CS246 Final Project Design Document (Fall 2022)

Sankeeth Ganeswaran Marc Balgobin Jeffrey Xuzhang

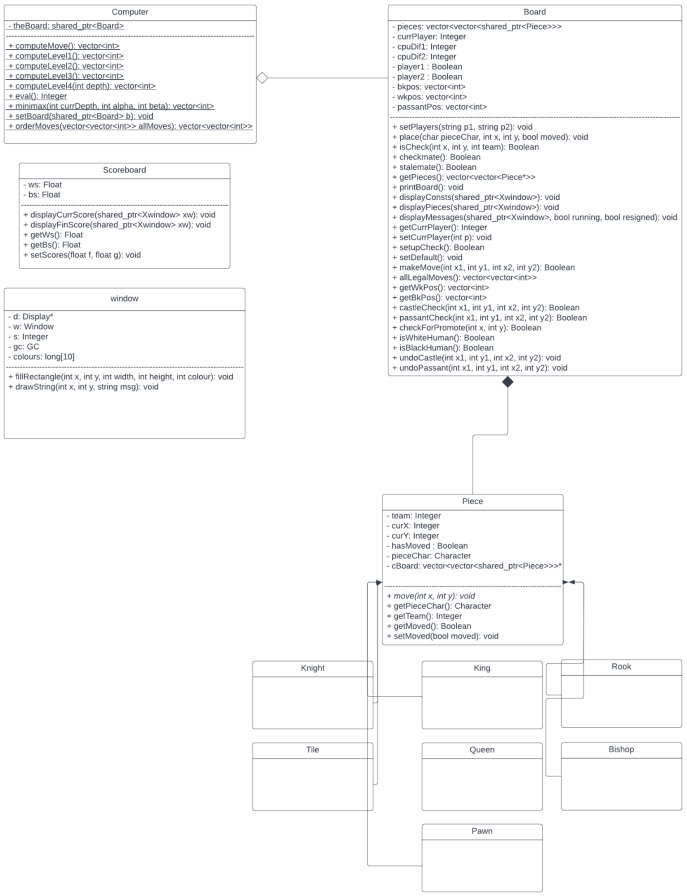
Introduction

We decided to make chess for our project as we felt that it would give us an effective way of using a plethora of the object-oriented concepts that we had learned throughout this course in order to design a well-defined game with established rules. Chess is a staple in coding practices, and thus we wanted to test our knowledge on the classic Chess game.

Overview

The overall structure of our project consists of a main class where the input and output takes place, a Board class representing the chessboard, an abstract Piece class representing any chess piece with concrete derived classes for each different type of piece, a Computer class representing the computer player, and a window class for the graphic display.

UML



Design

The Board

The first design task was to figure out how we wanted to represent the chessboard in our program. This is an important design decision, as the way we implement this would ultimately have an effect on almost every other design aspect of the program as we continue to develop the project. We decided to create a class for a Board, which would have fields relating to the state of a game and methods for checking game conditions and rules. We decided that this was a sufficient design choice as each game only utilizes one board, and only a board can only have one game being played on it, so this 1:1 mapping means that it makes sense to be checking overall game state and conditions in the board class, without making some sort of separate Game class. The Board class would have a field that is a two dimensional array, consisting of 8 arrays, each with 8 elements in them to represent the 64 square chess board, where each element would represent a piece on a space on the board. This would allow us to easily check the positions of pieces and the state of any square on the board, using 0-indexed x and y coordinates. We implemented this two dimensional array using vectors, despite the fact that a dynamically sized array wasn't necessary, as the board wouldn't be changing dimensions in any way. This was purely for the sake of memory management.

The Pieces

The next decision was to decide how we wanted to represent different pieces on the board. We realized that this was a perfect opportunity to use the concept of inheritance and polymorphism, as we could classify each different type of piece (King, Queen, Bishop, Knight, Rook, Pawn) to derive from an abstract Piece class, as they all share similar attributes and characteristics with respect to the game (being able to move on the board, having an x and y coordinate position, being affiliated with either black or white, etc.), while also having some varied functionalities, in the way they move, capture, and more. As such, we implemented a Piece class to represent exactly that; containing attributes and functions shared by every piece, as well as a virtual move function to represent the different ways that each piece can move. This makes the Piece class abstract, and so we implemented a derived class for each different type of piece that overrides the move function implementation to represent its own unique way of movement. We also made each Piece object have a character pieceChar field, which is one of the characters 'k', 'q', 'b', 'r', 'n', 'p', each corresponding to a different type of Piece, as this would give us an easy and simple way to access an object through a pointer to the base Piece class, and know which what type of piece we are accessing.

The Board and the Pieces

The Board class's two dimensional vector would now consist of pointers of type Piece, which could refer to any of the derived objects King, Queen, Bishop, Knight, Rook and Pawn. This naturally posed the question of how we could then represent an empty space on the board, a square with no piece on it. Rather than simply let it be equal to a nullptr or something equivalent, we decided to make a seventh derived class of Piece, a Tile class that simply represented an empty spot on the board. We decided that this would be a lot easier to work

with down the road, in the case that we ever needed to treat an empty square as if it housed a piece, as we suspected that a nullptr would cause some issues or at the very best awkwardly designed code. At this point each derived class still possessed the same fields and parameters in their constructors, so we wouldn't have to worry about any problems with mixed or partial assignments. The only difference between each sibling class was with the implementation of the overridden move function. It is also important to note that each Piece had a field that was a pointer to the same 2D vector that the Board class has a pointer too, but NOT the Board object itself, as would cause a cyclic reference that we did not want (or need). The reason for this is justified later.

Setup

With this established class design for the board and the pieces, we were able to start implementing the setup command. We created a place function in Board, which takes in a character representing a piece, as well as 2 integers representing an x and y coordinate on the board, and places the corresponding piece by creating an object of the corresponding derived class of Piece and setting the 2D vector's element at that index to a pointer to that piece. The affiliation of the piece is determined by the case of the character, in accordance to the specifications. We then created a function which had the sole purpose of parsing a position such as e2 into the two appropriate 0-indexed coordinates to be passed into the place function. We were then able to call this place function whenever the "+" operation was called during setup to place a piece. The "-" operation was as simple as placing the empty Tile piece at the given position. The "=" just involved inverting the currPlayer field in Board, which was an integer that was 1 when it was white's turn, and -1 when black's.

Printing and command loop

We then created a printBoard() function which just utilized a nested for loop to iterate through the two dimensional vector in order to print out the state of the board, using the pieceChar and team field of each piece to determine the representation of each piece on the board. Whether or not an empty tile should be displayed as black or white was done through a check of if the parity of the x and y coordinates were the same or not. The setup command loop took place in main.cc, within a larger command loop where inputs like "game" were checked for. Inputting "done" would break out of this inner loop, effectively ending setup. In the outer command loop, when "game" is inputted, a boolean gameRunning would be set to true. When this boolean was true, inputting "move" and coordinates would execute a move for a human player.

Move command

We needed to make sure that we planned out our move implementation carefully, to avoid issues and headaches down the road. We decided that the way we wanted to design it was so that each piece's move function would take in coordinates representing the desired destination coordinate for the piece, and using the fields representing its current x and y coordinates, and the 2D vector representing the chessboard, return a boolean representing whether or not the move was valid. At this point, we were only concerned with giving each type of piece the ability

to move in accordance to the way it's defined in chess, without concern of checks or other rules. This means that the move function would ensure things such as a rook moving orthogonally or a bishop only moving diagonally, along with others. For example, it would make sure that the destination coordinate for a rook shared one of 1 of the current x and y coordinates of the rook. This means for instance if the rook was at coordinates (1,2), going to (1,5) or (5,2) are valid moves but (2,1) or (3,4) are not. Using different arithmetic checks like this allowed us to implement this move function for each piece. Using the 2D vector field, the move function also made sure to check that pieces couldn't move through other pieces (except the knight), by checking if the pieceChar of a piece at a position on the board was 't', which represents a Tile, an empty spot on the board. The affiliation of any piece at a position was checked as appropriate, through the team field for each Piece. This was done to implement things such as a pawn being able to move diagonally only if it's making a capture, or in other words, there was any enemy piece at the destination coordinate. Once these move functions were implemented for each Piece, we created a makeMove function in Board, which actually executes the move by taking in the original and destination coordinates, calling the move function for the piece at the original coordinates with the arguments being the destination coordinates to verify if the move is valid, and it is, calling the place function with the appropriate arguments. The move command for a human player would then call the makeMove function for the Board, and then continue the command loop. It's worth making it very clear that "moving" a piece is placing it at a destination coordinate and placing an empty tile at the original coordinate, and that a capture at this point is simply moving a piece to a destination where there already exists an enemy piece, replacing the enemy piece with the moving piece.

Graphical Display

Now that we had a working move command together with the Piece and Board class, it was a good time to get started on the Graphical User Interface. First, an Xwindow class was created using X11 libraries. In the Xwindow class, colours were set up for the board tiles and background. In order to draw on an Xwindow, the methods fillRectangle and drawString were implemented. To reduce how much of the screen is redrawn while playing a game, we split the responsibility among three methods in the Board class: displayConsts, displayPieces and displayMessages. displayConsts displays the background, row numbering and column lettering. displayPieces displays the tiles and the pieces on the board. displayMessages displays all required messages that can occur during a game such as "White is in check." and "Black's Turn." Next, to display the current and final scores a scoreboard class was created. In the scoreboard class, the method setCurrScores was used to access the private score fields and update them whenever a game ended. The methods displayCurrScore and displayFinScore were used to display the scores before end of file is reached and after respectively.

Check

The next task was to implement the concept of check into the game. We decided that the way we could do this was to create an isCheck function which took in integers representing the coordinates and affiliation of a king, and checked if any enemy piece on the board was

"attacking" that position, which essentially means checking if calling their move function with the king's coordinates as the destination would return true. The Board class would keep track of the positions of both kings throughout the game as fields, so that we could detect for a check at any point without having to search through the board to find out the king's position every time. We then called the isCheck function in the makeMove function as a part of verifying if a move is valid. This was done by keeping track of the current position, making the move by calling the place function, calling isCheck, and if it returns true, placing the moved piece back (and if a piece was captured, replacing it). This means that a move would not be valid, if it would leave the king in check as a result.

Checkmate and Stalemate

Checkmate and stalemate both happen when the current player has no legal moves left to play, with checkmate having the additional constraint that they also be in check. As a result we created a function in Board called allLegalMoves that will return a list of all the legal moves that the current player can make, in the form of a vector of 2 vectors (with an additional char element if the move was a promotion), representing the original and destination coordinate of the move This would involve looping through every piece on the board, and then calling the move and isCheck function to check the other positions on the board they could make a valid move to. If this function returns an empty list, and the current player is also in check, it is checkmate and they have lost. If they have no moves but are not in check, it is a stalemate and the game is a draw. We created the functions isCheckmate and isStalemate to check for these conditions, and they're called after every move is made, and when returning true end the current game by setting gameRunning = false in the main command loop, and incrementing the "bwin" and "wwin" variables representing the black and white players' points appropriately.

Castling, En Passant and Promotion

With the basic moves and game conditions down, it was time to implement these additional moves in chess. We decided to implement them as separate functions in the Board class, which were to be called as part of the move verification in the makeMove function as we figured that they would require information about the state of the Board that may not be accessible to the individual pieces, we didn't want to break encapsulation, and moves like castling involves multiple pieces so it's a bit unintuitive to only include it in either the King class or Rook class. For en passant, we realized that we could keep track of whether or not the last move made by the opponent was a double advancement in the Board class, in order to check if the en passant move was possible. After implementing these functions, which simply took in the original and destination coordinates of a move, and returns a bool representing whether or not it's a valid execution of the special move, we realized that we should also create undo functions for these moves, in order to do things such as make sure they don't lead to the king being in check. If a promotion move was deemed valid, the command loop would then prompt for the character representing the piece to promote the pawn to, in accordance with the specifications.

Computer player

The computer player was implemented through the static class Computer, and it simply contained functions for computing a legal move and returning it. We made it a friend class of Board, so that it would be able to easily access information about the state of the game when computing a move. Implementing level 1 was as simple as calling the allLegalMoves function in the Board class, and generating a random number from 0 to the size of the returned list and returning the move at that random index. Level 2 also generates all legal moves, but then goes through each move, and adds it to either a list of capturing moves by checking the piece at the destination coordinate, or a list of checking moves by making the move, calling isCheck, and undoing the move. Then, if the capturing list is not empty, return a random move from that list. If capturing is empty and checking isn't, return a random one from there. If they're both empty, return any legal move. Level 3 uses a bit more of a sophisticated system. The value of a move is appended to the end of the vector representing the move, and is initially set to 0. All legal moves are generated, and then they're checked if they are a capturing or checking move, like in level 2. They are now also checked if they are avoiding a capture, by checking if the destination coordinate is not being "attacked" by an enemy piece (an enemy piece can make a valid move to it or it can be en passant'd). If they are avoiding a capture, 4 is added to the value, if a capture is made, 2 is added, and 1 if a check is made. This way, the priority as specified in the document is upheld, and moves that accomplish multiple conditions are given more priority. The moves are iterated through to find the move with the highest value, and a random one is chosen if there are multiple.

Resilience to Change

We feel that our project can easily implement new changes to our code and design. We feel that we had low coupling and high cohesion, thus implementing new changes would be a quick fix. For example, if there were a new rule implemented on how Pieces move, we would simply edit and change each Piece class to accommodate the new rule. If the input syntax were different, we would go into our Main class and change how we receive input. If there were a whole new feature to the program, we would add a new class or function, depending on the change to implement the new feature. Regardless of the change, we feel that our code has strong pillars and fundamentals to support most possible changes and variations to Chess.

In our code, we feel that we had decently low coupling and high cohesion in our implementation. We have very limited friend-relationships in our modules, only with the Computer class being a friend to the Board class. The majority of our modules pass arrays and smart pointers, rather than the values itself thus decreasing coupling and increasing security. None of our classes share any data and all non-primitive data types are handled with smart pointers and/or arrays. Additionally, all functions in our Piece class deal strictly with Pieces, and the same logic is applied to our Board class and Computer classes. All in all, due to our low coupling and high cohesion of our code, implementing any new change does not largely impact how our code is structured.

Answers to Questions

1. We would implement a book of chess openings by having a list of trees. Each node in the tree will have a string field representing a move, for example the string "e2e4" means move the piece at e2 to e4. The root nodes of the trees in the first list will represent the first moves of different openings that the player can play. The immediate children nodes of these root nodes will represent the opponent's possible moves in response to those moves. Then, each of those children nodes will have children nodes that represent the move the player could make in response, with reference to some opening strategies. Then each of those nodes will have children representing the opponent's possible moves, and so on until the opening is complete. If the computer is going first, it can randomly select from the list of trees, and execute the move in the selected tree's root node. If the computer goes second, it will select a tree from the list based on what first move the opponent made. In either case, once a tree is selected, it will be traversed down one level per turn, where each level alternates between representing the computer's move and its opponent's move. When the opponent makes a move, we find the node in the current level that corresponds to that move to see what the computer can do in response. If there are multiple possible moves that the computer could make (since different openings can start the same way), the computer will simply randomly choose a move, and traverse down that node. If we reach a node that has no children, that means there is no opening in our book that considers that case, and the computer will move past this opening phase and play the game as normal. Constructing each tree to represent openings would be very time consuming, so it would probably be helpful to write a parser that will take in an opening written in a string such as "e2e3 d7d6 e3e4 d6d5" where moves alternate between each player and are separated by spaces, so that each move can be parsed and added to a tree as a node accordingly. This would allow us to simply write a bunch of openings in this easy to write format, and feed it to our program as a file.
2. In order for players to "undo" their moves, everytime a player makes a move, I would save a copy of the board to a board pointer. Thus, if the player would want to redo, I would simply assign the board to the previous board pointer and thus the move was "undone". To have an infinite number of undos for each player, I would simply create a Stack of boards. We can call the Stack "gameHistory" and everytime we undo, we would pop the top of the stack and assign the current board to that board. By having this "Stack of boards", we would also be able to have a game highlight/replay function since we have access to all the boards that were played and not undo'd in the game.
3. To make our program into a four-handed chess game we would first have to modify theBoard in the Board class to extend each side of the board by 3 ranks. In four-handed chess there are 2 game modes: Free-for-All and Teams. Therefore, in the Board class we will also need a string called "gameMode", to indicate which game mode is being played. In Free-for-All mode, the win condition is typically based on points, thus, we will need a method called "addPoints" which decides how many points should be given to a player based on some criteria. Also, we will have to update the "checkGameEnd" method to check for points instead of checkmate to decide who won in this mode. In Teams mode,

the rules of the game are very similar to regular chess. The main change is that there are now 2 players per team instead of 1. We will need an additional integer field in the "Piece" class called "player" to decide whose turn it is and which pieces they own. Lastly, in both game modes the rules for pawn promotion differ from regular chess. In Free-for-All pawns promote to queen on the 8th rank(middle of the board) and in Teams they promote on the 11th rank. We will need to modify the "promote" method in "Pawn" class to account for this.

Extra Credit Features

Undoing Moves

Once we finished all the required parts to chess, we decided to create an undo function. To undo moves, a vector of Board pointers were created in main. After the setup, and after every move, the vector of Board pointers would "update" by pushing the most recent Board onto the stack. Once undo is called, we pop the most recent Board iteration, and assign the current game to the top of the stack. The main challenge with "undo" was working with mutating values that pointers were pointing to, and we solved it by implementing a copy constructor for our Board class. Thus we have "undone" our move with all the correct pieces in the correct spots.

Smart Pointers and Vectors

We extensively made use of shared pointers and vectors in order to represent the chessboard, the pieces, and lists of moves and coordinates. This meant we didn't have to manage our own memory at any point, as per the specifications for this extra credit challenge.

Computer level 4+

We decided to implement level 4 and above for the computer player using the minimax algorithm. This means that for any given board state, the computer generates all legal moves, and then for each of those legal moves, generates the possible opponent responses to those moves, and its own responses to each of those moves, and so on for a specified depth. Then for each sequence of moves, the resulting board state is evaluated according to some function, which returns a value that is more positive if white is winning, and more negative if black is winning. Then we propagate these board states back up this "tree", where each level alternates between nodes representing the maximum value of the moves that white makes and the minimum value of the moves that black makes. This way, the algorithm makes the assumption that each player will always choose the response that gives them the most preferable board state. Ultimately the computer will then receive a value for each legal move it can currently make, and then make the move with the highest value if it is white, and lowest if it is black. Our evaluation function simply added up the weighted values of white pieces (queen being worth the most and pawns being worth the least) and subtracted the sum of values of black pieces. A significant value was added or subtracted if one of the players were in check, and an enormous value (effectively infinity) was added or subtracted if a player was in checkmate. Since we end up generating and evaluating a lot of moves with this algorithm, it can get pretty slow pretty

quickly, and we saw that it was taking over a minute on average to just look 3 moves ahead. In order to speed it up, we first made the optimization known as alpha-beta pruning. This is where if a child of a maximizing node B has a value greater than the value of a previously evaluated sibling maximizing node A, then we can discard node B from consideration, as it's guaranteed that it will have a value greater than node A, which means the parent minimizing node C of A and B would never choose the value of B instead of A. This means that we can avoid checking the rest of the children of B, which can save us some time. A similar logic applies in the inverse with disregarding minimizing nodes. This optimization usually saved a good amount of time, but it wasn't completely reliable, as it only had a significant benefit when the moves were ordered in such a way that better moves were evaluated first, which would allow for future worse moves to be pruned. In order to accommodate for this, we implemented a moveOrder function, which takes in the list of moves generated by allLegalMoves and appends a value to the end of each move vector representing the expected value of the move. In this function we used some heuristics to estimate which moves are more likely to be considered good, by considering whether or not the move was a capture move, and if it was, the difference in value between the capturing and captured piece (because it's usually a much better move to trade a pawn for a queen than the other way around), if the move checks the opponent, and more. After implementing this function, the alpha beta pruning was able to eliminate a lot more redundant searches and save us more time. This algorithm was used for all computer levels 4 + x, where x specified the depth. So level 5 looks 1 move ahead, level 6 looks 2 moves, and so on.

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?
This project taught us a lot about the importance of being able to subdivide the workload when working in teams, as being able to work on different files simultaneously greatly increased our productivity and efficiency when working on the project. The work distribution also goes beyond just deciding who writes code for which files; at one point we had someone working on developing some of the core mechanics of the game, someone working on making the graphical display, and someone working on making various test cases for different scenarios. All of these roles were equally as important, but are still very different from one another. We also learned the importance of convening frequently to discuss blockers we were currently facing, and went over them together. These meetings also helped with getting each other up to speed on whatever parts of the project we were working on, so that there wouldn't be conflict between the way different components worked with each other.

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 use github to work on the project together as opposed to updating files on google docs. This would have made updating files more seamless and safeguard against mistakes. This would have also made undoing changes much simpler and available backups would have given us peace of mind. Additionally, it would have been more efficient to get the core board and piece class working sooner so that we could work on our

individual segments sooner and facilitate early testing. Another change would have been to implement the graphical display after the main function was complete to reduce the amount of changes needed to be made to the graphical display functions. Lastly, if we had used smart pointers from the beginning, debugging memory leaks would have been much simpler.

Conclusion

In the end, the Chess project was challenging and time-consuming but extremely rewarding. Being able to extend our knowledge from CS246 to an applicable real-life game was captivating and engaging. It allowed us to dip our toes into a different realm of coding which was both gratifying and compelling, and encourages us to dive deeper into the world of computer science.