Chess Design

Ryan Schindler

Introduction:

I decided to work on this project alone. In the end, I managed to pull off a functional Chess game. This is an accomplishment, as I ended up struggling a bit more than expected. Luckily, I was able to get everything working, with the sole exception being the en poissant and castling features.

Overview:

First, the main function of my program is used for reading commands. More specifically, the user can choose to input the game or setup command. No matter which you choose, a game object with an empty board is initialized beforehand. The setup command is almost entirely implemented within the main function, calling methods of this game object to insert or remove pieces. When the game command is called, the board will be setup up if it is currently empty, otherwise, a game will be started using the pre-existing board setup.

The board class contains most of the information required to run the game. It contains a vector of Squares that contain pointers to all the current pieces. Pieces are not attached to player objects, but rather moved using methods from the board class and specifying a color. This choice was made so that setup mode would be able to add pieces without any players initialized. The board class is therefore able to determine whether certain moves are legal, a given player is in check, all squares are empty, where an opponent's king is located, as well as generate all available moves of a given color.

The Piece class has different subclasses for every piece, allowing for unique move generation methods. To determine where other pieces are, Piece objects also contain a pointer to the board. Pieces can also determine whether they have placed the opponent in check and return the character (used in the text display) of their given piece.

The game class contains pointers to two Player classes. These may be any combination of the 5 subclasses of Human and Computer1 to Computer 4. Each Player subclass overrides the move method. For example, a Human player would accept input from cin while a Computer Player would systematically determine a move to make based on its level. The players also contain a pointer to the board, allowing them to access methods which determine the position of different pieces and complete their moves.

Finally, the Board class contains pointers to a text and graphics display. The text display contains a vector of char vectors, which can be updated with access to a given square object. This is almost identical to the Text Display implementation from A4 question 1. The Graphics Display is also like A4 with a few adjustments. The window class now has a method to draw pieces, which functions by loading bitmap files.

Design:

One of the trickiest aspects of the project was determining whether a move is valid. Ultimately, I decided that the most effective way to do this would be to generate all valid moves before a player's turn. By taking this approach, it is easy to simultaneously determine whether a player has won the game (their opponent has no moves available). Generating valid moves are implemented within a 2-step process. First, all potential moves are generated within a piece's specified move function. Next, moves that would place the players own king in check (not allowed), are removed from the list. To determine whether a move will place a player in check, the move is completed (without updating the display) and the opponents individual pieces check whether they can kill the king. This method is not very efficient, but I determined it would be the most straightforward as trying to write a more advanced algorithm could be prone to error.

Another tricky aspect of the project was how to utilize the graphics display. Initially, I thought it would be easiest to output each piece as a letter, but this turned out to be very challenging in X11. The default font was much too small to see, I was clueless on how to add new fonts, and worried any font I used would not work during the demo. Instead, I found loading bitmaps of each piece to be much simpler. The only issue was that I needed four bitmaps of each piece, which would be used depending on both the color of the piece as well as the square behind it. Since loading the bitmaps could be slow, I had to ensure that both only one square was loaded at a time, and that temporary moves (for calculating checks) were not shown.

In terms of cohesion, my program is admittedly on the low side in some areas. Some classes, such as my Piece and Player classes, are very focused and have relatively high cohesion. However, classes such as my Game, and especially Board class have way too much going on. Currently, my Board class is responsible in some way for handling nearly everything that happens in the game, which is not ideal.

As for coupling, I would consider my program about average. In general, most classes are not very independent of each other at all. The exception here is once again the board class. I ended up giving nearly every class in my program a pointer to interact with the board. This ended up working when I was in a pinch, but I doubt was the ideal way to code my solution.

Resilience to Change:

My program is very resilient in some ways and not at all in others. One of the biggest flaws of my game is that it only works with 2 players, one black and one white. Nearly all my functions utilize these color inputs to work, and therefore it would be very hard to add any additional players. I chose to take this approach as it made determining legality of moves much simpler, as well as knowing when to promote pawns, detect check, and other things.

However, it would be relatively easy to add new types of pieces and higher levels of computers to my game. This is true as you would only really need to create new classes with their given specifications. In general, I believe my program is fairly resilient as long as the basic rules of chess stay the same.

Answers to Questions:

Chess programs usually come with a book of standard opening move sequences, which lists accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. See for example C 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.

To do this, I would implement a class called OpeningMove. This class would contain 2 coordinates (struct), one for the piece to move and the other for the destination. It would also contain a float for its percentage of effectiveness.

Since each move requires a specific board setup (I am assuming exact), I would need a way to know whether a move is currently available for use. To achieve this, I would use an observer design pattern, allowing OpeningMove objects to be attached to any amount of other OpeningMove objects. I would then attach all moves that can come next (for the opposing team of the current move) to the current object. For example, if an OpeningMove moved a white piece, all the moves that black could do next would be attached.

Next, I would implement a class called Book. The Book class would contain a vector of OpeningMove objects, each of them moves white can do using the initial board setup. It would also contain a pointer to a Board object, with all pieces initially set up in their default positions, as well as the text and graphics displays.

Finally, I would implement the book command in the main function, allowing players to use the opening book. This would start out as a normal game, displaying the default board setup. However, it would also output a variety of move options, ordered from highest to lowest percentage of effectiveness.

Instead of typing in a move to make, you would input a number based on some provided options. Like Chess.com, the player would choose options for both black and white pieces, alternating each turn. The first options would be those in the initial vector within the Book class. For the second move and onwards, the options given would be the observers of the previously chosen move. The game would proceed as normal, and the user would be given the option to exit or restart when no more moves can be located.

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

To allow a player to undo their last move, I would need to save the previous position of whatever piece they moved, as well as any piece they may have captured. To achieve this, it would make sense to create a new Move class. This class could contain two sets of coordinates (a struct containing integers for the row and column), as well as a value to indicate the previous piece. As I intend to delete each piece as it is captured, I would also include an enumerator that could hold a value of none, pawn, king, queen, rook, bishop or knight to represent any piece that was captured.

To allow for an unlimited number of undos, I would save the history of moves as a vector in the Board class. Whenever a move is made, a move object would be created and pushed to the end of this vector.

A method within the board class called Undo() would then be able to reverse the last move that was made. It would look at the last Move within the vector and locate the piece to be moved based on the saved coordinates. It would then move that piece back to its old coordinates using the standard method. This should not cause any complications, as the square it is moved back to should always be empty. If the move indicates a piece was captured, a new (opposite color) piece would be created at the specified location. This would be like using the setup command to add a piece.

Finally, the pop_back() method would be applied to the vector. Since we went back a move, the turn would also need to be changed back to the previous color, and the legal moves recalculated for each piece.

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.

It would be relatively difficult to convert my program into a four-handed chess game, as the current structure of my program relies on only having two colors. First, I would have to add 2 more Players to my Game class, and assuming each player would have their own set of pieces, 2 more possible colors to be applied to each team and piece. I would then need to adjust all of my board functions to stop checking for black and white colors. For example, I current use if statements that assume any color other than black it white, which of course would need to be adjusted.

With four players in a game, the NextTurn() method would then be adjusted to loop through the vector of players, starting back at the beginning if the last player is reached. The legal moves for each Piece of a player's color would also be calculated after the player before them has made their move. In terms of check and checkmate, all Pieces of a different color would be considered as if they were on the same team.

Pieces would mainly behave the same way as in standard chess, with the ability to capture any color that is different to yours. The one exception to this rule would be pawns. This is the case as my current plan is to give pawns the ability to move either up or down depending on their color and transform after reaching the end of the board. In four handed chess, I assume it would be necessary to account for 4 directions, and so I would need to make some minor adjustments.

Finally, it would be necessary to have a larger board size. Currently, I plan to make the board a fixed size, but this would have to be replaced with a variable. This grid size would determine the size of the vector in Board, and likely need to be passed to other objects to ensure they do not try to access any out of bounds indexes.

Extra Credit Features:

Since I worked alone, I ended up with no time to complete extra credit features. The one feature that could be considered extra credit is that my graphics display shows images for each type of piece. This was done by creating bitmaps in Photoshop. I had to make 24 in total, as I could not get an alpha channel to work and needed to account for not just the color of the piece, but also the tile behind it. Additionally, I implemented a function in the window class (used from A4), that was able to output these bitmaps.

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?

I worked alone on this assignment. The most valuable lesson I learned throughout this project is the importance of planning out the structure of your program ahead of time. At first, I thought chess would be easy to implement, but ran into some difficulties with move calculations as I worked on the project. Luckily, with the way my classes were set up, it was easy for me to add new functions to account for things I may not have prepared for. Had I not planned out my program beforehand, it is likely my solution would have been too much of a mess to make any future alterations.

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

If I had the chance to start over, I would alter the way moves are made on the board. Instead of comparing the color and looping through pieces on the board, I would use an observer design pattern to attach pieces to the player object. I believe this would be much more efficient and allow for the game to run faster overall. It would also make my program more resilient, as currently my program entirely relies on having only two different colors on the board.