**Introduction**

Over the past few weeks, our group has created a near-comprehensive chess game using C++. Our game uses many OOP and general programming principles, and can facilitate a full game of chess. Some features/capabilities include support for a human player and three different computer levels (being able to play a game with any combination of the player types), as well as chess game mechanics such as castling, en passant, and pawn promotion.

**Overview**

The structure we have chosen for our program was to split the elements of chess into smaller, more manageable parts through the use of OOP and inheritance.

Our program is run by a Game class. This class stores all aspects of the game, including the players, the board, and the two displays (text and graphical) used to output the game board and the pieces.

The gameboard class owns all the pieces used during the game, as well as manages the pieces in play. Additionally, the gameboard stores a vector of Move objects that are used to track how the pieces move around the board.

All our individual pieces are subclasses of a Piece base class. They all inherit attributes such as position, colour, a symbol. Additionally, all pieces can determine things such as whether their king is in check or whether the piece is able to move.
Each piece abstracts functions that determine whether a piece is able to move a certain way or capture another piece, and these are all specific to the piece.

The game progresses over a series of Moves, which are all stored in Move objects. These move objects store the start and end location, what piece was being moved, and what piece was captured, if one exists.

Additionally, each Game consists of two players. Each of these players are abstracted from a Player abstract base class. Possible players are humans, where all moves are determined by a user, or computers 1-3, where moves are determined at random, with each subsequent computer level making slightly better, and more sophisticated moves.

Finally, there are two types of displays, a text-based display as well as a graphical display using XWindow. These are called by the game function.

## **Design**

*General Design:*

Deciding the exact role of each class, and which specific functions should be implemented was quite the challenge. There were many functions necessary where we felt two different classes could both be responsible for the implementation. For example, we did not know exactly which object should be responsible for making the move or the validation for whether a move puts the King in check. Ultimately, we chose a design in which the piece validates its own movements, and will make the move. The player invokes movements of the piece. The game board is used to track the position of all the pieces, modify the board, and also allows each piece to have access to all the other pieces. Lastly, the game class is used to control the flow of the game.

When making such decisions, we consistently referred back to a real chess game to determine exactly who does what. The chessboard stores, and is able to access all the pieces. Each piece has its own internal move logic and validates whether a movement will put its King in check. The players invoke movement of the pieces.

*Changes to Original Design:*

While we did not have many big structural changes to our design from our initial plan, there were several smaller changes and additions we had to make to ensure our program would be able to run smoothly and efficiently.

Smaller/trivial Changes

- We had various small changes to variable and function names in order to improve clarity and consistency throughout the program.
- Some select functions had their return types changed to increase efficiency.

Function Additions

- Many functions were added to the GameBoard class to help us navigate our program more efficiently. These include functions such as setPiece and setPieceAt which add a piece to the game board, either by using the pieces index value or overriding it with a value passed into the function in the case of setPieceAt. More examples of functions that were added were removePieceAt, which does exactly as you would expect, and validBoard, which determines the validity of the board prior to exiting setup.

Pointer to Reference Changes

- Many pointers storing data were changed to references. This is because a lot of these data points did not have to use pointers, and it made memory management easier.

PossibleMove Object

- PossibleMove no longer inherits from Move, as we found that Move had extra data, such as a captured piece field, that PossibleMove did not need. Additionally, PossibleMove had no use for Move functions, so we decided to make it it’s own standalone class.

A comprehensive list of changes can be found in Appendix A of this document.

**<u>Resilience to Change</u>**

Our program already follows all of the program specifications as outlined in the project guidelines. Furthermore, our program is structured in such a way where should it be necessary in the future, additional features can be easily implemented into our current code. Below are some examples of said features, and how they could be implemented:

Undoing a Move

- As we already have a vector of Move objects stored in the GameBoard class that contain data for every move (what the piece was, its start and end location, if a piece was captured and if so what the piece was), in order to implement the undo feature, we would simply need to pop the last Move element of the vector, and use its data to reverse the move (i.e. moving the piece from its end location to its start location, putting the captured piece at the end location if the piece exists). Additionally, we can determine if a special move was performed by analyzing the move (eg. if a king was moved 2 spaces, then it must have been a castle, and we need to move the rook accordingly). More information is available below where we cover undoing moves.

Better Computer Players

- All of the logic to determine whether a move is valid or not is already taken care of. Thus, the only thing that would need to be implemented would be the logic for selecting moves, as well as the additional class (which would have an identical structure to our current computer player classes).
- Additionally, many of the things that would probably be considered when making smarter computers would be easy to implement through the use of existing functions. As an example, if you wanted to have a computer prefer making moves where the opponent cannot check you on their next move, you would simply loop through all the opponents pieces (a function exists to obtain all your opponents pieces into a vector), check their valid moves (a function exists to validate moves and return them in a Move vector), and check to see if your king would be in check. Then, you would need to alter the logic in the preferredMove function located in each of the computer player classes (not including computer1) to only add those moves where your opponent has no checks to your preferred move vector.
- Another possible addition to a smarter computer would be to prefer capturing pieces of higher value. Again, with our existing code, this would not be that difficult to implement. You would be required to alter the canMove() function in piece.cc to return an int, where the int is the value of the piece at the end location of the move. As an example, if you were to move your rook from a1 to a8 with a8 being empty, it would give that move a value of 0. If there was a queen on a8, it would give that move a value of 9 (the value of a queen in chess). You could add points to the value if the move avoided capture or

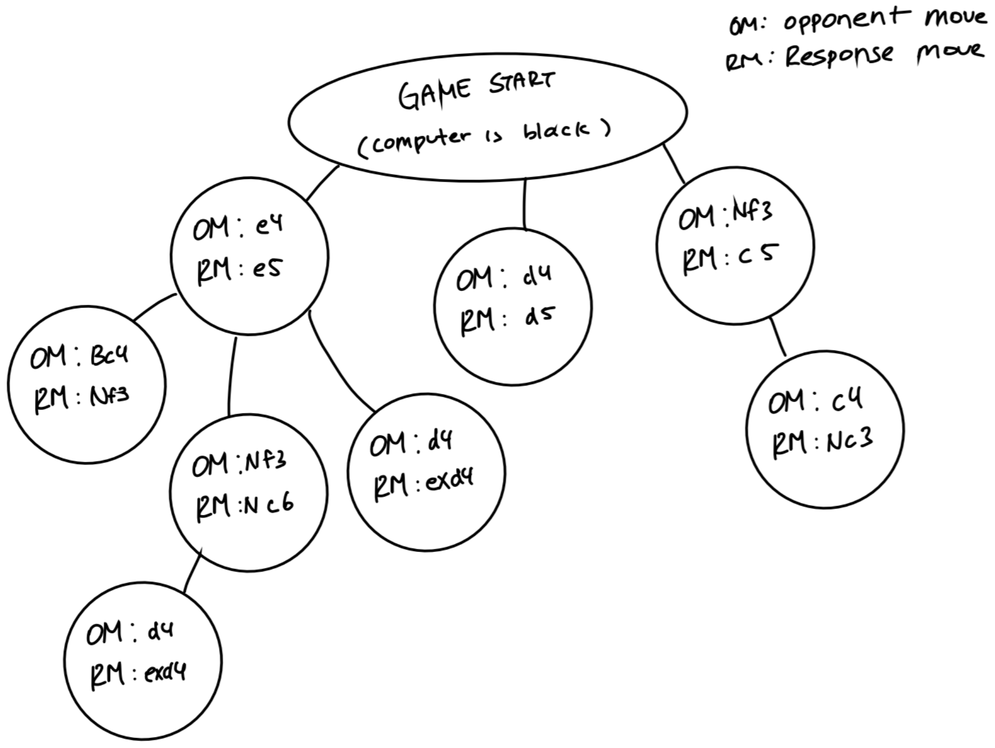
gave a check, or take away points if your piece could be captured upon moving to that square.

We have structured our code in such a way where the addition of new features would be as seamless as possible. Many functions exist that can be used in a variety of situations, making our program overall very resilient to change.

## Answers to 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.**

A good way to implement this feature would be through the use of a tree. Each node would contain an opponent's move, and what the program's response move should be. Each child node would then have common responses to the move the program just played, and in turn what the program should play in response to that, and so on. Once the program can no longer find a child node that contains the opponent's response, it is then time for the program to find new moves on its own. As an example, consider the following tree:



And the following game:

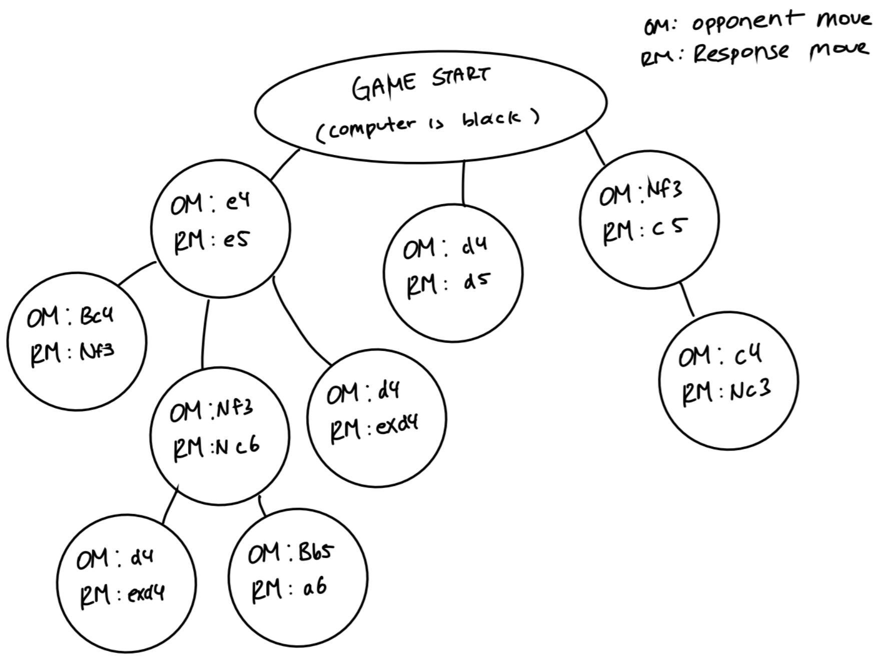
| Opponent move (White): | Computer Move (Black): |
|---|---|
| e4 | e5 |
| Nf3 | Nc6 |
| Nc3 | Bc5 |
| Nxe5 | Nxe5 |

In this scenario, the computer would follow the tree until the third move. The computer would start at the root of the tree. Since the opponent played e4, the computer moves to the leftmost node and makes the move e5. The opponent then plays Nf3, so the computer moves to the middle node and makes the move Nc6. Now, the opponent plays Nc3. Since this move is not stored by any of the child nodes, the computer must now make a move on its own. The game continues from here, with the computer having to make its own new moves.

Adding a new line (or sequence of opening moves) using this structure would be quite simple. Keep following the tree until you reach a point where none of the child nodes match the move response you are looking for. Then, add a new child node with the new response, and then add subsequent child nodes for each following move you have stored. Say you wanted to add the following line (move sequence):

| Opponent move (White): | Computer Move (Black): |
|---|---|
| e4 | e5 |
| Nf3 | Nc6 |
| Bb5 | a6 |

The computer would again search through the tree until it encounters an opponent's move which it does not have any data for. Upon this happening, we create a new node of the opponent's move and the computer's response move and set it as the child of the last node we have data for. If there happen to be multiple new moves in the sequence we are adding, we would just create one long chain of nodes outlining said sequence. Here is what the tree would look like after the inclusion of 3. Bb5 a6:



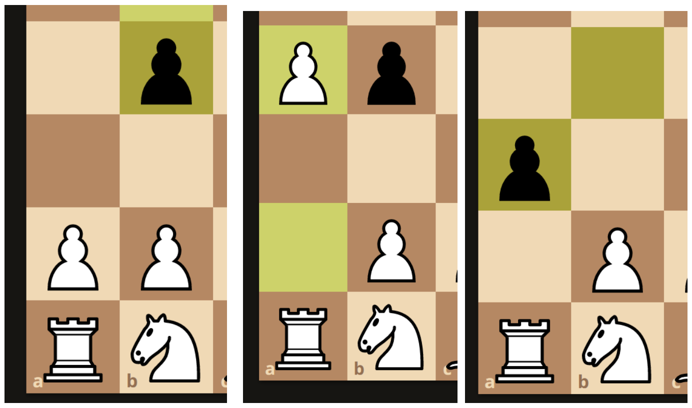
**Question: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?**
We could store a vector containing the set of moves that had happened throughout the game. This vector would be in chronological order, and would store the piece that moved, its start and end location, as well as a pointer to the piece that was at that location previously. Note that this could be a nullptr, indicating that the space is empty.

In order to undo a move, we would just have to traverse the vector in reverse. Starting with the last element of the vector, we move the piece that was moved to its starting location, and if there was a piece at the ending space, place that piece back at that space. We then remove that move from the end of the vector, and continue on with our day. We can now either keep moving forwards in the game, or undo another move.

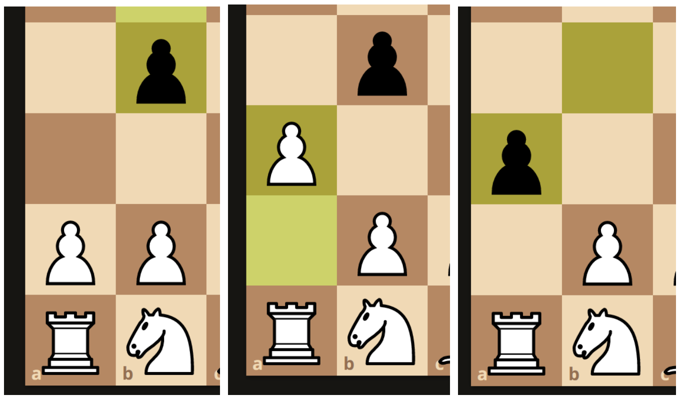
This process gets a little bit more complicated for special moves such as castling and en-passant as we must check previous moves to see the location of the opposing pieces.

In the case of en-passant, we must check one move previous to the move we want to undo (eg. If we want to undo black's move 20, we must first check white's move 20. If we want to undo white's move 16, we must check black's move 15). This is because we must know if the pawn was taken normally, or through en-passant. As an example:

| Case 1: En Passant |
|---|



Case 2: Regular Capture



Simply knowing that the black pawn took the pawn on a3 does not tell us whether it was a normal capture or an en passant capture. We would only know that black moved from b4 to a3 with its pawn, capturing white's pawn in the process. We would need to check the previous move to see specifically if white played a4. If they did, then black capturing on a3 is an en-passant capture, and white's pawn should be moved back to a4 after undoing the move. If white did *anything* else, then this was just a normal capture, and white's pawn should be placed back on a3.

Castling also poses a problem, although a bit easier to deal with. If we see in the previous move that the king has moved two spaces, we know the only way this would be possible was through a castle. We then must locate the rook that the king castled with, and move both back to its appropriate square.

Lastly, in the case of pawn promotion, we would need to confirm that the previous move was a pawn move. Again, simply knowing that (for example) a piece moved from a7 to a8 and is now a queen doesn't tell us if the piece was a queen to begin with, or was a pawn and promoted to a queen upon moving to a8. We would need to see what piece was associated with the previous move in order to determine what piece to place on a7 upon undoing.

**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**

- Increase the board size from 8x8 to 14x14, ensuring that there is a 3x3 grid on each corner where no pieces can be played.
- Add in two more players, which can either be humans or computers.
- Change the move order from alternating between white and black to cycling between players 1 through 4.
- The pieces can no longer be identified as isWhite or not isWhite. This variable could likely be replaced with an int (players 1-4), or a char (players a-d).
- The game should now end when 3 of the players are in checkmate, rather than just one

## <u>Final Questions</u>

a) What lessons did this project teach you about developing software in teams?

As a group, we all learned a few valuable lessons about developing software as a team. First, we learned how important communication and transparency is. We thought that this would be a given at first - obviously communication is important in a group setting - however we failed to realize the actual significance that communication makes in a group setting.

As we are all developing, using, and making changes to the same program, edits or changes one person makes could greatly impact the functionality of another part of our program that someone else might have worked on. There have been multiple instances where one part of our program stopped working correctly after someone had made an edit, not anticipating that it would have an effect on another part of our program. Thus, knowing exactly what parts of the program might use a function ‘foo’ is valuable information to someone who is editing ‘foo’ in the future. The most effective way to get this information through effective communication with your group members.

Communication doesn’t even have to be face to face (or in a voice call), it could be through documentation of a part of a program. As an example, if person A were to leave a note on ‘foo’ of what other functions use ‘foo’, when person B goes to edit ‘foo’, they know exactly where to look to see if their changes may negatively impact another part of the program.

Another thing we learned related to communication is how important it is to meet up as a group and ensure everyone knows what is going on with the state of the program, what is done, what needs to be done/fixed, and how the program works overall. If someone is unclear on how the program works and what parts of the program are functioning correctly or not, it will be hard for them to efficiently make progress in developing new features. Making sure everyone knows what is done and what works, and how the program works results in increased efficiency for everyone, a more logical finished project (i.e. no redundant functions, no absurdly long function calls (board.piece->canMove.canCapture(this, isWhite), as an example)), and most importantly less stress for everyone involved (debugging is hard :( ).

Lastly, we found out how effective it is to consult your team members if you encounter a problem and don't know what to do. Oftentimes, using two (or three) heads is better than one, and it isn't uncommon that one of your group members has already encountered a similar error. As an example, one of our members Ananya might be better at dealing with memory issues, while Justin is generally better at dealing with logic flow through a program.

(b) What would you have done differently if you had the chance to start over?

If we were to redo this project, one thing we would do is to spend more time planning. While we did create a general structure and had a pretty good idea of how our program should function and operate throughout, a good portion of our program had to be altered on the fly. We had to create and implement new functions that we did not anticipate during our planning (eg. many functions in the Piece class needed to be created after our planning phase), alter the purpose, location, or logic of some other functions (eg. checking if the king is in check while making a move rather than in each individual piece class), and change the structure of some objects to increase functionality and efficiency (such as PossibleMove not being a subclass of Move, as we found that the objects do not need to be that similar).

In order to accomplish this, we would try to create some pseudocode during our planning phase. It doesn't have to be that much, but just a general outline of how the logic should work. Knowing what functions will need what information can help us better organize where our functions are located and how to structure our classes.

Obviously, making changes after our planning phase is nearly inevitable, but spending more time planning earlier will pay off with dividends when we are actually trying to implement and test our program.

## **Conclusion**

In conclusion, this was an enjoyable project for our entire group, and it was a good culmination of our learned skills throughout the course. We are all very proud to have made a working project, and will hopefully be making bigger and better things in the future! :)

Appendix A

## UML Diagram Changes from Due Date 1 to Due Date 2

Piece

- All member attributes are now protected, and accessible by the inherited classes. These were previously private.
- board is a reference to a GameBoard object. This was previously a pointer.
- Added canMove (int posTo, bool &captureOrCheck) checks whether a move is possible.
    - Takes into consideration whether the king will be in check after the move.
- Renamed movePutsYourKingInCheck () to isKingInCheck (). This is also not a pure virtual function anymore and its implementation is completed in the base class.
- Move () function is not a pure virtual function anymore, and its implementation is completed in the base class.
- Pawn derived class now declares and defines the function en passant (int posTo).
    - This determines whether the move of a pawn to posTo is a legal en passant move.
- King derived class now declares and defines the function canCastle (int posTo).
    - Can castle determines whether the move of a king to posTo is a legal castle move.

PossibleMove

- Does not inherit from the Move.
- Just contains position from, position to and the piece itself.
- This saves memory.

GameBoard

- Added the member variable moves, which is a vector of pointers to Move object's.
- setPiece (Piece* piece) function now takes a pointer to a piece as an argument instead of an integer argument.
- Added setPieceAt (Piece* piece, int pos) function sets the element at index pos in the board to piece. It also changes pieces current pos (piece->setCurrPos(pos)) to the argument pos.
- isOccupied (int x, int y) has been removed. All positions on the element are accessed by their index from 0 to 63.
- Added removePieceAt (pos), which removes the piece from the board at index pos.
- Added addCapturedPiece()
- Added validBoard(), which validates that the board has the current setup conditions.
- Added addMove(Move* move), which adds the argument move, to the list of moves.
- Added lastMove(), which returns the most recent move (the last element in the moves vector).
- Added deleteBoard(), which clears the game board.

PossibleMove

- A possibleMove object used to have a Move object.
- Now, the possibleMove object just has the posFrom and posTo variables, and the pointer to a piece that it moves.

Game

- Rather than have pointers to the TextDisplay and GraphicalDisplay, we directly declare and initialize the objects in the Game constructor.
- p1 and p2, which were pointers to the Player objects, were renamed to white and black for clarity.
- Rather than have a reference to the currentPlayer, we now have a boolean variable called whitesTurn to keep track of which player's turn it is.
- Added member variable gameRunning, which keeps track of whether there is a game currently in progress.
- Added member variable setupDone, which keeps track of whether the setup process was completed.
- Added the following functions to control game flow:
  - endGame()
  - initializePlayers(string whitePlayer, string blackPlayer): initializes players based on passed argument type (human, computer1, computer2, computer3)
  - initializeDefaultGame(): initialize default game with default chess setup
  - makeMove(): makes a move
  - resign(): resigns, and opponent is declared as the winner
  - display(): displays the current board
  - addPiece(string piece, string pos): added piece of type piece to position pos
  - removePiece(string pos): removes a piece at pos
  - switchTurns(): switches turns between white and black player
  - canExitSetupMode(): determines whether the board is valid
  - printScoreboard(): prints the current scores of the players

Player

- isWhite renamed to white
- removed isincheck variable
- removed isTurn variable
- removed level variable
- Added function hasDrawnGame() which increases scores by 0.5
- makeMove() was renamed to makeAMove()
- added moveSemantics which deals with all special move cases (en passant, pawn promotion, castling)
- added hasValidMove() which checks to see if a player has a valid move or not

Human, Computer1, Computer2, Computer3

- Renamed from HumanPlayer, ComputerPlayerLevel1... for simplicity

Computer1

- Removed scoreMoves because computer1 doesn't consider moves are good or bad

Computer2
- Removed scoreMoves, instead uses generateMoves() and generatePreferredMoves(), and if there are no smarter moves, generate will take over and generate a random move

Computer2
- Removed scoreMoves, instead uses generateMoves() and generatePreferredMoves(), and if there are no smarter moves, generate will take over and generate a random move

TextDisplay
- No friend function to overload the output operator
- Display is done through function, notify();

GraphicalDisplay
- Added drawBoard(), which draws the empty board.
- Added drawPiece(int index), which draws a piece at index.
- Added notify(), which draws the current state of the board.
- Changed parameters for the functions to draw each individual piece
    - Before functions just required bool isWhite. Now, the color, x and y coordinate and the size must be passed.
    - drawPawn
    - drawKing
    - drawQueen
    - drawRook
    - drawBishop
    - drawKnight