along y-axis by -1

### Description

#### inputnames():

* + This function prompts each player to enter their name. A typewriter effect is applied to the prompt to give a visual delay in text output.
  + It stores the entered name for each player object in the global players array.

#### verifyInput():

* + Checks for invalid input. If an error occurs in user input (e.g., entering a non-number when a number is expected), it displays an error message and clears the input buffer to continue with valid input.

#### idToColor(int id):

* + Returns a color name as a string based on the player's ID. The ID corresponds to a specific color: RED, GREEN, CYAN, or YELLOW. The function uses a switch statement to map the player ID to the corresponding color.

#### getIndexByTurn(int turn):

* + Calculates the index for a player based on their turn. The formula used is turn \* 13, which likely maps players to different positions in the game grid based on their turn.

#### isFinalWay(int turn, int choice, int index):

* + Determines if a player has reached a final position or specific spot on the board based on the player's turn and choice. It checks if the player's token is in the final section of the board based on their turn and whether the player has removed a token.

#### moveOnFinalWay(Player players[MAX\_PLAYER][4], int turn, int choice):

* + Moves a player's token on the final segment of the board. The direction of movement (x or y axis) depends on the player's turn (ID). For example, player 0 moves along the x-axis, while player 1 moves along the y-axis.

This structure ensures that the game's player management, validation, and movement mechanics are handled correctly based on the player's actions and turn. The isFinalWay function helps track players' progress, while the moveOnFinalWay function handles the specific movements when players reach their final destination on the board.

## gui\_helper.cpp:

### Pseudocode

Function clearScreen:

If the operating system is Windows:

Clear the screen using "cls" command

Else:

Clear the screen using "clear" command

Function printWithDelay:

Loop through each character in the text:

Print the character without flushing the buffer

Wait for the specified delay (in milliseconds)

Print a newline after finishing the text

Function centerText:

Calculate the padding needed to center the text:

terminalWidth = 80

textLength = length of the text

padding = (terminalWidth - textLength) / 2

If padding is negative, set it to 0

Print the padded text with the specified color

Function horizontalLine:

Print a horizontal line of specified length using the given character and color

### Description

#### clearScreen():

* + This function clears the terminal screen. It checks if the system is Windows or a different OS using the \_WIN32 macro.
  + If Windows, it uses the cls command; for others, it uses clear to clear the console screen.

#### printWithDelay():

* + This function prints text character by character with a delay between each character.
  + It uses std::this\_thread::sleep\_for to introduce the delay, making the text appear in a typewriter-like fashion.
  + Once all characters are printed, it outputs a newline (std::endl).

#### centerText():

* + This function centers the given text on the terminal screen.
  + It calculates the amount of padding (spaces) needed to center the text based on a predefined terminal width (80 characters).
  + The text is printed with the calculated padding and the specified color, followed by a reset to the default color (RESET).

#### horizontalLine():

* + This function prints a horizontal line using a specified character and color.
  + The line length is determined by the length argument, and the color is applied before printing. After printing the line, the color is reset.

### Use Case and Flow:

* clearScreen() is useful when you need to refresh the console view (for example, clearing the screen between game rounds).
* printWithDelay() is ideal for displaying text in a visually engaging way, like when showing a message with a typewriter effect.
* centerText() is great for formatting header text or messages in the center of the screen, such as game titles or sections.
* horizontalLine() can be used for creating visually distinct sections in the console UI, like separating sections with a line of characters.

These functions enhance the console-based user interface by providing clear screen handling, formatted text output, and visually appealing structures such as centered text and horizontal lines.

## Illustrations of Operating System Concepts in Pseudocode:

* Threading: Your gameplay mechanics involve threading, especially in functions like PlayerThread or MasterThread. The threading concept can be illustrated in pseudocode as concurrent operations, like handling multiple players’ turns in parallel.
* Semaphores for Synchronization: If you are using semaphores for synchronizing access to shared resources (e.g., the game state), you can show how semaphores control access to game variables or board states in the pseudocode.
* Process Management: If you are using processes or threads for managing different components of the game (e.g., dice rolling, player turns), illustrate how each thread or process is spawned and managed.

# System Specifications:

* Hardware/Software Requirements: You should specify the system requirements, such as the operating system, compiler (e.g., GCC), required libraries (e.g., pthread, SFML), and hardware (e.g., CPU, RAM).
* Game Rules/Logic: You can specify the rules of the game, number of players, and team configurations. For example, the game supports 4 players, each with 4 tokens, and each player has a team color.
* Game Flow: You can outline how the game progresses (turn order, dice rolling, player movement, winning condition).

# Short Paragraph on Implementation in Another Scenario:

For example, the threading and semaphore concepts in your Ludo game could be applied in a **multi-user online chat application**:

* Threading: could be used to manage each user's connection and message sending concurrently.
* Semaphores: would be useful to synchronize access to shared resources, like a message queue, ensuring that messages are sent in the correct order without race conditions.

Another Example:  
In a **banking system**, you could implement a similar thread-based system where each user (represented as a thread) interacts with the shared account (a global resource). Here, you would need to use synchronization mechanisms like semaphores or mutexes to ensure that one user’s transaction (e.g., withdrawal or deposit) doesn’t conflict with another's. Threads would be used to simulate multiple customers interacting with the system concurrently, and the database transactions would be synchronized to maintain account consistency and prevent data corruption.

This system design could be scaled further to simulate real-time banking services with additional features like account history, interest calculations, or loan management.

# Teamwork Division

This project was divided into three main components, each handled by one team member:

1. **Team Member (Zubair Ali): Board and Gameplay Logic**
   * Focused on implementing the board layout, initialization of the game state, and managing player moves.
   * Implemented functions like initBoard(), initGame(), and display(), as well as the gameplay utilities to check for valid moves and player interactions.
2. **Team Member (Zunnon Waheed): Display and User Interface**
   * Responsible for rendering the game board and tokens on the console.
   * Developed the displayCurrent() function to provide dynamic updates of the board's state and implemented visual enhancements such as token colors.
   * Worked on GUI helpers for printing with delays, centered text, and horizontal lines for improved user experience.
3. **Team Member (Abdul Haq): Main Game Logic and Synchronization**
   * Handled the core logic of the game, including managing player turns, dice rolls, and ensuring smooth gameplay using multi-threading.
   * Worked on the PlayerThread() and Masterthread() functions to handle player interactions in parallel, allowing real-time updates of the game state.

# Conclusion

The Ludo game project is a complete simulation of the traditional board game, designed to be run on the terminal/console. The codebase is structured into different modules, each focused on a specific functionality, allowing for clear separation of concerns and easier maintenance. The team worked collaboratively, with each member handling a distinct part of the project, ensuring that the game runs smoothly and provides an engaging user experience.

This game leverages multi-threading and synchronization to allow real-time interactions between multiple players. The interface is designed to be simple yet visually appealing, with dynamic updates on the game board, making use of color and layout enhancements. Through this project, **we were able to demonstrate their skills in game development, multi-threading, and user interface design**.