The program contains the main function which starts the program. Very little code is included in the main function. It's only purpose is to initialize the GameMaster object (described below) and start the game via this object.
The GameMaster class is the overarching class which manages the game on the highest level. It contains separate functions to start the game and make a move. It keeps track of the game state and informs the user when a player wins. Getting input from the user is managed by this class, as well as checking if the move is valid. Actual moving of the piece is delegated to the Board object. Valid moves for a piece are checked through a function implemented by each piece.
A Square class models a square on the board, keeping track of it's position on the board as well as the number of pieces that can currently attack the square. This attack information is modelled by an Attack class, which the Square class depends on.
Every piece has similar properties, which is modelled by the abstract Piece class. Every piece has it's own class which inherits from the Piece class. It contains information about the piece position, piece colour and piece type. I've also added an invisible property, which is useful when imitating a move to check it's effects without actually deleting a piece on the board if the move simulates taking a piece out. Apart from basic getters and setters, each piece has the following functions:

- attacksSquares: Returns a list of squares on the board that the piece can attack from it's current position
- validMoves: returns a list of squares that are valid to move on from the current position
- canMove: returns a boolean indicating whether the piece can move to the given position

Every piece then has it's own implementation of these functions.
The Board controls and manages the 8x8 grid of squares as well as the pieces on the squares. I decided to use shared pointers to represent the pieces since I wanted to represent a null value on positions with no pieces, which I can do through the use of nullptr. I decided to make it shared since other parts of the program are allowed to have references to this pointer, though ownership will only be maintained by the variable in Board. The Board has functions to move a piece, check if moving a piece puts the king in check, and calculates the attacks for each Square after a move is made.
There is a ComputerPlayer class which is in charge of implementing an algorithm for recommending a move based on the current Board object. The recommended move is plugged into the same function as a human move in GameMaster. This allows us to isolate the algorithm from the workings of the game.
Finally, we have an abstract View class which can be extended to create different types of views. The only view I have implemented is a text-based view, modelled by the class TextView. It contains a function to draw the board given a Board object.
The GameMaster class knows about every object, while no class knows about the GameMaster. This ensures that each class works on a specific function without worrying about the other components of the game.
A high overview of the interaction of the different classes when a move is made is as follows:
GameMaster reads move || Checks if move is valid by checkMove function in Piece || checks if move puts king in check by calling the appropriate function in Board || If valid move without check, passes move to Board to move pieces || Board moves the pieces || Board calculates attack on each Square || If square which king is on is attacked, GameMaster warns user of check || If check, GameMaster checks if king can move by a function in Board || If king cannot move, GameMaster tells the user who won.
If there are any changes in the rules, only changes to validMoves has to be made to get the set of valid moves according to the new changes. If input syntax changes, we only need to change the way input is received in GameMaster, convert it to a Square object, and pass it onto the corresponding move function in GameMaster which accepts Square object input. As we can see, the design allows itself room for modifications with relative ease.

(a) 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?

Working alone, the hardest part was keeping track of all the moving components and how changing one component affected the other parts of the program. The biggest lesson I learned was the effectiveness of modular code. Keeping functions small and to the point helped me put the pieces together on a larger scale. Separation of concerns was also a big lesson for me. Having parts of the program only focused on doing one task without worrying about the other helped me isolate problems and break down complex problems into easier, manageable parts.

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

I would definitely have spent more time on the design aspect of the project. My first approach was going head first into programming while slightly keeping the original UML design in mind. This led to a whole host of problems. Bad design, memory leaks, high coupling and huge files were some the difficulties I started encountering. Midway through the project, I took a step back and started over, this time keeping design on the forefront of my mind. I found this approach to be much better.

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.

I would implement the logic for this in the ComputerPlayer class. By adding an additional parameter to accept the opponent's move, I would run it through a pre-built dictionary of moves and see of there is an appropriate response which I can return.

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

I would keep a list of moves played, along with a list of pieces which were killed at every move (null if no pieces were killed). The undo command would then just play the move in reverse and add the appropriate piece back to the Board.

instance.

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.

The game board would need to be changed. All logic for the valid moves would need to be changed. Logic for handling 2 more players would need to be added to GameMaster.