Plan of Attack

It will be easier to progress through the project if we start with a “shell” first. This will include all of the different files and classes created, with most/all fields and methods declared, but not yet implemented. This will also make it clear to determine what still needs to be completed.

We will be starting the coding part of the project on Wednesday, July 15. We want to finish early so the schedule will be fairly aggressive. We hope to have the shell completed on Wednesday. This will include all of the listed classes in the UML as well as the fields, which are not listed.

After that we can fully implement the tile class since it is the simplest. Next, we can begin working on the piece classes. We will have the theory of how the game will run so we don’t need to implement that just yet.

For the piece classes, we should be able to fully implement these. move() will be a fairly complicated function, as it will have to check to make sure moves are valid, send updates to the game, and re-print the display. This might be split up into other functions later on. It will simply take in a string and move the piece’s coordinates to that string’s location (ex e5 = [4,3]. castle() will be called by the player. This will check to see if castling is valid, then game::setposition will be called to change the positions of the rook and king, instead of the move function which could create complications.

Player could be implemented next. The functions are fairly simple so this shouldn’t take too long. Resign() would call game->endgame(this) to end the game. movePiece would be similar, where it calls game’s movePiece.

Player and Piece could be worked on simultaneously; Hayden works on one while Nick works on the other. We hope to get this far by the end of Friday.

Next would be the implementation of the game class. There could be an option in a new game to use text or graphics, and the corresponding function for output would be used for the rest of the function’s execution time. endGame would end the current game and add one for the player’s score. Is check would be called every turn, and isCheck would call isCheckMate if isCheck is true. PrintFinalScore would be used for the text-based version, and this would print the scores, then call the proper destructors to ensure there are no memory leaks. We hope to finish this by Saturday or Sunday.

After the game class is completed, we will work on making sure all rules are completed (pawns reaching the end, *en passant*, etc.). This should only take a few hours. The computer would be implemented next, then the graphics. We hope to finish by Sunday or Monday.

Questions

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.

Answer: You could create a class called *moveSet* that contains data for a specific move. If you only wanted the book of standard openings to display what to do, it could simply have a function that outputs something like "move pawn forward 2" or "move bishop forward left". If you wanted to actually have *moveSet* perform an action on the board, it could hold a pointer to a piece (ex pointer to queen) and the information for the move (move number, current position, opponent's move), and it could call piece->move.

This could be held in a 2d array. So the first row would be for the first move, and would hold different options (ex arr[0][0] would be move left pawn up 2 spaces, arr[0][1] could be move knight forward left, etc). Second row would hold the 2nd play, etc

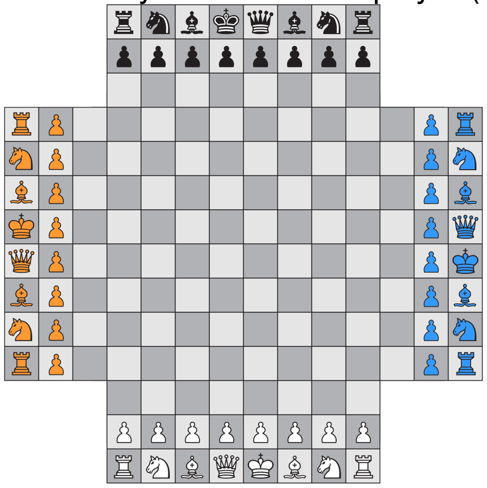
Assuming the player using the book of moves knew which each did and when to use them, it could be held in a tree to lookup whatever move they typed in (ex "firstMovePawnE2").

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

Answer: You could use a stack for the player's move. If it's only undoing their last move, it could be called before their turn ends. Each piece could have a vector of its move history, and the board can have a vector of pieces moved. So every time a player moves a piece, that piece is added to the board's lastMoved stack and where the piece moved is stored in the piece's lastMove stack. To undo a move (or unlimited), you could simply say "undo" and the board would run a function undo() to undo the last move, essentially popping the piece off of the stack of last moved pieces.

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.

Answer: First, you would need to create 2 more players. The board size would have to be increased, you would have an extra 3 rows or columns to each side respectively. Example below. There could be an option for the opposite sides to form teams to create 2 teams of 2, or an option for a free-for-all. You would also have to make sure turns taken are rotated and not back and forth. Pawns could reach the left and right ends as well as the far end to change pieces. The game would only end when one player (or team) is left.



Question: Assuming you have already implemented all the commands specified in the Command Interpreter. Is there a way to implement the "Load a pre-saved game" feature, with maximal code reuse. Explain.

Answer: You could create a function for new games called "setup". As specified in "Command line options", loading a pre-saved came consists of 9 lines, 8 that contain the board's information. This could be loaded in a data structure/vector/array. To load the game or start a new game, a function called *setup* would be called. *setup* will have default parameters *in the function(ex. setup(x=0,y=1,z=2))* that are specifications for a new game. If you wanted to load a new game, you can just call setup with the default values. If you wanted to load a pre-saved game, you can enter in the data to override the default parameters to create a new game with your specifications.