Chess in C++

Hyunbeen Kim, Ryan Fang

Introduction

Welcome to our project! We have developed a fully functional Chess Game in C++ using Object-Oriented Programming principles. This project is a testament to the power of OOP and its ability to create complex, interactive programs like a chess game.

Our program captures the essence of the classic game of chess, including rules such as castling, en passant, and pawn promotion. Beyond that, it even allows players to set up and play with custom boards, as well as play against chess engines of varying difficulty.

The game is designed using various OOP principles, which has made the code more manageable, scalable, and maintainable. The use of classes and objects allows us to represent the chess pieces and the chessboard in an intuitive fashion, inheritance allows us to capture the common characteristics of the pieces, and polymorphism is used to implement the different movements of each piece.

We have also paid special attention to the user interface, ensuring that it looks and feels smooth when engaging in gameplay. The game provides visual feedback for every move, and checks the validity of the moves to ensure they follow the rules of chess.

This project was a challenging yet rewarding experience. It required a thorough understanding of the game of chess, C++ programming practices, and principles of OOP, and the result is a game that we are very proud of.

Overview

The project is structured around the Model-View-Controller (MVC) pattern. This allows us to easily separate concerns and enhance the maintainability and flexibility of our code.

The graphical representation of the game is handled using the x11 library, providing a visually appealing and intuitive interface for the users.

Our Chess Game project is structured around several key components, each with its own set of dependencies. Here's a high-level overview of the structure:

- **Main**: This is the entry point of our application and the **Controller** element of the MVC pattern. It contains:
    - **Board**, where pieces are stored and important information is calculated
    - **View**, which updates the graphical display whenever necessary (moving a piece, game end, etc.)
- **View**: This component is the **View** aspect of the MVC pattern and depends on the Board class to get the current state of the game and render it to the user interface.
- **Board**: This class is central to our application. It is the **Model** aspect of the MVC pattern and depends on:
    - **History**: This component keeps track of all the moves made during a game, which facilitates the use of en passant (as it requires knowing the last move)
    - **Piece**: This class is the base class of classes such as pawn, rook, etc. It is a pure virtual class and provides information such as the colour of the piece and the number of points it provides when captured.
    - **Pawn, Rook, etc.**: These classes represent the specific types of chess pieces. They inherit from the **Piece** class and override its methods to implement their unique movements.

This structure allows us to keep our code organised and manageable. Each component has a specific role and interacts with the others in a well-defined way. This modularity also makes our code more scalable and maintainable, as we can modify or extend one component without affecting the others.

Design

Starting in the main method, we use smart pointers wrapping Board, View, and Computer objects. This indicates ownership and automatically releases the memory occupied by them when the smart pointers go out of scope (i.e. when the program ends.)

Commands

- help
    - Displays a list of commands and what each one does.
- game white-player black-player
    - Parameters *white-player* and *black-player* can be either *human* or *computer[1-3].*
    - Starts a game of chess with the respective players.
- resign
    - Concedes the game and awards the other player one point.
- move
    - Followed by two parameters for a human player. The first indicating which piece to move, and the second indicating where to move it.
    - Followed by no parameters for a computer player.
- setup
    - Enters setup mode.
- +
    - Can only be used in setup mode.
    - Followed by two parameters. The first indicating a piece to place, and the second indicating where to place it.
- -
    - Can only be used in setup mode.
    - Followed by one parameter indicating which piece to remove.
- done
    - Attempts to exit setup mode. If the board is valid, succeed in doing so. Otherwise stays in setup mode.
- quit
    - Exits the program, has the same effect as Ctrl-d

Classes and subclasses

- Board
- History
- HistoryComponent
- View
- Window

- Computer
    - Computer1
    - Computer2
    - Computer3
- Piece
    - Pawn
    - Rook
    - Knight
    - Bishop
    - Queen
    - King

Methods of Note

- Board
    - Board has a vector of vectors of unique_ptr to piece which is size of 8x8. It is a board and each piece is stored in it. The board has ownership to every piece it has.
- Move
    - Human
        - For a player's move, the input is received through cin and checked for validity. This includes checking for invalid input, verifying if a game has started, preventing the movement of the opponent's piece, ensuring there's a piece in the position, and confirming promotion specification when necessary.
        - Upon receiving a valid command, the movePiece function in the board is called. This function calls the isMoveLegal function for the piece at the given location. If the move is legal, the piece at the destination is removed (captured) and checks are made for 'en passant'. Pieces changing locations are removed from aliveWhite or aliveBlack and added back with the correct location. If the piece is a Pawn, Rook, or King, its movement is recorded by calling the setDidFirstMove function.
        - Following this, the history is updated and checks for castling are made. Finally, the changedPosition is returned and sent to the View to update the display.
    - Computer
        - The computer's moves are implemented differently across levels, utilising getLegalMoves and/or getCaptureMoves. getLegalMoves retrieves all possible moves for a piece, excluding those that would put its own team in check. getCaptureMoves returns all potential moves that would capture an opponent's piece without putting its own team in check.
            - Level 1

        - The computer retrieves the locations of alivePieces using aliveWhite or aliveBlack. It then obtains a Piece pointer for each piece and calls the getLegalMoves function to gather all possible moves. Finally, it selects a move at random.
    - Level 2
        - The computer retrieves the locations of alivePieces using aliveWhite or aliveBlack. It then obtains a Piece pointer for each piece and calls the getLegalMoves function to gather all possible moves. Next, it copies the current board and executes the move on the copied board. It then uses the getPoint function to check the board's point value. If the move captures an opponent, the point value increases by the value of the captured piece. The Level 2 computer selects the move that can capture the most valuable piece. If there are multiple moves tied for this, it prioritises capturing moves and selects randomly if none exist.
    - Level 3
        - The computer retrieves the locations of alivePieces using aliveWhite or aliveBlack. It obtains a Piece pointer for each piece and calls getLegalMoves to gather all possible moves. It then copies the current board and executes the move on the copied board. Next, it checks if the opponent can capture any of its pieces using the getCaptureMoves function. For each move, it stores the minimum possible point value. From these moves, it selects the one with the highest point value. If there are multiple moves with the same value, it prioritises checks and selects randomly otherwise.
- Castling
    - For each King's move, if the King hasn't moved before, it checks if the Rooks also haven't moved. If they haven't, it verifies that there are no pieces between the King and the Rook. If the path that the King is moving along isn't in check, castling can occur. The board registers the two-row move of the King and notifies the View.
- Capture
    - If there is an opponent in the destination, it makes the position a nullptr and removes it from aliveWhite or aliveBlack.
- Pawn
    - Move and Capture
        - A pawn can only move to an empty space. If it hasn't moved before and the destination is empty, it can advance two columns. If there's an opponent's piece diagonally in front, it can capture it.
    - Promotion

        - Each time a Pawn moves, the Board calls checkPromotion. If a promotion is possible, it requires the piece to be promoted. It then removes the old Pawn from the board and creates a new piece at the destination.
    - En Passant
        - When a Pawn moves or checks for a valid move, it examines the last HistoryComponent. If there's a Pawn at the destination and it has moved two columns and is adjacent, an 'En Passant' can occur. It first moves to the position of the captured piece. Then, the board detects the Pawn's row move and moves one column before calling the update for the View.

Resilience to Change

Change in the context of chess could come in a variety of forms. The following are some ways in which the program specification can be changed, as well as how our program would be able to accommodate these changes.

Different board sizes

Since our program makes use of constants for the width and height of the board, it would be incredibly simple to change the size of the board. A constructor could even be made so that the user would be able to play with a board of (almost) any size.

New types of pieces

All individual piece classes such as the pawn and the rook in our program extend and inherit from the Piece class. This allows for easy creation of a new category of piece. Its unique movement and/or rules, when implemented, will override the getMove() function.

Handicap system

Since human and computer players alike can vary widely in terms of skill, it would make sense to implement a handicap system, perhaps where a player of higher skill level has pieces removed in order to level the playing field and give the more experienced player a challenge.
This could be easily implemented with either the setup mode that is already implemented, or dynamically implemented by removing random pieces based on the skill disparity between players. In gaming standards, this disparity can be quantified by an ELO rating system, which might make it easier to implement this.

Q&A

Question

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.

Answer

We can make a tree of opening objects that stores the name of an opening, its moves, and other information like win/draw/lose percentage. Each node of the tree is a status of the board with the position. It starts with the initial setup and each time the user makes a move, we go to the corresponding node and display the right information. With this implementation, the program does not have to search through all the chess openings and only search for the corresponding node and.

Question

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

Answer

We can implement a history class that stores moves in the order in which they are played. Each move would be an object that stores information on where a piece was moved from, where it was moved to, which piece was moved, and which piece, if any, was taken. To undo a move, the controller object could pop the last move, move the piece back to its previous position, replace the taken piece, and switch the turn. An undo command should also be implemented.

Question

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. If it's important to your answer, state whether you're assuming free-for-all or team rules and then answer the question.

Answer

Assuming a free-for-all rule, the program would use a boolean "isWhite" to figure out the team. However, I should make it into an integer to support a four-handed chess game. It will be followed with many features that use isWhite changed. Those features are as follows:

- Turn logic changes from boolean to integer.
- Score changes so that it can support four players' score

- Moving and capturing logics are using isWhite so it should change to use the team integer.
- Computer should now use new logic because it should calculate more situations and turns. Also if a computer with level 4 or higher is using points for each piece. However, using more than 2 players' points should have different logic.

Also, I should modify the board since it gets larger in four-handed chess. Since a four-handed chess board is not square, I add logic for squares that cannot be entered. Also, starting locations of the pieces should change.
Some moving and capturing logics should also change to support new En passant and promotion logic.
Resign logic should change so that one player's resignation doesn't end the entire game unless 3 players resigned or got captured King.

Extra Credit Features

Invalid input protection

Program doesn't crash when met with improper input.

Smart memory management

Used smart pointers to indicate ownership and not have to worry about manually calling delete. Sometimes we had difficulty figuring out what object should have ownership of another.

Display pieces with images

Freeing the XImage element was very annoying: Had to use XFree() instead of XDestroyImage()... So unintuitive…

Help command / readme

A comprehensive guide on how to use the program.

Final Questions

What lessons did this project teach you about developing software in teams?

- Large projects require a division of labour. It is important to identify strengths and delegate tasks accordingly.
- It is very important to coordinate schedules with your teammates, as some parts of the code may depend on others that have not been written.

- Understanding a different perspective and keeping an open mind can help solve problems you might get stuck on by yourself.
- OOP and design patterns help many people working on the same project while maintaining consistency in the code.

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

- Spend more time in the planning phase, anticipate potential challenges and prepare accordingly.
- Manage time more effectively, leave enough time for debugging, testing, and writing documentation (there are less than 11 hours until this is due.)
- Fully read the documentation of libraries being used! Spending hours trying to figure out how to safely destroy an XImage was NOT fun.
- Think more about how to reuse the code so that it has less code and is easy to maintain or change.
- If there is a very long error, check for the error related to the standard library instead of looking through random code blocks.

Conclusion

In conclusion, our Chess Game project in C++ showcases effective use of coupling and cohesion principles, resulting in a robust, maintainable, and scalable design. Using Object-Oriented Programming (OOP), the MVC pattern, and the x11 library, we created an engaging and user-friendly game.

Coupling and Cohesion Principles:

We achieved low coupling by defining clear interfaces and using design patterns. The MVC pattern separates the game's data (Model), display (View), and logic (Controller), ensuring that changes in one part do not affect others. For example, the Board class manages the game state, while subclasses like Pawn and Bishop handle movement rules, enhancing code clarity and maintainability.

High cohesion means each module has a clear, focused purpose. The Board class manages the game state, while the View class handles the display. This separation makes the code easier to understand and maintain.

Our project's modular design, separation of concerns, and dependency management contribute to its high cohesion and low coupling. This balance is crucial for its success and sets a strong foundation for future development. By integrating these principles, we have created functional, adaptable, and maintainable software, a valuable lesson for any software development endeavor.