University of Manchester

Cluedo Board Game Program

Anna Ivahnenko *10207948*
Object-oriented programming in C++ project report
(Dated: September 10, 2021)

The aim of the project was to create a digital version of the deduction board game Cluedo in C++. Making use of the Algorithm part of the Standard Template Library, the code effectively checks where each player can move after throwing a dice, what they can do when landing on a given cell and if the user inputs are correct. Such advanced features as unique pointers, function templates and lambda functions play a crucial role in the program's method. With minor differences like the details of the game board and visuals, the process of playing the original game is accurately recreated.

I. INTRODUCTION

Cluedo is a famous detective board game, where the players try to solve a murder mystery by deducing where, how and by whom the victim was murdered [1]. The game process is focused on moving between the rooms on the game board (which represents a mansion where the murder took place) and making suggestions about the room, the murderer and the murder weapon. There are 8 rooms, 6 suspects and 6 weapons, each represented by a clue card. At the start of the game, one card for each of the room, suspect and weapon is removed from the deck and hidden in an envelope. The rest of the clues are distributed among the players. By strategically making suggestions, each player collects clues from the others until someone has enough information to deduce the correct answer.

The project aimed to recreate the game process of Cluedo digitally, making use of such C++ features as inheritance, polymorphism, function templates, unique pointers, iterators and the algorithm library. This report explains the main aspects of the code and presents the results achieved.

II. CODE DESIGN AND IMPLEMENTATION

The main functionality that the code aimed to achieve:

- A function that prints the game board and the correct positions of 2-6 players
- An algorithm that finds the allowed moves for a given dice throw
- User input validation for making suggestions, accusations etc
- A way of checking a suggestion against the clue cards of other players
- Options to save and load game

Abstract base classes for the game components and processes, as well as derived classes for players, clue cards, player turn and the game itself, were created. The player class stores the name, piece colour, position, and the cards of each player. The card class stores all the clue cards. As the game process consists entirely of players taking turns, the player turn class stores all the main functions that are used during the game. These include the functions for moving around the board, making suggestions and accusations. The global game process class is used for making a new game, saving progress, and loading a previously saved game. An important part of the method is implemented by a function template that validates different types of user input (integer and string).

Since each player object should be created and destroyed only once, all operations with these are performed by using a vector of unique pointers. When this vector is passed to a function that should be able to modify the pointer itself, the ownership is transferred via `std::move`. These include the functions for starting, saving and loading the game, but also the function for making accusations. If a player's accusation was incorrect, they immediately lose the game, and so the corresponding pointer should be destroyed together with the object. In other cases, the vector of pointers to players is passed to functions by reference.

The code is spread across five files: two .h header files, one for each base class and its derived classes, two .cpp files for the implementation of these classes, and the main cluedo.cpp file with that contains only the main function.

Polymorphism of the `game_components` class is imple-

mented by the `get_name` function, which returns a character name for a given player object and a card name for a given card object. The `game_processes` class makes use of inheritance by giving its derived classes access to the correct answers and all the clues.

A. Game board

The game board is the only part of the game that is considerably different from the original. By analysing the game rules carefully, the complex original board was found to be redundant for this project. The core principle of the game is based on using deduction and not relying on luck to move around the board. To be able to focus on more important algorithms that make the game more interesting, it was decided to make the corridors smaller and omit the walls. The relative locations of the rooms were recreated accurately, as well as the mechanics of moving around with a throw of a (virtual) dice.

The function that prints the game board does so iteratively, by checking if there are players located on each cell. To be able to hold up to 6 players in each cell, there is a one-character-long space dedicated for each piece specifically. The code checks if a given player's position equals the coordinates of each cell and prints either the letter representing their piece or a whitespace. This way of printing the game board might not be the most visually appealing, but it was found to be suitable for the purposes of this project. A possible improvement of the game's interface would be to make use of an external library that allows to include graphics and use a picture of the original board.

B. Moving around the board

The allowed moves are found as a subset of the positions on the board where the player can move for a given dice throw. The positions are defined as a vector of tuples. First, the allowed displacements are found. The sum of displacements in the x and y directions should always equal the dice result. Then, the possible new positions are computed by applying the possible displacements to the player's current position. Finally, the positions that fall outside the board bounds are eliminated to get the final result, as shown in Figure 1. The user is then asked to choose where to move their piece and the function of the player class is called to change the position of the piece accordingly.

```
for (int i{0}; i < possible_delta_x.size(); i++)
{
    // Position of piece after move
    int new_x = x_position + possible_delta_x[i];
    int new_y = y_position + possible_delta_y[i];

    if (new_x >= 1 && new_y >= 1 && new_x <= 5 && new_y <= 5)
    {
        all_moves.push_back(std::make_tuple(new_x, new_y));
        count++;
    } else {
        continue;
    }
}
```

FIG. 1. The algorithm that looks for allowed positions on the board. It takes the possible displacements in x and y as an argument and outputs a vector of tuples that represent the coordinates on the board.

```
for(auto it = choices.begin(); it < choices.end(); it++)
    if(std::find_if(player_cards.begin(), player_cards.end(),
        [&](card* &temp){return temp->get_name() == *it;}) != player_cards.end())
    {
        matching_cards.push_back(*it);
    }
```

FIG. 2. The part of the code that iterates through the room, suspect and weapon from a chosen suggestion to look for matches in the cards of another player.

C. Main algorithms

Such crucial functions as the ones for input validation, moving, acquiring evidence, making accusations, saving and loading the game make use of the `<algorithm>` library, mainly its `find` and `find_if` functions. Together with `find_if`, lambda functions were used as a predicate. One example of this is demonstrated in Figure 2, which shows a part of the function that looks for evidence among the cards of a given player. This part of the code iterates through a vector containing the chosen suggestion (`choices`) and looks for matching cards in the player's hand (`player_cards`). The `find_if` function returns an iterator to the element that matches one of the choices if a match is found. Otherwise, it returns an iterator pointing after the last element (similarly to the `vector::end` function) [2]. The if statement is designed in a way that acts on this property: if there is a match, the vector of matching cards is appended with the corresponding card value and does nothing if there is no match.

Since the game interface is command-line based, each choice that the user makes is handled as a text input.

```
auto it = std::find(room_positions.begin(), room_positions.end(), player_current.get_position());
if(it != room_positions.end())
{
    std::string current_room = room_names[std::distance(room_positions.begin(), it)];

    std::cout<<"\nYou are in the "<<current_room<<". Do you want to make a suggestion? (enter y or n) ";
    std::string making_suggestion = validate_input<std::string>(std::vector<std::string> {"y", "Y", "n", "N"});

    if(making_suggestion == "y" || making_suggestion == "Y")
    {
        std::vector<std::string> choices = suggestion(current_room, player_current, other_players);
```

FIG. 3. A code snippet from the player turn function showing how the program checks if the player is in one of the rooms and if they want to make a suggestion.

There two main types of input: string and integer. The validation of both input types is represented by a single function template. It takes a vector of allowed inputs as a parameter and continuously asks for input until the condition is met. The check against the allowed inputs is made by using the `std::find` function similarly to `find_if` in the example above.

A substantial part of the code uses if-else statements, as there are multiple possibilities at each stage. For example, a player can choose to make a suggestion after entering one of the rooms. The code snippet shown in Figure 3 demonstrates how it is implemented. First, the code checks if the player has moved to a room. If the player's position matches the coordinates of one of the rooms, the if statement condition is met. The `validate_input` function then acquires the user's answer to the question about making a suggestion. If they agree to make one, another if statement is entered and the corresponding functions are executed.

D. Saving and loading the game

There is an option to exit the game after every turn. Upon choosing to do so, the user can save the game if they want to. The crucial game details are saved in a plain text file. These include the name of the player whose turn it was, the three answers, the sequence in which the players take their turns together with the cards that each of them has.

If a game was saved, there is an option to load it and continue playing where left off. The `load_game` function reads in the corresponding data from the .txt file. It also rearranges the vector of players so that the game can be continued by the player whose turn it was before saving the game.

```
Welcome to Cluedo.

Choose the command: NEW GAME (n), SHOW RULES (r),  EXIT (e): n

Do you want to read the rules? Enter y for yes or n for no: n
HOW MANY PLAYERS (between 3 and 6)?  4

CHOOSE the piece COLOUR for player 1 (available colours are b y r w g p) :
```

FIG. 4. The initial menu of the program and the next two outputs when choosing to start a new game. The user is asked if they want to read the rules, how many players are going to participate and what piece colours they want to have.

III. RESULTS

The program has a command-line interface. The game is played by taking turns answering questions, mainly regarding where the player wants to move, and what suspect and weapon they want to include in a suggestion.

There are two ways of starting a game: to create a new one or to load one. When starting a new game, the user is offered to choose the number of players and piece colours for each of them, as shown in Figure 4. The option to load game is not available unless there is a saved game file in the directory.

Each colour is represented by a letter on the game board and can be chosen only once. After the players choose their characters, the three correct answers are randomly selected and the rest of the cards are distributed among the players. Each player has a chance to see their clues before the game begins. The starting positions are fixed for each character, as shown in Figure 5, and the sequence in which players take turns is determined randomly. When loading a game, the positions of the pieces are initialised to where they were before.

At the start of a player's turn, a random number between 1 and 6 is generated (simulating a dice) and the player is asked to choose the cell where they want to move by entering its index number, as shown in Figure 6. The code outputs all the available moves for convenience. If

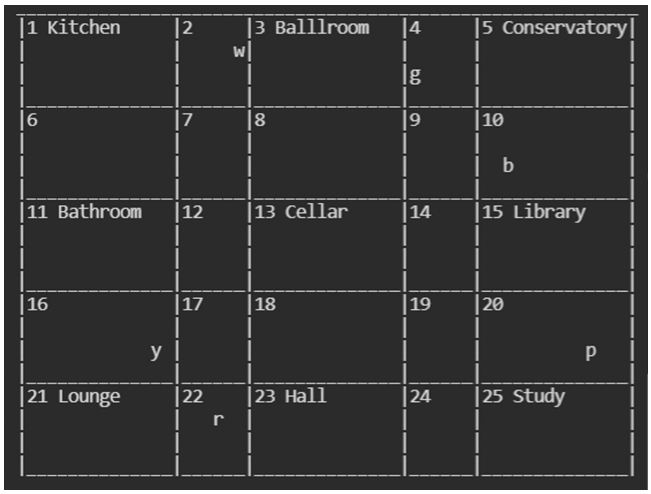

FIG. 5. Game board as outputted by the program with all six pieces at their initial positions.

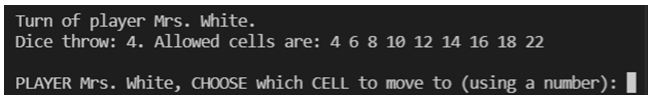

FIG. 6. Screenshot of the program output when the user playing for Mrs.White is asked to move her piece from the starting point as marked by "w" in Figure 5. The allowed positions are shown for a dice throw of 4.

the player lands on one of the rooms, they are offered to make a suggestion. Similarly, there is an option to make an accusation if they land on the Cellar cell.

When making a suggestion, the player chooses a suspect and a weapon they want to check against the clues of other players. These, together with the room where the player is located (since the suggestions can be only made about a room where the player is) are then compared to the cards that the other players hold. If a player has more than one card that matches the suggestion, they can choose which one to show. This is where the program is more effective than the board game as it eliminates the human error factor. When playing with physical cards, one may not notice one of the cards they have, therefore giving an unfair disadvantage to the player whose suggestion it was.

To make an accusation, the player inputs the room, suspect and weapon that they believe are the answers. If these are correct, this player wins the game; if not, this player is removed from the game and the other players continue taking turns as usual until someone else makes the right accusation or all players except one are eliminated.

IV. DISCUSSION AND CONCLUSIONS

Although lacking a visual interface, the program correctly replicates the core principles of the game Cluedo. Making use of such C++ specific features like inheritance and polymorphism, the game components and processes were implemented as different classes. Advanced features like unique pointers, lambda functions, STL algorithms and a function template were also used. Apart from the differences in the game board design, all components of the game are recreated close to the original. This version has some additional functionality that is not possible in the traditional version. There is an option to save the game and return to it later. The program creates a text file where all the crucial information is stored and the players only need to save their notes if they take any. Furthermore, by checking the players' cards automatically, the program reduces the chances for human error, therefore making the game more fair. As a result, the users can enjoy the full experience of playing Cluedo with the help of this program while also training their imagination.

A direct extension of the project would be to improve the interface by using an external library that can help to create a GUI. The use of the program would be easier if the players could choose their moves, suggestions and accusations by clicking on the corresponding buttons on the screen. Since the game board is not an exact copy of the original, one could add more cells and make an algorithm that allows to only enter each room through its door. Another disadvantage of this version of Cluedo is that at some points players should take turns operating the device while hiding the screen from the others. To avoid this, one could make an app to represent the cards of each player. The players would then see their clues at all times and could secretly show their phones to each other when sharing evidence.

[1] *Cluedo Instruction Book* (Hasbro, Waddingtons, Parker Brothers, Winning Moves, 2002).

[2] The c++ resources network, https://www.cplusplus.com/ (2020).