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. 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.

Openings are typically defined by the first few moves, so we can preprocess the most popular openings with a tree structure. Each edge will represent a move and the leaf nodes represent the different openings. We will pick the most common openings to display during the game.

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

We would use a stack to keep track of the moves, and rollback the moves as we want to undo them (by popping them from the stack) and updating the board accordingly. We would also need to consider the previous person who moved, the pieces that were removed from the board and the pieces that were promoted. Additionally, in the case where an end game condition such as checkmate or stalemate is reached, the program might simply exit the game. In these cases, we would have to make a way for the program to stay on the final checkmate/stalemate board until no undo is confirmed.

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. You don't need to get too specific into the rule set changes in answering the question though; your focus should be more on what would need to be altered at the high level of the design?)

Free-for-all:
We first must modify the game so that it supports 4 players instead of 2. This can be done by modifying the Game object, creating a third and fourth player field. Then, we can also change the board to be a bigger board in the shape of the four-handed chess game. Additionally we would also need to adjust the restrictions on the valid moves to ensure that the coordinates imputed are valid on the irregularly shaped board.

Two teams:
Similar to free-for-all, we would change the board to be a bigger board with new restrictions. Then for the teams, we change the players movement so that they also take into account allies. They cannot move ally pieces, but when running into an allied piece, they will treat it as if it is their own piece instead of as a piece that can be captured.

Plan

Main Class

In the main, we will be creating a Game object to start the chess game program. The main function will be used to accept the setup and start game commands.

Game Class

The game class will be responsible for the functionality of the program. It can set up a new chess game, keeps track of who's turn it is, and the overall score. It stores all the individual chess games that the player has played.

Observer Class

The observer class is made to take away the responsibility of displaying the game from the game class. There are two types, textObserver and graphicsObserver. The game object will notify these when they want to display the game.

This class is accompanied by the Subject class. Therefore, when the Subject class notifies their observers, the observers will perform the appropriate actions. In this case, the Subject class is the Game class and the Observers are the TextObservers and the GraphicsObservers.

TextObserver Class

When notified by a Game object, this object will output the current state of the board requested by the Game object to the screen.

GraphicsObserver Class

When notified by a Game object, this object will output the current state of the board requested by the Game object to the screen. This will use Xming for the graphics and the Xwindow class is created for this purpose.

Xwindow Class

This class is responsible for outputting graphics onto the screen. Therefore, the graphics observer can call on this object's functions to display what they want onto the screen.

Colours

This is an enumeration for the color of the pieces as well as the color of the checkerboard squares.

Board Class

The board class contains the information of the entire board for its chess game. It will have a history of moves in the movesMade vector. It will also have a 2D array of Square objects to keep track of the state of the current board. It can also check to see if a move is valid, and all the possible moves. When checking to see if a move is valid, it will call the Square object in that place and ask to see if it can move there. Then it will check for checks and checkmate and return the result accordingly.

Square Class

Square contains the information of a single square on the board. It contains the virtual fields of its position on the board as well as the color that the piece is. Pieces such as Rook, Queen, Knight, Bishop,

etc. inherit from Square, and the type of piece can be returned through returnType. Additionally, Square has the function verifyMove that takes in the board and confirms if the piece is allowed to move to that square. getPiece returns the Square itself which is used in the Board class.

PieceType
This is simply an enumeration that makes it more clear and easier to identify the type of piece by assigning each to a number.

Rook
Rook inherits from Square and contains the current coordinate the rook is on, the player the rook belongs to. Given a new position, it can also check to see if it can travel there.

Knight
Knight inherits from Square and contains the current coordinate the knight is on and the player the knight belongs to. Given a new position, it can also check to see if it can travel there.

Bishop
Bishop inherits from Square and contains the current coordinate the pawn is on and the player the bishop belongs to.. Given a new position, it can also check to see if it can travel there.

Pawn
Pawn inherits from Square and contains the current coordinate the knight is on and the player the pawn belongs to. Given a new position, it can check to see if it can travel there. The pawn also has an extra method called promote that takes in the PieceType of the piece the pawn wants to promote to.

King
King inherits from Square and contains the current coordinate the king is on and the player the king belongs to. The verifyMove method in King is different from the rest in that it also needs to check that castles don't pass through attacked squares.

Queen
Queen inherits from Square and contains the current coordinate the queen is on and the player the queen belongs to.

EmptySquare
EmptySquare is a square with no piece on it. It will return false for any call to verifyMove().

Player
The player class is either of type Human or Computer. It keeps track of the player types that are currently playing for the Game class.

Computer
Computer inherits from Player, and overrides its own virtual getMove function that takes in the Board.

Level 1,2,3,4
The levels inherit from Computer, and override the getMove function. Each generates a move based on the board. For level 1, a random move from the list of possible moves is used. For level 2 and 3, moves are searched through for ones that are preferential for each algorithm. For level 4, we are implementing

the minimax algorithm, which also requires the list of all possible moves taken from the possibleMoves function from Board.

Human

For this type, it represents that a human is playing on this side of the board. For each getMove() call, it will expect console input to receive a move from the user.

Additional Features

There are a few additional features we will work to add.

First, we will try to add a book of standard moves to the game which can be visible if the user would like. This can be implemented through a tree-like structure as described in the question.

We will also add the functions to undo moves. This can be implemented as described above.

Schedule

Friday, July 19th, 2024

- [x] ~~finish the UML + plan || collaborative~~
- [x] ~~create the github for the project || collaborative~~
- [x] ~~create all the class files as outlined in UML (.cc and .h files)~~
    - [x] ~~square + its children || Aaron~~
    - [x] ~~player + its children + BoardIterator || Mincy~~
    - [x] ~~board + game + objects required for observers || Lin Xin~~

Saturday, July 20th, 2024 (all console)

- [x] ~~write all the constructors + deconstructors~~
    - [x] ~~square + its children || Aaron~~
    - [x] ~~player + its children + BoardIterator || Mincy~~
    - [x] ~~board + game + objects required for observers || Lin Xin~~
- [x] ~~figure out how to use enum || Mincy~~
- [x] ~~take in input for the setupGame() function || Aaron~~
- [ ] create the methods used in the setupGame() function || Lin Xin
- [ ] finish setupGame() in Game class || collaborative
- [x] ~~read up on minimax algorithm for level 4 + update group members on it || Aaron~~

Sunday, July 21st, 2024 (all console)

- [ ] take in input + parse it for playGame() || Lin Xin
- [ ] fill in the checkMove() function for the pieces
    - [x] ~~queen, rook || Aaron~~
    - [x] ~~king, bishop, empty || Lin Xin~~
    - [x] ~~knight, pawn || Mincy~~
- [x] ~~complete the functions in Board || Mincy~~
- [x] ~~complete the observer functions + textObserver (not graphics) || Mincy~~
- [ ] figure out how to put the pieces on the screen || Lin Xin
- [x] ~~start writing minimax algorithm || Aaron~~

Monday, July 22nd, 2024 (all console)

- [x] ~~implement computer level 1 || Lin Xin~~
- [x] ~~implement computer level 2 || Mincy~~
- [ ] implement computer level 3 || Lin Xin
- [x] ~~write the main function || Mincy~~
- [ ] finish minimax algorithm for level 4 || Aaron

Tuesday, July 23rd, 2024 (graphics)

- [ ] tie the loose ends || Mincy + Aaron
- [ ] complete the graphicsObserver || Lin Xin

Wednesday, July 24th, 2024

- [ ] book of standard openings || Aaron + Lin Xin
- [x] ~~undo last move(s) + swap turns || Mincy~~
- [ ] start the design document || collaborative

Thursday, July 25th, 2024

- [ ] finish the design document || collaborative
- [ ] prep for the demo || collaborative
- [ ] margin time || everyone

Friday, July 26th, 2024

- [ ] ideally nothing