Chess Project Summary

1. Introduction

This project simulates the classical chess game, brought to life through C++ programming. The game offers complete adherence to standard chess rules, and ensures a user-friendly experience through a combination of terminal-based and graphical displays. In addition, numerous special features have been included to promote a flexible game experience. For example, players not only have the options to undo or redo their moves in the middle of the game, but can also choose to play against the computer in numerous difficulty levels, as well as to start their game on a customized board configuration.

2. Technical Overview

2.1. Overview

The game is programmed in C++, leveraging important principles and design patterns in object-oriented programming. The source code includes the following classes to represent objects of major importance: Board, Square, Piece, Player, and Move.

In particular, Piece has subclasses inherited to represent each individual piece type, and Player has subclasses to represent different levels of computer player. Details on each class will be further discussed.

2.1.1. Board (.h/.cc)

The board class is the main object in the game and stores numerous key information about the current game state. These include

- the current state of the board (represented as a 2D vector of Squares)
- the move history (used to implement the undo function)
- connection (pointers) to the text and graphics display classes
- tracker of current player’s turn

The board class is also designed to perform many major operations in the game, which include:

- initializing default or customized board configurations, and displaying them at each turn
- sending movement instructions to pieces and making corresponding changes to the board
- checking if the game is ongoing or finished after each turn

In addition, the board is the only instance of the subject class in the game, more details will be discussed.

2.1.2. Square (.h/.cc)

Each square represents a specific cell on the board grid, and is either occupied by a piece (pointer to a Piece) or empty (nullptr). The major role of the Square class is to keep track of whether the corresponding cell is currently under the threat of capture (i.e. if there are pieces that can occupy the cell on their immediate next move). This feature is important to determine and notify if a player's king is in check or checkmate at any point.

Each square is defined by

- either nullptr or a pointer to a Piece
- its coordinate on the board (not to be changed)
- a vector that, for each party (color), tracks the numbers of opposition parties that poses a threat of capture to the cell

2.1.3. Move (.h/.cc)

Every time a movie is made by any player, an instance of Move is created to store important information about the move, and is added to the move history (in the Board class) to be accessed for the undo function.

A Move doesn't perform any operations on the game state, but contains the following information which are accessible by getter methods:

- the piece that performed the move
- any piece that was captured during the move
- start and end coordinates of the moving piece
- trackers of special events that happened, such as en passant or pawn promotion

2.1.4. Piece (.h/.cc)

The Piece class represents the chess pieces, and have subclasses inherited to represent each individual piece type: Bishop, King, Knight, Pawn, Queen, and Rook. All pieces share the following attributes:

- inherent information, such as party (color) and piece type
- its current coordinates on the board
- a pointer to the board

In addition, each piece is equipped with their own movePiece() and validMove() functions that work together to

1) [movePiece()] take in movement instructions from the board (in the form of coordinates)
2) [validMove()] check for the validity of the move (based on self piece type)
3) [movePiece()] if the move is valid, change the piece's coordinates and inform the board to update the board as well.

In addition to shared attributes Pawn has a field to decide if it is still eligible for en passant, and King and Rook both have an attribute to check if they can castle.

2.1.5. Player

The Player class is used to represent the computer player, with four subclasses to model the four difficulty levels. The Player class has a single makeMove() method which returns a move to be made by the computer player upon instruction. The implementation of different computer player levels was done using the factory method design pattern, to be discussed in more details in 2.2.2.)

The behaviors of the computer player at different levels are modeled as follow:

Level 1: makes random legal moves

Level 2: if possible, makes a move that puts enemy's king in check. Otherwise, it prioritizes moves that can capture an enemy piece if there are any

Level 3: moves its piece away from any cell under the threat of capture. Otherwise, it first prioritizes putting enemy's king in check, then prioritizes moves that can capture enemy's piece

Level 4: Employs a negamax algorithm to scout all possible moves ahead for the current player, as well as optimal responses by the opponent. To optimize the speed of this algorithm, an alpha-beta prune is done, essentially ruling out certain moves that seem irrelevant. Also, the move progression is stored in a tree so recalculation is not needed. Then, once the depth of the initial search is fully reached, a quiescence search is performed, scanning for “noisy” moves (moves with captures), to ensure the horizon effect is not present. There is an evaluation function based on piece positionings to determine what constitutes a “good move”.

2.2. Design Patterns & Challenges

2.2.1. Observer Pattern

The game infrastructure adopted the observer design pattern to manage updates to the game state as well as the text and graphics displays.

Since the Board class oversees many important aspects of the ongoing, such as the board state and the move history, it is designated the only instance of Subject, equipped with the generic attach, detach, and notifyObservers methods. The observers of the board are notified every time the board state changes (after move, undo, etc.). Instances of the Observer class include:

1. Text & Graphical Display

   - Upon being notified, the text and graphical displays update themselves by resetting each cell with its corresponding state stored in the subject’s current 2D board vector.

2. Piece

   - Upon being notified, the piece checks for all squares on the board that it now poses an immediate threat of capture to, adding themselves to the array of attackers for each of these squares.
   - The notify member function is declared purely virtual in Piece and is to be overridden by individual piece types, as they move according to different rules.

2.2.2. Factory Method Pattern

The Factory Method design pattern was used in designing and implementing different difficulty levels for the computer player. As previously mentioned, behaviors of the computer player were modeled by the Player class, which is responsible for creating and returning an instance of the Move object when called.

In the factory method pattern, the makeMove member method is regarded as the factory method, and is implemented differently in each difficulty level to ensure that it follows the unique Move creation algorithm designed for each level. Such design improves customizability by allowing us to flexibly design Move creation algorithms for new levels and store them in an organized manner.

2.3. Resilience to Changes

Having used numerous principles and design patterns in object-oriented programming, our program is capable of accommodating several possible changes to be made to the game specifications, as follow:

1. Additional difficult levels

More difficult levels can be easily added to control the computer character. Due to the flexibility offered by the factory method design pattern, we can create new levels without altering the existing code. All we have to do is to create additional subclasses of Player that overrides the makeMove() factory method with its own behaviors, and make minor changes to the input options (i.e. added a possible "computer5" command to main.cc). That said, we can also conveniently redefine behaviors of existing difficulty levels by simply modifying their respective makeMove() methods.

2. Adding customized piece types

The observer pattern allows us to conveniently define and add new piece types to the game if needed. To add a new piece type, all we have to do is create a new subclass of Piece with its own version of notify() and validMove() checker. Any other mechanisms, such as its position update on the board and subsequently on the displays, are taken care of by existing code.

3. Access past moves

We have designated the Move class to remember important information about any moves that have been made, and have them stored in the main board object. This design allows us to implement any additional features that require access to past moves, such as undoing and redoing moves, as well as providing a rundown of the game history. Another flexibility that the Move class offers is the ability to accommodate changes in the formats of input commands. As piece movements are instructed by Move objects, it acts like a processing layer between raw input commands and valid movement instructions, preventing the need for us to adjust the Piece or Board classes to accept modified input commands.

2.4. Answers to Questions on Project Design

Question 1: Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. See for example https://www.chess.com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

We will first need to somehow import or create a database of openings. Once we have this ready, we can characterize each opening by a sequence of moves. We can observe that these moves create a tree: that is we start off with a "dummy node" that represents the 0th move, and this connects to every possible first move. Subsequently, each move connects to every possible next move. Each move can be represented by a string; for instance "1.e4" would represent the first move of the game being moving a pawn to e4, and "2.Nc6" would represent the second move of the game being moving a knight to c6. To build this tree, we can have a node object which stores the current move as a string, and stores a list of nodes that represent the possible next moves from this node. For example, the 0th "dummy node" would be something like {move = "", nextMoves = {{"1.a3", …}, {"1.a4", …}, {"1.b3", …}, …}}. Then, let's say "1.e4" gets selected. The next node would look something like {move = "1.e4", nextMoves = {{"2.a6", …}, {"2.a5", …} …}}. So, at each step we display the current move, and all of the possible next moves. When the user selects a next move, we replace the current node accordingly. If we wanted to implement backtracking, we could also add a "parent" field to the node so that we can go back to the previous node.

Question 2: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

An undo feature can be realized through keeping track of the move history of the board. To expand, we can think of each move as an invertible operation. We can implement this in our program by using a stack data structure; whenever any player makes a move, we push that move onto the stack. If an undo action is needed, it is as simple as popping off the top of the stack (which is the most recent move) and "inverting" the move - a piece moves from its destination back to its origin. There are certain edge cases to consider when determining how to store each of these moves:

- If a piece is captured, we should store the captured piece in the move history. Then the inverse move would reinstate the captured piece at the destination, and the moved piece at the origin.

- No extra info is needed to identify a castling move. This is since the king moves two squares along a rank on the board if and only if it was a castling move. Then, the inverse move would return the king to its origin and also return the rook that it castled with (which can be determined from the direction of the king's move).
- An en-passant move can be identified from a single boolean (true if it was an en-passant move, false otherwise). To undo an en-passant move, the moving pawn should be returned to its origin and an enemy pawn (stored as a captured piece, since this is after all still a capture move) should be reinstated at the rank behind the pawn's destination.
- A pawn promotion needs no extra information to identify - if a pawn's destination is on 8th (if white, or 1st if black) rank, we can undo this move by removing whatever piece is at the pawn's destination (noting that it had promoted), and reinstating a pawn on the origin. If the pawn captures AND promotes, we just need to combine this case and the first. In summary, a move should be stored as a class containing a pointer to the piece that moved, its origin and destination, an optional pointer to the piece that was captured (nullptr if no piece was captured), and whether the move was an en-passant move. This info uniquely identifies a move and its inverse move.

Question 3: Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

There are several changes we will have to make to accommodate the rules of a four-handed chess game. As outlined below:

- The interface. Instead of having an 8x8 chess board, we will now create a board formed by attaching another 3x8 board to each edge of the original chess board. An approach would be to use a 14x14 two dimensional array to represent the board, where all squares in the 3x3 sub-board at each corner are labeled invalid. In addition, several changes need to be subsequently made to any functions that check for move validity (e.g. validMove() in the Piece class). We not only have to check for the possibility of pieces landing in invalid squares, but also need to consider the possibility of pieces (like bishops) crossing through invalid squares to land on valid squares.
- Turn tracking. Instead of having a boolean value to keep track of which player's turn it is, we need to have an int that cycles through 0, 1, 2, 3, each of which represents a player. Corresponding changes need to be made to any color attributes in the Player and Piece classes.

3. Extra Credit Feature: Undo

The game offers players the option to undo any number of moves if applicable.

An undo action means returning to the game state before the most recent move was made. This restores the board configuration and switches the game back to the previous player's turn. The undo action can be executed multiple times in a row if needed. However, it cannot be done if a game is not ongoing, in its setup mode, or in its starting configuration where no moves have been done by either parties.

To implement the undo functionality, a stack is declared in the board object to store all instances of Move that have previously been carried out. In particular, an instance of Move is appended to the vector if and only if it was a valid move given by a player and has caused the board to update. When an "undo" command is received, the undoMove() method in Board is executed. If undo is allowed, undoMove() removes the most recent move from the move history and constructs an "opposite move" by reversing the start and end coordinates of the piece moved. This opposite move is applied to the board to reverse the last move, in addition to restoring any captured piece as well as the turn tracker.

Special considerations were given to the cases of a recent en passant move or castling. These are slightly different then regular moves and cannot exactly be "reversed". That is, in an en passant, the captured piece is not on the same square that you move to, so special care must be taken to handle that. For castling, you need to move your own rook back to its original spot as well, which is quite different from other moves.

4. Final Questions

Question 1: What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Through the project, we gained important insights into the workflow and team dynamics of software development. In particular, below are some of the important lessons we have learned:

1. Importance of the planning stage

As part of the project, we were given the task to design an UML diagram along with a brief planning document prior to implementing the game. From this experience, we learned about the critical importance of project planning, having found it particularly useful throughout the implementation stage following up. Instead of coding individual game components from scratch, we had a comprehensive blueprint of the game infrastructure ready to use, allowing us to implement each part while knowing its purpose, making adjustments to it only if needed. In addition, we learned that it was important for all members of the development team to be able to engage in this planning process, as to ensure everyone stays on the same wavelength. If, on the contrary, a member on the team is uninvolved in the planning stage, they will need to spend extra time trying to understand the code logistics mapped out completely by the other members.

2. Importance of effective task delegation

The implementation stage of the project has trained us to effectively divide tasks among team members. Throughout implementation, our team used GitHub as the primary means of collaboration, having faced issues on code conflicts on numerous occasions. To overcome the issue, we would schedule group gatherings on a regular basis to discuss task delegation, ensuring that all of us will work on different files / game components each time. This method seemed to be effective in preventing version control conflicts.

Question 2: What would you have done differently if you had the chance to start over?

While implementing the project, we made the mistake of trying to complete the entire game at once before attempting to compile and debug. When we first compiled the program, we received errors that were made significantly more difficult to debug due to the size of the program. For example, we were prompted with numerous segmentation faults without knowing their sources, which forced us to manually set and remove breakpoints at different parts of the executed code. In addition, we also encountered compatibility issues between individual game components, forcing us to modify the original plan of design on numerous occasions. If we were to start the project again, we would have started with individual classes / components, and ensured that they compile without bugs before integrating them with other components.