# In-Depth Analysis of the LUDO Game Implementation in C

Kaveen Amarasekara - 23000066

1st September 2024

## 1. Structures Used to Represent the Board and Pieces

In this C-based Ludo game, the core gameplay mechanics are integrated within several key structures defined in the `types.h` file. These structures are essential for representing the state of the game and managing the interactions between the various game elements.

### a. Enumerations

- **Color Enumeration:**
  - The `Color` enum is a simple yet effective way to distinguish between the four players in the game. It includes the values `YELLOW`, `BLUE`, `RED`, and `GREEN`. This enumeration ensures that each player and their respective pieces can be easily identified and processed within the game logic.
- **Status Enumeration:**
  - The `Status` enum represents the various states a piece can be in during the game. The possible statuses include:
    - `IN_BASE`: Indicates that the piece has not yet been moved onto the board.
    - `IN_X`: Represents the start position X state of the player.
    - `ON_PATH`: The piece is actively moving along the board's path.
    - `IN_HOME`: The piece has reached the player's home area.

    * `HOME_STR`: Represents the piece's position within the home straight towards home.
    * `ON_APPROACH`: Indicates the piece is on the approach cell.
  - By using this enumeration, the program can easily manage and update the state of each piece as the game progresses.

- **Direction Enumeration:**
  - The `Direction` enum provides the direction of movement for a piece, with values `Clockwise` and `Counter_Clockwise`. This is particularly useful in games where direction matters, such as when pieces might move differently depending on specific rules or special board conditions.
  - *Note:* Though this enum exists, logic for counterclockwise movement was not implemented.

## b. Piece Structure

The `Piece` structure is one of the main parts of the game, encapsulating all the necessary attributes for each game piece:

- **Attributes:**
  - `int id`: A unique identifier for each piece, allowing the game to track and manage individual pieces effectively.
  - `Color color`: The color associated with the piece, linking it to a specific player, included in the `Color` enum.
  - `Status status`: The current state of the piece, as described by the `Status` enum.
  - `int position`: The piece's position on the board, critical for determining movement and interactions with other pieces.
  - `Direction direction`: The direction in which the piece is moving, influencing how the piece progresses along the board as per the `Direction` enum.
  - `int capture_count`: A count of how many times the piece has captured other pieces, which could influence scoring or other gameplay elements.

This structure is designed to encapsulate all relevant aspects of a game piece, making it easy to implement complex game logic while maintaining clarity and organization in the code.

### c. Player Structure

This structure includes all necessary elements for each player:

- `Color color`: The color associated with the player, linking it to a specific player, included in the `Color` enum.
- `int num_pieces_on_board`: The number of pieces on the board at an instance.
- `int num_pieces_in_base`: The number of pieces in the base at an instance.
- `int num_pieces_in_home`: The number of pieces in the home after the round.
- `int approach_position`: The approach cell value for each player:
  - Yellow: 0+52 = 52
  - Blue: 13+52 = 65
  - Red: 26+52 = 78
  - Green: 39+52 = 91
- `Piece pieces[NUM_PIECES]`: Initializes 4 `Piece` structs within the player.

## 2. Functions Used in the Program

1. `const char* colorToString(Color color);`
   - This function returns the relevant color string according to the player's color using the `Color` enum.
2. `void initializePlayers(Player players[NUM_PLAYERS]);`
   - This function initializes 4 players and their 16 pieces' attributes.
3. `void initializeBoard(Board *board);`
   - This function initializes the mystery cell for a game round.
4. `void printBoard(Board *board);`
5. `void printPlayerStatus(Player players[NUM_PLAYERS]);`

- These two functions print the players' pieces and board (mystery cell) status after each round.

6. `void printMysteryCellStatus(Board *board);`
   - This function prints the mystery cell position.
7. `int rollDice();`
   - This function uses the `rand()` function to return a dice value between 1 and 6.
8. `void movePiece(Player *player, int pieceIndex, int diceValue, Board *board);`
   - This function includes the core logic for moving pieces after retrieving player behavior through relevant functions.
9. `void handleMysteryCell(Piece *piece, Board *board);`
   - After a piece moves, this function checks for any mystery cell activity.
10. `void redPlayerBehavior(Player *player, Board *board, int diceValue);`
11. `void greenPlayerBehavior(Player *player, Board *board, int diceValue);`
12. `void yellowPlayerBehavior(Player *player, Board *board, int diceValue);`
13. `void bluePlayerBehavior(Player *player, Board *board, int diceValue);`
    - These four functions contain the core player behaviors. They are similar for all players and call the `movePiece()` function after defining player behaviors.
14. `int determineFirstPlayer(Player players[NUM_PLAYERS]);`
    - This function ensures the first player rolls the dice throughout a game and determines the order for the rest.

## 3. Justification for the Used Structures

The use of these specific structures and enumerations is well-justified for several reasons:

### a. Ensuring Data Integrity and Readability

- **Enumerations for Fixed Categories:**
  - Using enums for `Color`, `Status`, and `Direction` ensures that only valid values are assigned to these attributes. This not only enhances the readability of the code but also prevents errors that might occur if arbitrary integers or strings were used instead.
- **Encapsulation in the `Piece` Structure:**
  - The `Piece` structure encapsulates all attributes related to a single game piece. By grouping related data together, the structure makes it easier to manage and manipulate pieces within the game logic.

### b. Simplifying Game Logic Implementation

- **Modular Design:**
  - By dividing the game logic into different functions that operate on well-defined structures, the code becomes more modular. This modularity makes it easier to understand, test, and debug individual parts of the game.
- **Clear Abstraction Layers:**
  - The game logic is clearly separated from the underlying data structures. This separation allows for easy updates or modifications to the game rules without requiring significant changes to the data structures.

## 4. Conclusion

This C-based Ludo game implementation leverages structures and enumerations to effectively represent the game state and manage game logic. By using well-defined structures like `Piece`, `Player`, and `Board`, as well as enumerations for fixed categories, the code achieves clarity, maintainability, and scalability. The modular design and clear abstraction layers further contribute to the robustness of the implementation, making it easier to extend or modify the game in the future.