Overview

The overall structure of our project is as follows:

The main function in chess.cc includes the main command interpreter and handles most of the input and output, both in text and graphically. It contains a pointer to a Board object, pointers to two Player objects (one white and one black), and additionally keeps track of the overall score of the white and black sides. Upon receiving a command from the user, the main function interacts with these objects to achieve game functionality, and updates text and graphical output when appropriate.

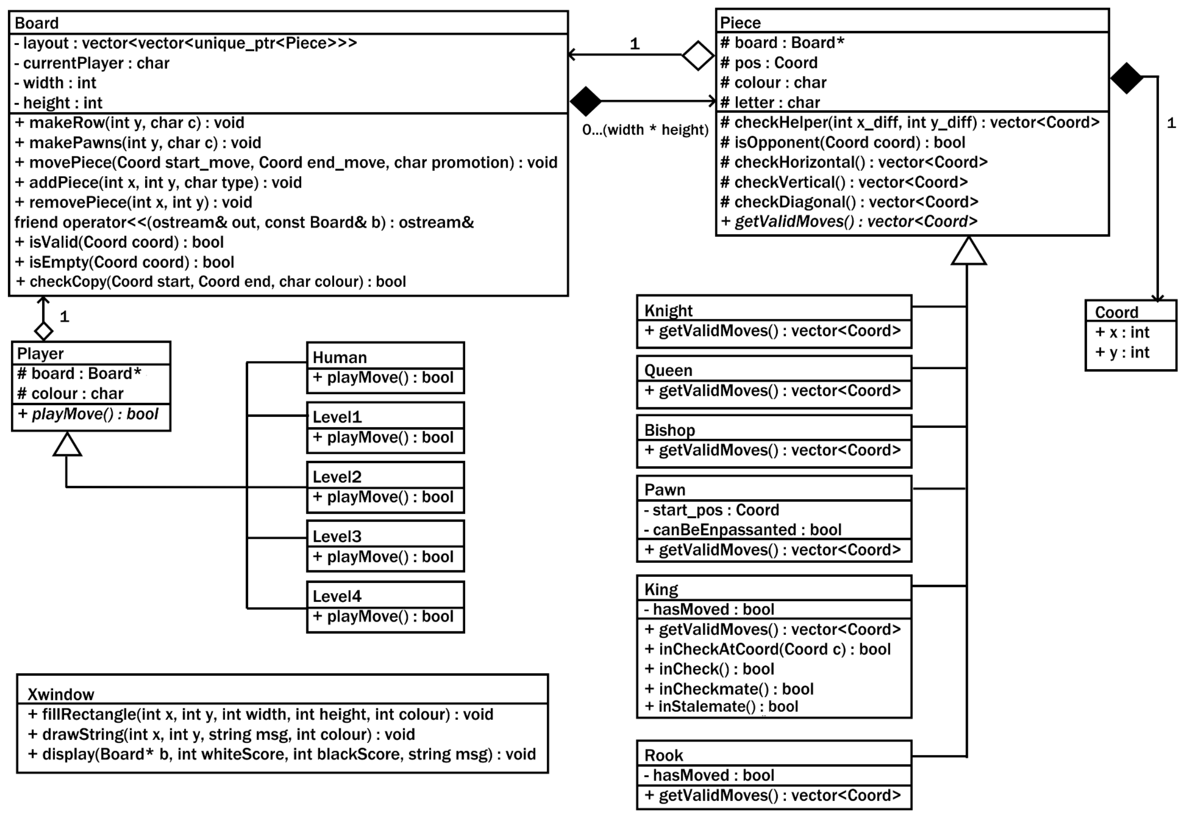
The Board object contains a 2D vector of pointers to Piece objects, and keeps track of the colour of the player whose turn it is currently. It includes addPiece() and removePiece() methods for use with setup mode, an output operator overload for the text display, and a movePiece() method for use with the *move* command, which moves a piece from one coordinate to another. This movePiece() method assumes that any pair of coordinates passed in constitutes a valid move, and includes additional handling for pawn promotion, en passant, and castling.

Piece objects all include a public getValidMoves() method, which returns a vector of valid coordinates that the piece can move to. This method is implemented differently for each of the concrete subclasses of Piece (Pawn, Rook, Knight, Bishop, King, Queen), however to cut down on code repetition the Piece superclass includes protected helper methods to determine these coordinates. The Pawn, King, and Rook classes have additional fields to assist with en passant and castling, and King has additional public methods to return whether or not the piece is in check or checkmate.

Player objects rely on a playMove() method, which is pure virtual in the abstract Player superclass and implemented in the concrete subclasses of Player (Human, Level1, Level2, Level3, Level4). In the case of a Human, this method is implemented by retrieving a pair of coordinates from standard input, then doing checks to make sure that the pair constitutes a valid move (e.g. Is there a piece at the starting coordinate of the player's colour? Is the ending coordinate a valid move for that type of piece? Does the move place the player's king in check? etc.), then finally calling Board::movePiece() once the move has been verified. In the case of a computer player, the state of the Board is evaluated to determine a list of valid moves that can be made. Then, the most appropriate of these moves is determined, and then Board::movePiece() is called as before.

Finally, board coordinates are expressed using a dedicated Coord class holding x and y values, and the graphical display is achieved using an Xwindow class (adapted from the graphics demo included in the project folder) which has a method to produce a display when passed a Board pointer and other game info.

Updated UML


Board
- layout : vector<vector<unique_ptr<Piece>>>
- currentPlayer : char
- width : int
- height : int
+ makeRow(int y, char c) : void
+ makePawns(int y, char c) : void
+ movePiece(Coord start_move, Coord end_move, char promotion) : void
+ addPiece(int x, int y, char type) : void
+ removePiece(int x, int y) : void
friend operator<<(ostream& out, const Board& b) : ostream&
+ isValid(Coord coord) : bool
+ isEmpty(Coord coord) : bool
+ checkCopy(Coord start, Coord end, char colour) : bool
1
0...(width * height)
Piece
# board : Board*
# pos : Coord
# colour : char
# letter : char
# checkHelper(int x_diff, int y_diff) : vector<Coord>
# isOpponent(Coord coord) : bool
# checkHorizontal() : vector<Coord>
# checkVertical() : vector<Coord>
# checkDiagonal() : vector<Coord>
+ getValidMoves() : vector<Coord>
1
Coord
+ x : int
+ y : int
1
Player
# board : Board*
# colour : char
+ playMove() : bool
Human
+ playMove() : bool
Level1
+ playMove() : bool
Level2
+ playMove() : bool
Level3
+ playMove() : bool
Level4
+ playMove() : bool
Knight
+ getValidMoves() : vector<Coord>
Queen
+ getValidMoves() : vector<Coord>
Bishop
+ getValidMoves() : vector<Coord>
Pawn
- start_pos : Coord
- canBeEnpassanted : bool
+ getValidMoves() : vector<Coord>
King
- hasMoved : bool
+ getValidMoves() : vector<Coord>
+ inCheckAtCoord(Coord c) : bool
+ inCheck() : bool
+ inCheckmate() : bool
+ inStalemate() : bool
Rook
- hasMoved : bool
+ getValidMoves() : vector<Coord>
Xwindow
+ fillRectangle(int x, int y, int width, int height, int colour) : void
+ drawString(int x, int y, string msg, int colour) : void
+ display(Board* b, int whiteScore, int blackScore, string msg) : void

Design

We decided to use an ownership relation between the Board and the Pieces it associates with as there is no reason to have a Piece on its own without an associated Board. With this in mind we decided to implement the game board itself using a 2D unique_ptr<Piece> vector, so that any deletion of Pieces would be automatically handled upon the death of the Board object. This permitted us not to write a custom Board destructor. On the other hand, Pieces have an aggregation relationship with Boards as it would make sense for a Board to live on if a Piece is deleted (for example, if a Piece gets captured - the Piece object gets deleted but the rest of the game continues). However Piece does have an ownership relation with a Coord which stores its x and y position on the game board. As the Coord object exists only to store information about the Piece, we believe an ownership relation is most appropriate here.

With respect to the Piece and Player classes, we decided to use the strategy design pattern. This is because each of these types have certain operations that are common to all objects of that type. In the case of Player, this is the playMove() method, which determines which move the player should play. In the case of Piece, this is the getValidMoves() method, which determines which spaces on the board that piece can move to. As the implementation of these methods varies only with the dynamic type of the object, we decided it would be preferable to use the strategy pattern so that the algorithms behind these methods can be selected at runtime. On reflection, we believe that this pattern was the right choice for the Piece classes. However, another pattern may have been more efficient for the Player classes, as discussed in the "Final Questions" section.

Finally, we will discuss coupling and cohesion. The coupling shown by the modules of our program is medium. We have certain public methods that return vectors rather than primitive types (such as Piece::getValidMoves()), and some cases where the result of a method from one module will affect that control flow of another. For example, in Human::getMove(), a different result can be returned (indicating an invalid move) if the desired move would put the King in check. However there is no global data shared among modules and the only friend in the program is the overloaded output operator for the Board class. Meanwhile our cohesion is relatively high, as each element in each module manipulates the state of one particular object over time.

Resilience to Change

We have designed the program in an attempt to minimize the work needed to accommodate changes in program specifications. In this section we will go over some possible ways that the specification could change, and then discuss the work required to accommodate these changes.

Change in piece behaviour:

As each type of piece is implemented in its own subclass, we can change the behaviour of that piece in isolation fairly easily by modifying that piece's getValidMoves() implementation. For example, if we were to decide that the Pawn should be able to move backwards, we could simply add an additional section to Pawn::getValidMoves() checking if the space behind the pawn is open to move in. For more complicated behaviour in the vein of castling, especially behaviour where two or more different pieces interact with each other in some way, some additional handling may be required in Board::movePiece() as well.

Addition of piece types:

As each piece type is implemented as its own subclass we could add piece types by adding concrete Piece subclasses. We would also need to update the Board::addPiece() method to properly instantiate a new object of the additional type when passed as an argument.

Change in computer player behaviour:

Each computer player level is implemented as its own subclass, so changes to the behaviour of individual levels can be made in the desired concrete subclass. However, in the case of a more general change to all computer players (for example, if we decide that all computer players should no longer move rooks), this means that each difficulty level would have to be changed individually.

Addition of computer player difficulty levels:

As with pieces, new computer player difficulty levels can be added easily by creating new concrete Player subclasses, with the desired behaviour implemented in the associated playMove() method. Additionally the command interpreter in the main function would have to be updated to properly instantiate a new object of this type when handling the *game* command.

Change in game rules:

The work that would have to be done in this case depends on which aspect of the game rules changed. For changes to individual pieces, changes can be made to the Piece subclasses as explained above. If the rule changes concerned the interaction between pieces during gameplay, then most of the modifications would have to be done within Board::movePiece(), as well as additional changes in Piece subclasses if necessary. For changes to the rules of the check or checkmate conditions, then alterations would have to be made to the logic in King::inCheck() and King::inCheckmate() respectively.

Change in input syntax:

Most input is handled by the command interpreter in the main function. The only other place in the program where input is handled is by the Human::playMove() function, where a human player can input their desired move. Therefore changes to the syntax for arguments to the *move* command can be accommodated in Human::playMove(), while any other changes in program syntax can be changed by modifying the expected syntax in the command interpreter.

Change in output:

All textual output for the game board is handled by the overloaded Board output operator, which is implemented in Board.cc. Therefore changes to the game board output can be made in that function. On the other hand all graphical output is handled by the Xwindow class. Certain portions of the graphical display which are expected not to change, such as the background and game title display, are handled by the constructor, while other portions such as the game board and score display are redrawn every time Xwindow::display() is called. Therefore changes to the graphical display can be made in those sections. If we were to add additional information to the graphical display (e.g.

display a notification if setup mode is currently active) then we may have to modify the signature of the Xwindow::display() method and pass in additional parameters.

Additional possible changes to the program (board size/layout, number of players, additional features) and the changes they require are discussed in the following "Answers to Questions" section.

Answers to Questions

1. *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.*

Chess openings and chess moves are often recorded in Portable Game Notation (PGN) which is a text file that is both human readable and computer interpretable. In order to implement a book of standard openings, the project folder would have to contain PGN files containing standard openings. Then, we would modify the *Board* class to take as input the PGN files. This would require writing a method that parses the PGN files and calls *Board::movePiece()* for the players' respective moves. Depending on the requirements, after reading the file, the *Board* class could "hand off" the game to the players of the game to continue playing. The response to this question is unchanged from our original design document.

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

This would require keeping track of all moves made by the player in the game along with the captures associated with the move. We could do this by storing the moves in a vector of structs called perhaps *move_info*. *Move_info* would contain the starting and ending coordinates, the piece that was captured if a capture occurred and whether a promotion occurred. If the player undo's, we read in the most recently played move and reverse the starting and ending coordinates and play that move. The "move" that would be played may not be valid since it is backwards, so we may need some additional functionality to ignore the valid moves check, perhaps by a boolean *ignore_valid_moves* passed into the *movePiece()* method in *Board*. Then if the move was a capturing move, we can undo the capture by placing the piece that was captured onto the ending coordinate. If the move was a promotion, we can demote the promoted piece back to a pawn. If we want to undo an unlimited number of times, we just repeat this process.

However a human player should probably not be able to undo the moves of a human opponent. The response to this question is unchanged from our original design document.

3. *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.*

- Since the board in four-handed chess is different, we would have to modify the board layout in *Board* so that it is of the correct shape. One way this can be done is by making the rows in *board_layout* have different length.
- Due to the changes in the board, the bounds checking of the *Board::movePiece()* method also needs to be changed for the special case of castling, as the coordinates are currently hard-coded for that of a standard 8x8 board.
- We would also need to support more than 2 players. We can naturally use the *Player* class to support multiple players where each can be different levels of computer or a human. There are also different game modes in four-handed chess (e.g, teams and free-for-all), so we would have to indicate which players are on the same team and modify the game logic accordingly.
- Four-handed chess also has different game logic than regular chess. So we would need to modify the game logic which can be done in the concrete *pieces*.
- Additionally, the graphical output would have to be adjusted as the window size and screen layout are designed with an 8x8 chess board in mind. However, this would simply amount to rearranging some elements on the screen and making the window size dynamic.

The response to this questions is largely unchanged from our original design document, save for the addition of the discussion of the graphical output, and an alteration of the second bullet point discussing bounds checking in *Board::movePiece()*.

Extra Credit Features

We have written the program in such a way that we do not explicitly manage the memory ourselves. All pointer fields representing an ownership relationship are implemented using C++ unique_ptrs, and as a result the program contains no delete statements without leaking memory.

Final Questions

1. *What lessons did this project teach you about developing software in teams?*

One aspect of this project that was particularly helpful was the split between Due Date 2 and Due Date 3. As the "plan of attack" was due well in advance of the final project, we were forced to spend a significant amount of time seriously considering different designs for the program, and the advantages and disadvantages of each. This allowed us to have a better understanding of whatever the other partner was working on at any given point, as well as of the overall structure of the program in general. Additionally, the fact that the design was solidified before we began programming allowed us to write code quicker, as we did not have to consult each other as much about design choices in the middle of programming. This was particularly helpful as the timespan between Due Date 2 and Due Date 3 was relatively short, so to be delayed by design discussions would have been especially detrimental. The UML that we submitted was also a great resource for quickly checking the structure of the program when needed. In general, the main thing that we took away from this project is the importance of planning before programming.

2. *What would you have done differently if you had the chance to start over?*

One shortcoming in our design is the implementation of computer opponents. There is much shared behaviour between the different levels of difficulty: for example, Level2 takes the behaviour of Level1 but prefers capturing moves and checks, and Level3 builds on Level2 by avoiding capture. However as each difficulty level is implemented as its own separate class, there is much code repetition between the difficulty levels. This could have been alleviated by using a different structure for the computer players. For example, we could have set up an inheritance hierarchy where Level3 inherits from Level2 and Level2 inherits from Level1, such that any "Level1 behaviour" can be implemented there, and then "Level2 behaviour" can be implemented by first calling Level1::playMove() function, and then built upon with additional logic. We could have also used the decorator design pattern, where certain behaviours such as "avoid capture" and "prefer checks" were decorators for some "computer player" component.

Another area of improvement relates to the access to the 2D vector of Pieces owned by the Board object. With our current design, when iterating through every possible Piece on the board, we have to use a 2D for loop, going row by row in the 2D vector and then space by space for each row. As this type of access is quite common throughout the program, there are many 2D for loops throughout our code. This decreases readability and introduces the opportunity for mistakes. If given the opportunity to start over, we would instead provide an iterator for the Board class for ease of use.