Overview

Describe the overall structure of your project.

Our chess project has the source code in the folder src. We have subfolders piece, player, and render, that contain headers and implementations of all descendant classes of Piece, Player, and Render, respectively, as seen in the UML diagram. All other classes have their related files in the top level of src.

Game

Manages the main game loop. Connects the InputHandler, Chess, TextRender, and GraphicRender classes. Handles tasks external to the game of chess itself.

InputHandler

Manages all user input. All user input first gets captured into this class to interpret user commands. We support all commands in the format as required in the project specification.

Chess

Handles mechanical aspects of the game of chess not related to the board itself, such as keeping track of the current player, setup mode, resigning, and winner.

Chessboard

Handles all interactions related to the chessboard in chess. By default, it is set up with the classic piece configuration. Here, we control the logic of players making moves, and complex interactions between multiple pieces, such as pawn promotion, en passant, and identifying the check, checkmate, and stalemate states of the game. Essentially, the chessboard is a 2D array, but we access indices with a Position class through public methods of this class.

Position

The Position class contains numerous constructors so we can convert information like "e8" and "(8, 0)" into a common data format for use with the Chessboard. We overload math operators on Position so we can do arithmetic on these objects.

Render

The Render class is an abstract class that represents the different renderers for the chess game. It has a pure virtual render() method that will be overridden in the text and graphics inherited classes.

TextRender

An inherited class from Render that overrides the render() method and prints the chessboard to the terminal.

GraphicalRender

An inherited class from Render that overrides the render() method and renders the board using X11 for graphics

Player

An abstract class encompassing Human and Computer classes. Pure virtual method getMove() makes this class abstract. Contains a char color, as all players must have a identifiable team, and a function to check whether this Player is human.

Human

Overloads the getMove method to get two coordinates (from, to) from cin. This was so that InputHandler could call getMove after receiving a move command without knowing whether the moving Player was Human or a Computer. Sets a isHuman flag to true upon instantiation.

Computer

Must be created with a shared pointer to a chessboard, as computer classes need to implement this information. Level1, Level2, and Level3 inherit Computer and follow the guidelines set out in the project guide. Level4 is more advanced by assigning a value to each chessboard piece, computing the value of its own pieces minus its opponents', and computing the minimum value across all board states up to 3 moves away. It takes a random move corresponding to the maximum minimum value computed, a rough approximation of the 'minimax' algorithm. We weigh moves more heavily if they put the opponent in check, or it immediately promotes our pawn. However, a check move may be ignored if the algorithm foresees it to be too costly compared to other moves.

Piece

An abstract class that has method getAllMoves, and stores its team, symbol, and whether it has moved. All chess pieces override getAllMoves to specify their unique moving behavior.

Updated UML

The updated UML chart can be found in **uml.pdf**.

Design

Describe the specific techniques you used to solve the various design challenges in the project.

Polymorphism for Pieces/Player/Render

In our chess game, we extensively utilized polymorphism within the `Piece` class hierarchy, making `Piece` an abstract parent class to facilitate a flexible and extensible design. The `Piece` class serves as the base for all chess pieces, encapsulating common attributes and behaviors while allowing specific pieces to define their unique rules and characteristics. Key fields in the `Piece` class include the private `symbol`, which uniquely identifies each piece type, the protected `team` field, denoting the team of the piece, and the private `moved` field, a boolean tracking whether the piece has moved, which is crucial for special moves and rules associated with specific pieces. Central to the `Piece` class is the pure virtual method `getAllMoves()`, which is intended to be overridden by each concrete subclass, ensuring that each piece type provides its own implementation of move calculations.

The concrete subclasses of `Piece`, such as `Pawn`, `King`, `Knight`, `Bishop`, `Rook`, and `Queen`, each inherit from `Piece` and provide their own implementation of the `getAllMoves()` method. This polymorphic design allows us to treat objects of these subclasses as instances of the `Piece` class, enabling us to write code that can operate on any piece type without needing to know its specific subclass. For instance, when calculating possible moves for a piece, calling `piece->getAllMoves()` ensures the appropriate method for the piece's type will be executed thanks to dynamic dispatch. Polymorphism helps solve the design problem by providing a clear and consistent interface for move calculation, allowing for easy extension and modification of the game. This ensures that new piece types can be added with minimal changes to existing code, maintaining a clean and maintainable codebase. Overall, this approach enhances code readability and maintainability, while also facilitating future game extensions and feature additions.

We also leveraged polymorphism within the `Player` class hierarchy to create a flexible and efficient design while avoiding code duplication. The `Player` class serves as an abstract parent class for both the `Human` and `Computer` classes, each of which implements its own specific behavior. The `Human` class interacts with standard input, taking in start and end positions from the user, while the `Computer` class calculates its moves based on the level of difficulty. Polymorphism plays a crucial role in the development of the `getMove()` method, allowing us to write code that can interact with any `Player` object without needing to know whether it is a human or computer player. This dynamic behavior enhances code maintainability and readability by ensuring the game can seamlessly handle moves from both human and computer players.

Polymorphism allows the `Computer` class to override and extend the base functionality provided by the `Player` class, increasing the complexity of the move calculation process for increasing difficulty levels. This approach not only simplifies the development process but also lays a strong foundation for future enhancements and feature additions.

On top of that, we utilized polymorphism in our `Render` class hierarchy. The `Render` class serves as an abstract parent class, featuring a pure virtual method `render()` and a virtual destructor. This allows us to create a flexible and extensible design for rendering the chessboard. Concrete subclasses such as `TextRender` and `GraphicRender` implement the `render()` method according to their specific requirements. For instance, `TextRender` provides a textual representation of the board and includes functions like `getChar()` to return characters, while `GraphicRender` uses `XWindow` to graphically represent the chessboard. Polymorphism allows us to handle different rendering methods seamlessly, enabling us to extend or modify rendering functionalities without altering the core logic. This ensures that our design remains clean, maintainable, and adaptable to future changes.

Observer pattern to render graphics

We employed the observer pattern for our graphical renderer to prevent re-renders of squares that haven't changed. Naively, one can redraw the entire chessboard after move/setup, but we immediately see that this can be extensive for the program. The observer pattern lets the chess game subscribe to our graphics class. Every time we make a change to our board, we push the positions that need to be updated graphically through a public method in our graphics class. This allows our renderer to only render positions that have changed, making it faster to display.

Maintaining single responsibility through an input manager

We decided that it would make the most sense to separate the chess logic from the user interface and input validation. We maintain the single-responsibility principle of the chess game by not allowing it to decide invalid/valid commands. To make it easier, we injected the input manager class with our different renderers (text and graphics) classes. We had to employ this due to the `setup` command; it's required to render the board after adding/removing pieces. As we mentioned before, we implement the observer pattern for graphics - so we require our chess class to also be injected into the input manager.

Forward declaration

After a couple compiles with our makefile, we noticed that including most of the classes in the header file were trivially inefficient. To make it faster we just used forward declaration in our header files and included them in their respective C++ implementations.

Using exceptions to handle run-time errors gracefully

Our group had the foresight to implement exceptions early on and continuously integrate them throughout our application. When users provide invalid input, the program shouldn't terminate, but rather gracefully request new input.

Resilience to Change

Describe how your design supports the possibility of various changes to the program specification.

Adding new pieces

As demonstrated in our extra feature, adding a Princess chess piece, our design allows us to very easily support new piece specifications. The key is to define Piece::getAllMoves()

Adding new computers

We can create new Computer classes easily to support additional difficulties. Adding a new Computer class to our set just requires us to create its corresponding header and implementation file, and then allow a shared pointer to this new class to be returned in `InputHandler::createLevel(int, char)`. This method allows for the user's input: computerX to be associated with a pointer to a certain Computer class.

Adding more players

We decided not to fix white and black as `Player` variables in our design. Instead, we created a map where the key to each player is a character representing their pieces' colour. This is defined in the variable `Chess::players`. If we were to add a third or fourth player to the game, this map would save us from having to duplicate code or make complex checks to determine the active player. Instead, the active player is always defined as the Player in the map `Chess::players` at the index `Chess::currTeam`. We have a method `Chess::switchTeam()` that could cycle through players instead of just switching between white and black.

Although we only had to account for two players, we did our best not to assume the opposite of white was black. For example, when getting the opponents' pieces, we check for pieces that do not match our `Player::color`, instead of checking for pieces that are white if we are playing as black. This means the

Changing the chessboard dimensions

We have a property `Chessboard::width` that we obtain with `Chessboard::getWidth()`, instead of hard fixing the integer 8 everywhere. This means a game mode that expands the chessboard would be less destructive to our code. In our Position class, we never assume a fixed 8 x 8 board, and since we used Position to interface with Chessboard, our logic supports accessing

Separate compilation of new files

We created a makefile that scans all subdirectories of our project with a shell command for .cc files that we can compile. Everything in the makefile is handled without specifying files manually. This means any new files created will be picked up by our make command, streamlining the development process. We could have

chosen not to consider subdirectories, but that would have come at the cost of much more disorganization in our project.

Answers to Questions (Updated)

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. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

We can create an OpeningBook class that observes the Chessboard and is notified whenever either Player calls makeMove.

The OpeningBook class contains a pre-computed tree of all the standard openings we choose to include. Each internal node of this tree represents a state of the Chessboard (not the actual chessboard), each transition between nodes is a specific move, and each path from the root to a leaf node represents a complete opening move sequence. When drawn out, this tree would look like a NFA. At each leaf node, we can choose to label the name of the opening sequence that terminates at said leaf node.

When the OpeningBook receives a notification, it transitions a node pointer to the next state based on the algebraic notation sent by the Chessboard, such as "Nf3". This allows the OpeningBook to suggest opening sequences that are still possible from the moves that have happened and suggest the precise next move in the opening sequence, by printing out the array of transitions at that node. To label the name of each opening sequence we traverse the tree to the leaf node. The possible next moves may be associated with numerous opening sequences. We can display this to the user like this:

Next move:

e4: Sicilian Defense, Lopez

d4: King's Indian Defense

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

To let the player undo just their last move, we could do something naive like copying the Chessboard after each makeMove call by the player and storing it in a variable called previousBoard in game. This would also undo the move done by their opponent without their consent.

For an unlimited number of undos, we can implement a Move class that would represent the pieces moved/captured in a move. Notice that some moves can be considered to be second-order or third-order simple moves. Consider captures or castling (you move the rook AND king for a valid castle). We can implement an AtomicMove class that would contain a stack of moves (this handles the case of

captures/castles). If we maintain a stack of AtomicMoves made by each player, we can just pop the stack and consume the Move stack we receive. If we move a piece from position x to y, we just move it from y to x. For captures we move the piece back and then replace the piece captured, etc. Maintaining a class would be the cleanest way to implement undos while adhering to SOLID principles.

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.

We will still represent four handed chess in memory as a 14x14 2D-array. However, we will need to create a piece called Wall that blocks out the 3x3 grids in each corner. Our algorithms simply check whether a spot in the array is occupied to see if its a valid position to move to. The Wall piece can serve to block the position. Since it will have no moving logic, it will not be allowed for the player to move a Wall piece.

The initialization of the Chessboard with the pieces' initial positions will need to be changed.

Rooks and pawns would need to have their moving logic changed. They will need to know whether they are moving up-down or left-right. Ideally, this can happen within the Rook and Pawn classes we've already created, with an orientation variable. However, we can also choose to create new classes HPawn, VPawn, HRook, VRook if the code isn't easily changeable. The Pawn and Rook classes would then be made abstract. The other pieces can move irrespective of orientation.

The game would also not be set to end after a single checkmate, but up to three mates. When one person stalemates, that doesn't necessarily end the game. They would be skipped in the game loop, and they would lose. Depending on the version of 4 handed chess, we may also have to remove the losing team's pieces from the board (there isn't really a standard).

The Chess class does handle any number of players with its players array. We would need to create a variable of Player that designates them as active or inactive, or remove them from the players array, so the game loop knows to skip over their input.

More complicated Computer subclasses may have to handle another dimension of strategy. It may be advantageous to play more aggressively against the weakest player to eliminate them from the game. When faced with the opportunity to capture one piece from a weaker player or a stronger player, the weaker player's piece should have more weight in the algorithm.

Extra Credit Features

What you did, why they were challenging, how you solved them—if necessary.

Princess piece

As an extra credit feature, we implemented the Princess piece, a unique addition to our chess game that falls under the category of fairy chess pieces. This piece is represented by the symbol 'f' on the board. The Princess piece combines the movement capabilities of both a bishop and a knight, giving it a versatile and powerful range of motion. While this piece is also known as an archbishop or cardinal, we opted for the symbol 'f' to denote the fairy Princess, as the letter 'p' is already designated for the pawn. The Princess's unique ability to independently force a checkmate against an enemy king, a feat that neither a bishop nor a knight can achieve alone, adds a fascinating strategic element to the game.

To implement the Princess piece, we began by creating the `Princess::getAllMoves()` method, which is designed to enumerate all possible moves of the Princess. This method first checks if the piece's position is within the bounds of the chessboard, throwing an out-of-range exception if it is not. Ensuring the piece's position is valid is crucial for the integrity of the game's mechanics. Next, we focused on calculating the Princess's possible moves. We started by implementing the moves similar to those of a bishop, allowing the Princess to move diagonally across the board. We then added the knight-like moves, enabling the Princess to jump to specific positions following the L-shaped pattern characteristic of knights.

One of the more challenging aspects of implementing the Princess piece was determining its weight. The Princess's value needed to be balanced, falling somewhere between the weight of a rook and a queen. This required thorough testing and fine-tuning. We conducted extensive tests, particularly with a level 4 computer opponent that considers the weight of all pieces when making decisions. Through this process, we developed an algorithm that accurately incorporates the unique characteristics and value of the Princess piece, ensuring that it is both powerful and balanced within the game's overall dynamics.

This implementation of the Princess piece not only functions correctly but also enhances the strategic depth of the game. Players must now consider the additional possibilities and threats posed by the Princess, adding a new layer of complexity to their strategies. The inclusion of this piece showcases our commitment to expanding and enriching the game, offering players fresh challenges and opportunities for creative play. Overall, the Princess piece is a testament to the innovation and meticulous design that we strive to bring to our chess game.

Shared pointers

We didn't want to explicitly handle our memory, so we used the <memory> library and vectors, shared_ptrs to handle our heap-allocated memory. We only use raw pointers when a class is expressing ownership.

Final Questions

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?**

We learned that developing software in teams can be challenging. Especially, when it comes to planning and splitting work between group members. However, this project also provided us with a learning opportunity on how to improve our communication skills. With the help of consistent communication, we constantly made sure that everyone is on the same page and everyone knew what the rest of the team members were working to prevent conflict and waste of time. We tried to meet up in person as often as possible and when we couldn't we would schedule a meeting together online.

Overall, this project provided us with valuable experience that will be applicable in different real-world scenarios with C++. Along the way, it helped us learn more and grow as software developers.

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

If we had the chance to start over we would definitely try to manage our time better. We would start as early as possible to figure out all the design nuances so that we can get to the implementation. We were slow moving as due date 2 approached, and many other deadlines collided with due date 3. While we are satisfied with our completed final project, we ran a couple days behind the schedule laid out in our due date 2 report. This was a fun project and not all that bad, so we could have sought to finish it more ahead of schedule.

Another productivity improvement we could have done is setting milestones for ourselves to develop a minimally viable program. We had a lot of speed bumps at the start of the project. Our coordinate system was inconsistent at the start and we spent many hours working on integration problems.