Chess Final Report

1.0 Introduction

For this CS246 final project, we as a group decided to create chess using object-oriented design principles learned in lectures.

2.0 Overview

Our program loosely follows the MVC model. Starting with the model, this is primarily done through the board, cells, chess pieces (including subclasses), and the computer (including subclasses) classes.

The Board class includes all the functions needed to interact with the board, such as adding and removing chess pieces from the board, moving and capturing chess pieces, determining check/checkmate/stalemate statuses, calculating legal moves for all pieces, and more.

The Cell class stores coordinates, color, and any chess pieces that reside on a particular square of the chess board.

The Move class stores a potential move, that is it stores the starting cell and the destination cell of a potential move of any piece on the board.

The ChessPiece class, is a super class for each individual chess piece (i.e. Pawn, Rook, Bishop, Knight, Queen, King). Each subclass allows for us to determine the logic required to initiate a move/capture, determine if an attack is possible, as well as determine all possible legal moves for each individual chess piece.

There is also a Computer superclass, which essentially allows us to determine what level the computer will play as when running a game between a human and a computer, or a computer versus computer game. Consequently, there are 4 subclasses of Computer (level1, level2, level3, level4), which returns a valid move that most closely resembles the specifications outlined.

Finally, we have included a Controller class that allows us to control how the game is played and in what mode, what the level of the computer(s) will be, as well as set up a custom game to test and validate different strategies. This Controller class essentially allows a user to play chess, and also helps to display the chess board to the player.

3.0 Design

To start off, we created a Cell class, which essentially represents a square on a chess board. This class would store the coordinates of the Cell, the color of the cell (Black, White), as well as a pointer to a ChessPiece which currently resides on this cell. A pointer to a ChessPiece was included because chess pieces constantly move around the board, and as such we did not want a lot of copying of chess pieces occurring during the game.

The Cell class had various getter functions (i.e. returning x,y coordinates, returning ChessPiece pointer), as well as setter functions that set the coordinates of the cell, and the color of the cell. Functions to add a chess piece, remove a chess piece, as well as delete a chess piece were also included. There was also a friend operator<< overload which printed a chess piece, or a blank space / underscore depending on the color of the cell. This facilitated in printing out the board display.

Moving on to the Board class, this was the class where most of the interactions with the game can be done through. The actual Board was set up as a vector of vectors of cells, called theBoard, with both vector objects being size 8 (a constant integer variable). A board could be set to an empty board by looping through theBoard and using the setter functions for each Cell, set the coordinates of each cell as well as its cell color. A default starting board can also be created by initializing the appropriate black and white chess piece on its corresponding starting cell. Getter methods that returned a Cell reference were written, in addition to methods that placed ChessPiece pointers on a cell, as well as removing chess pieces from a cell.

One design decision that was made to slightly improve efficiency was to have 2 separate vectors of cell pointers, with one vector storing cell pointers that currently have a black chess piece, and the other vector storing cell pointers with a white chess piece. Essentially, these 2 vectors are subsets of theBoard, and help simplify some calculations that require only black or white chess pieces. After every move is conducted, these vectors are updated to always contain cells with black or white chess pieces only. They should never contain empty cells.

Another major design decision that was made was to store all possible valid legal moves in a vector of Moves. To do this, a separate Move class was created and it simply stored the start and destination cell of a valid move for an existing chess piece. After every move in a game, this vector would be updated with a new set of possible valid moves. The advantage of this Move class would be best utilized in checking for checkmates, stalemates, and determining a move for the various computer levels. This,

along with other methods of the Board class will be talked about more in detail after the discussion on the ChessPiece class.

The ChessPiece class is a superclass for all the types of chess pieces that exist on a chess board. There are 6 inherited classes for 6 of the different types of chess pieces (Pawn, Rook, Knight, Bishop, Queen, Knight). Along with some getter methods to access fields such as the color, the type of piece, and character display. The ChessPiece class is an abstract class that has 3 pure virtual methods: movePiece which is in charge of moving and/or capturing a piece, canAttack which determines whether a piece can attack and capture another piece, and determineLegalMoves which determines all possible legal moves for a piece based on the state of the chess board.

Without being too redundant for each piece, the canMove method works by determining what the possible cells are that a certain chess piece can move to, and then determining if the destination cell is a possible cell the chess piece can move to. It then determines if there is another chess piece located at the destination cell or another chess piece is “blocking” the chess piece in its path. The method also checks to make sure you are not moving to a cell that is occupied by a chess piece of the same color. Finally after going through these series of checks (and in some other cases, additional checks like when castling), any chess piece residing on the destination cell is deleted, and the starting chess piece’s pointer is attached to the destination cell.

The canAttack method is similar to the canMove method, however it only checks whether a chess piece can attack and capture a chess piece of the opposite color given the starting and destination cell, and returns true or false. It does not move the chess piece, and it does not capture any chess pieces.

The determineLegalMoves function takes in a starting cell where a chess piece resides as well as a board, and determines every single possible legal move for the chess piece on the starting cell. The logic to determine a valid destination for a cell is similar to that of canMove, however this function takes into consideration every possible valid destination cell, and using the board’s copy constructor, creates a new board and simulates the move. If the resulting move does not result in the king of the current color being checked, the move is added to the vector of allBlackLegalMoves or allWhiteLegalMoves, depending on the color of the chess piece.

Special moves like a pawn moving forward 2 squares, en passant, and castling are also implemented by checking for the necessary conditions that are required to execute these moves in the above 3 functions. In particular, the number of moves a chess piece

has moved, and whether a chess piece moved 2 squares last move, are checked to initially determine whether such a move is possible in the first place.

Moving back to the Board class, the activateMove method works to execute a move/capture of a piece on the chess board. This method is essentially a wrapper method for the canMove method of the ChessPiece class. Similarly, the attackPossible function is also essentially a wrapper method of the canAttack method.

The checked method of the Board class works by going through all the chess pieces of the opposite color of the king being checked, and determines if any of those chess pieces can attack the king. If that is the case, the function returns true, otherwise false.

The checkMated method works by calculating all

- Subset of board, having 2 vectors storing cells with Black pieces, and cells with White pieces
- Implementing a vector of all possible legal moves to simplify implementing computer levels.
    - Each subclass of Chess Piece has an overridden function to determine all legal moves for the specific piece.
- Overridden move function for each subclass of Chess Piece to actually move the chess piece to a new cell, as well as capture a piece if possible.
- Overridden canAttack function for each subclass of Chess Piece to determine if a piece can attack another chess piece, helping to determine checks and checkmates.
- Copy ctor for checkmates, move simulations, and implementing computer levels
- Inheritance for chess pieces, and computer levels

- Level 1 had utilized the rand() function with srand(unsigned) and time(NULL), allowing the random number generator to truly be random.
- Level 2, we prioritized finding a capturing move over a checking move as that seemed more appropriate, followed by finding a checking move (performing a simulation of each move, determining if that results in a check on the simulated board), and if none could be found, returning a random move.

- Level 3: Harsimran talk about this more
- Level 4: Harsimran talk about this more

- Controller Info: Hirav talk about this more

4.0 Resilience To Change

5.0 Answers to Questions

6.0 Extra Credit Features
Hirav talk about this

7.0 Final Questions

8.0 Conclusion