Overview

Our project design is split into three modules, following the MVC(Model-View-Controller) architecture. Our classes are organized into the modules as follows:
Model (holds the state of the game)

- Board
- Square
- Piece
- Pawn, Knight, Bishop, Rook, Queen, King

View (visual representation of the state of the game)

- Observer
- TextDisplay
- GraphicsDisplay
- XWindow

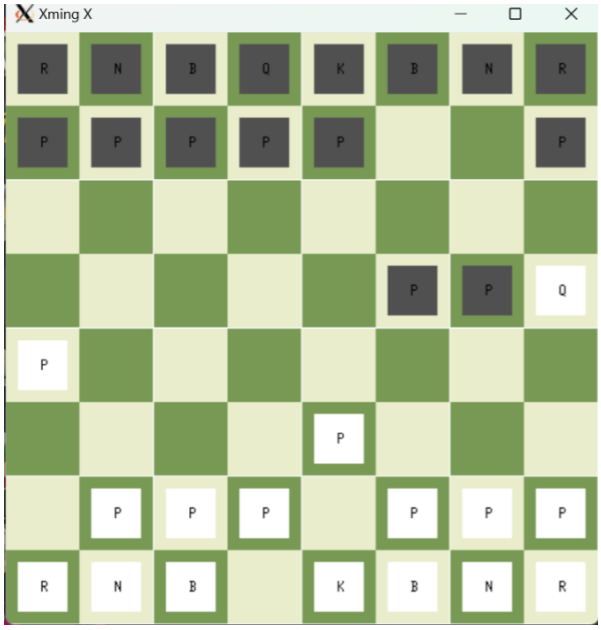
Controller (manipulates the state and manages the gameflow)

- GameController
- Player
- Human, ComputerOne, ComputerTwo, ComputerThree, ComputerFour

We also have various additional files:

- Enum classes: Colour, PieceType, MoveType
- Structs: Move, MoveInfo
- Utility: botutil

We have two visual representations: a text-based display using just characters and a graphic display with colours inspired by chess.com.



3 move checkmate

Design

To begin, we needed a general design for the modules of our program. We knew we wanted to use the observer design pattern to create a visual display that updates in real time. We also knew we wanted a module for storing the current state of the board. We realized that the MVC architecture seemed to fit our vision and used that as a model for our design. That gave us the final piece: the controller that manipulates the state of the game.

A problem we encountered early into our development was how we were going to represent the board. Our initial plan was for the board to be a 2-dimensional vector of pieces. However, that would mean we would need to have “empty” pieces. Something about that implementation didn’t sit right with us. We considered storing the pieces of each player in vectors or in maps(where the key is its position, like “e7”). However, we decided that a 2-dimensional vector made more sense to work with and came up with an alternative solution to using “empty” pieces: a Square class. Instead of pieces, our board would be a 2-dimensional vector of squares that could contain a piece. Furthermore, the squares would have observers attached, allowing them to notify the visual display whenever a square’s piece changes. This way, the GraphicsDisplay/TextDisplay can respond to board changes like piece moves, pawn promotions, and captures, and alter the presentation of the Board accordingly in real time.

One issue that came with designing a chess game were all the various pieces on the board, and how each piece had unique behavior. In order to store all these various Pieces on the board, and move them, inheritance along with polymorphism was used. The Square class has a pointer to a general superclass called Piece. Piece holds all the general information about a Piece(i.e color, it’s position, etc.), and the individual Piece classes(Knight, Queen, King, Pawn, Bishop, Rook) inherit from this superclass Piece. These individual Piece classes override the general Piece::getValidMoves(BoardInterface &b) method, and provide their specific valid moves instead. This way, a piece that is a knight has different valid moves than a piece that is a bishop or a piece that is a queen while still allowing them to be stored together using polymorphism.

The largest issue we came across during our development process was circular dependencies. It seemed to threaten our entire program design. Since the Piece class relied on the Board class to produce valid moves, the Board class relied on the Square class to create the board, and the Square class relied back on the Piece class to store a piece, a circular dependency was created in the very center of our program. Forward declarations alone were not sufficient because too many classes needed to call their methods which are not available through a forward declaration. Our solution, therefore, was to use an interface class: BoardInterface. BoardInterface is a pure virtual class that provides the methods that getValidMoves needs in order to know if a specific move is valid(i.e whether or not there’s a piece blocking the move, whether or not the piece is pinned, etc). Board is then a concrete class that provides the implementations of all the methods of BoardInterface.

In order to minimize coupling, we made sure our modules communicated purely through function calls. To illustrate simply, when a turn is played, the gameflow module gets a move

either from user input or generates one using a computer. It then calls the move function in the game state module which decides whether or not it is a valid move and changes the state of the game. Sensing a change, the notify function is called, notifying the observer module to change the visual display. Furthermore, we implemented interface classes to minimize direct dependencies. The BoardInterface class, for instance, makes it so that the controller module is unable to access the implementation of Board and can only call a strict set of methods defined in the BoardInterface class. This allows for minimal coupling and, at the same time, facilitates our design which would not function otherwise due to circular dependencies.

To maximize cohesion in our design, we started with modules that serve only one purpose and built each class purely to serve that purpose. The MVC architecture helped us a lot in laying out the foundation of our design. Once we knew what each module was supposed to accomplish, their elements naturally became highly cohesive. Thus, all the classes in each module are used in conjunction with each other to serve one purpose. For instance, Board, Square, Piece, and the subclasses of Piece work together to serve one purpose and nothing else. It does not manipulate the state, get user input, nor create output. It merely represents the state of the board while providing methods for another module to do the manipulating, to get the user input, and to generate output based on the state of the game.

Resilience To Change

Our design is quite resilient to change, due to the fact that each class follows the one responsibility principle. Each of the individual piece classes provide information on how that specific piece can move. The Text/Graphic display classes are only responsible for displaying the current state of the board. The Board is only responsible for managing the various squares/pieces on the board. It does not communicate with the user at all, nor does it output anything. The GameController is only responsible for inducing change on the board(whether it'd be through various moves, setup, etc.). The Player classes(Human/Computer) are only responsible for generating a move given a board(whether it be through standard input, or some algorithm). By adhering to the one responsibility principle closely and also by making classes communicate with each other via only functions(by encapsulating the implementation of each class), any change in the implementation/responsibility of one class will only affect that class and that class only. Other classes will remain the same, and so we mitigate the risk of one change having a ripple effect and breaking the whole program as a result.

As well, the usage of polymorphism and inheritance in our various piece classes made the program easily extensible. To illustrate, if a Pawn could suddenly move three squares at a time, Board/Square/Piece would all remain untouched. Only the Pawn::getValidMoves() function would need to be changed. By having all pieces inherit from the Piece class, various methods like the accessor and mutator methods along with getMove(int nr, int nc) (which returns whether or not a move on a piece is valid) can be reused for each piece. In addition, if a new type of piece were to be added we'd only simply need to create that new piece class, and none of the other pieces(knight, pawn, king, etc.) would be affected whatsoever.

The employment of various design patterns also made our codebase more resilient to change. Specifically, one such example is the observer pattern we used to represent the Board. By implementing the observer pattern, we make the data and its representation loosely coupled, so that changing the behavior of one of these classes would not affect the other. For example, if the square colors in the graphics needed to be changed, the Square class would be untouched. The Square class has no say in its representation(its job is only to hold the data). Only the GraphicsDisplay(whose job is to manage the graphics) would need to be altered.

Additionally, a specific example of planning for extensibility is our validMoves function. Rather than determining if a singular move is valid, we produce all valid moves of a piece. This way, in the future, we could implement the ability to show all possible moves on the board when the user selects a piece. Unfortunately, we did not get far enough to implement that, so we can only include it in the resilience to change section instead of extra credit features.

Our program recently exemplified its resilience to change when we added the ability to undo a move. The implementation, which was further explained below, needed the implementation of various methods like Board::undo(). Despite the various edge cases, ultimately the only two classes that were changed were Board and GameController. None of the other classes were changed(except GameController, which needed a small change to accept a new undo input from the user).

Answers to Questions

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.

This could be implemented by first adding support for representing a Board via a string, e.g the FEN notation(https://www.chess.com/terms/fen-chess). Note that a method would be created which would take in a Board, and return the corresponding Board as a string(in FEN notation). Then we would create multiple maps. One map where the key would be the board state(as a string), and the value would be a vector of Move structs, representing the accepted moves that could be played from the current position. Other maps would also have the key be the board state(in string form) and map to an integer representing the win/draw/loss percentages.

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?

Similar to question 1, we could add support for representing a Board as a string. We would then create a method which would take in a string, and return a Board. We would then have a stack(which could be implemented using a vector, and the pop_back()/emplace_back() functions)) of strings(called history) which is maintained during the gameplay, where each string in the stack represents the previous states of the Board also in FEN representation. Before every move, we add the current board(in FEN) to the stack. Note that a stack of Boards could also be used, but a string is more compact(as the Board contains Squares and Pieces which all contain extra integers/enum classes as fields). As well, copying a whole Board after every move would be rather expensive. Then when a player wants to undo, the top element in history is popped and the board in GameController is set to this popped element(the string should be converted back to a Board). We can do this as many times as we want, so long as the stack isn't empty, and so unlimited undos would also be supported. Note this implementation could be used in conjunction with our design, but due to optimization issues with the Computer Level 5 move generation we opted for an approach with no board copying(the method used is listed in the enhancements section).

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.

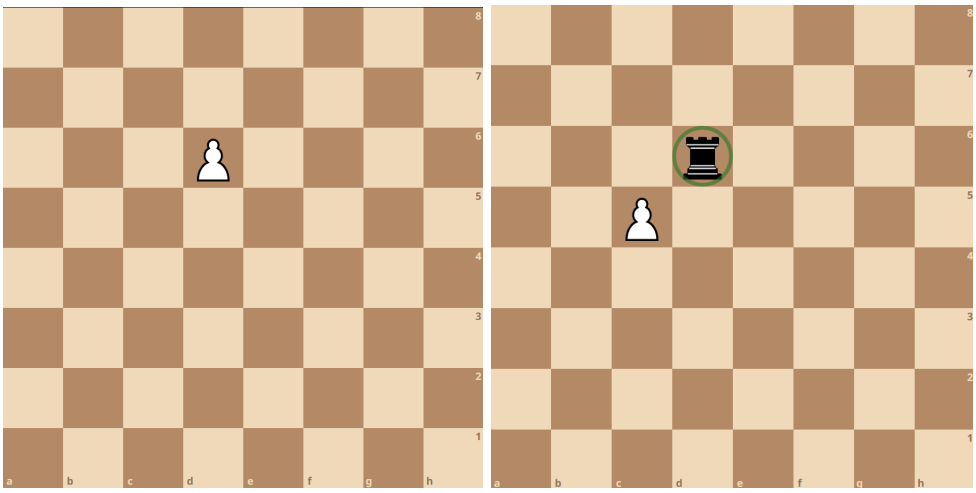
In four-handed chess(assuming FFA, last person standing), the design of the board is very different. Instead of an 8x8 grid we now have a 14x14, and additionally each corner has a 2x2 empty area. We can support the empty area by representing these "null" squares as null pointers on the Board. We can then represent the empty squares as Square classes with a nullptr for a Piece. Then since there are now 4 players, we need 4 colors, which must be updated accordingly in the Colour enumeration class. Similarly, the GameController must also have 4 players, instead of just 2, and the main runGame() method must cycle through 4 players instead of just 2. We would then need to also change it so that checkmates don't end the game(as there are 3 other players!). We would then also need to loop through all players to see if they are checkmated(as two people can get checkmated at the same time).

Extra Credit Features

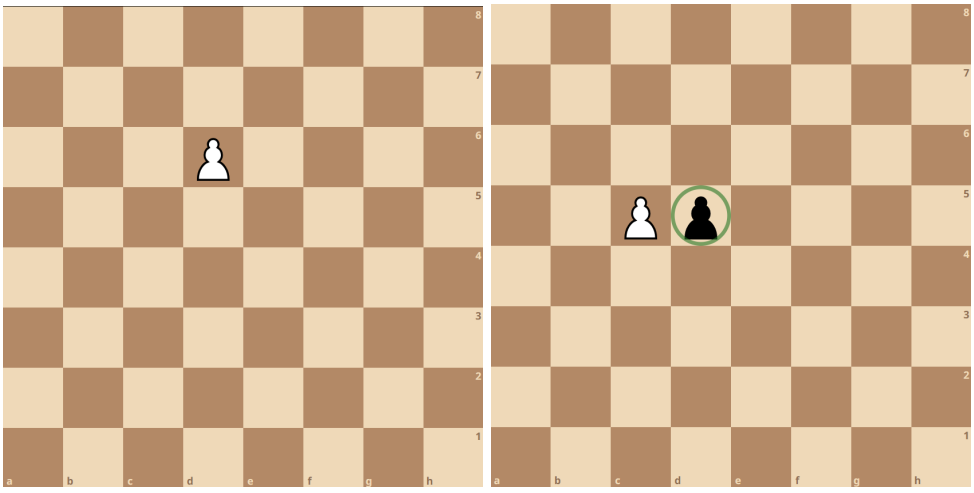
One of the extra features we were able to implement was handling memory management with the use of unique pointers. In doing so, we were able to follow the RAII idiom,

Another extra feature we were able to add was the undo command. In addition to the "move" command during gameplay, the "undo" command would reverse the previous move, making sure to re-create and capture pieces and also flip the turns, i.e undoing the previous turn. Our undo command can be called multiple times and the program undos as required, until the default (or custom entered) board setup is reached. The way we implemented this feature is by maintaining a vector of MoveInfo structs called moveHistory. The MoveInfo struct is simply the Move struct, wrapped with extra data needed to restore the captured piece(e.g the type of

captured piece(if there was a captured piece)). Essentially, the MoveInfo struct provides information about the move, and the state the board was in before the move(whether or not a specific player was in check, etc.). Every time a valid move is made on the Board, a Move and PieceType are wrapped together and pushed to moveHistory. Upon calling “undo” in the GameController, the Board will pop its moveHistory vectors to determine which squares and pieces to rearrange, and if it is necessary to create a new piece to undo a capture. Initially, the undo function was simply implemented by taking the piece at the ending coordinates of a move, returning it to the starting coordinates, and potentially adding a piece to the ending coordinates to reverse a capture. This naive approach worked for most basic moves but failed when considering castling, en passant and promotion. For castling, the position of the king reverses as expected but the rook needs to be accounted for additionally. For en passant, undoing the capture does not place the recovered piece in the new coordinates, instead, it must return an enemy piece in the square behind its ending location.



undoing a regular pawn capture, black piece takes original white pawn square



undoing en passant, square for recovered black piece is different

Undoing a promotion required the extra step of turning a promoted piece (Knight, Rook, Bishop, Queen), back into a pawn into its previous position on the second last row before the promotion. Having to account for these different cases made implementing the undo functionality quite challenging.

Note that the approach we listed in Question 2 was not utilized, as it relied on copying boards and our Computer Level 5 also utilized the undo function. The Computer Level 4 utilized the minimax algorithm(with depth 5) along with alpha-beta pruning in order to traverse all possible board configurations 5 moves ahead. Many different board configurations would need to be made, as a result of this algorithm. Initially, we produced these different board configurations by making a copy of the board, making the move on this board copy, and recursing on this new board copy. At depth 5, billions of copy operations could be called. With all these copy operations, the Level 5 Computer was taking around ~30-40 seconds per move in the mid-game. In order to remedy this, we made only one copy of the board(to search through and to not disrupt any of the observers), and after traversing the tree of each move, we'd undo the move. This way, we could perform all the searching on one board, with only one copy operation. This optimization alone brought our move computation time down to ~5-7 seconds. Finally, in order to maximize the amount of pruning done, we greedily looked through moves in an order based on their move type(i.e search through the trees of checks first, then captures, then other moves, etc). We also searched through the board from the center out(i.e in layers). This way, moves in the center(which are arguably more important usually) are also prioritized. This optimization finally brought down the move computation time down to ~2-3 seconds. Note that sorting the validMoves in a board position was also considered(i.e sorting captures by capturedPieceValue - movedPieceValue), but the extra logarithmic factor ended up making the moves slower.

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?

We learned that developing software in a team requires very careful planning and a ton of communication. By consistently discussing as a team about which new features to add, which bugs to fix, and how to efficiently divide up the work so no overlaps are made, we were able to maximize progress every single day. Throughout our program's development, we deviated quite a bit from our original plan. Thus, group meetings and discussions were crucial to stay on the same page. Since we made so many changes so often, it was difficult to rely on our UML and code each class independently so, often, each of us had to change multiple classes at once while implementing any new functionality. Thus, carefully planning each person's work for the day was very helpful in minimizing overlaps and merge conflicts. Although working as a group has its challenges, having teammates that provide fresh perspectives is extremely helpful. A bug that one person has been stuck on for hours can often be solved by another person

instantly. As a group, we were able to bounce ideas off of each other resulting in much more innovative design and problem-solving processes.

Question 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 put a greater focus on the initial planning. We believed that the plan was simple and did not think hard enough about the details or the potential problems we could run into. For example, we did not consider the circular dependencies which took more than an entire day to solve later on. Although most of our module and class structure survived, a lot of the methods were completely different by the end and many new ones we did not consider were added. For instance, we decided to make an enum class MoveType where the type of each move would be determined by the piece rather than the board. This changed a lot of functions and took much more time than if we had just thought of that at the start. If we could start over, we would spend just a bit more time at the beginning to properly and more thoroughly plan out the details. That would save us a great deal of time and energy later on.