CS 246 Chess - Brian, Ryan, Harish

Introduction

This project implements the game chess in C++ using object-oriented design principles and proper design patterns taught in the CS246 course.

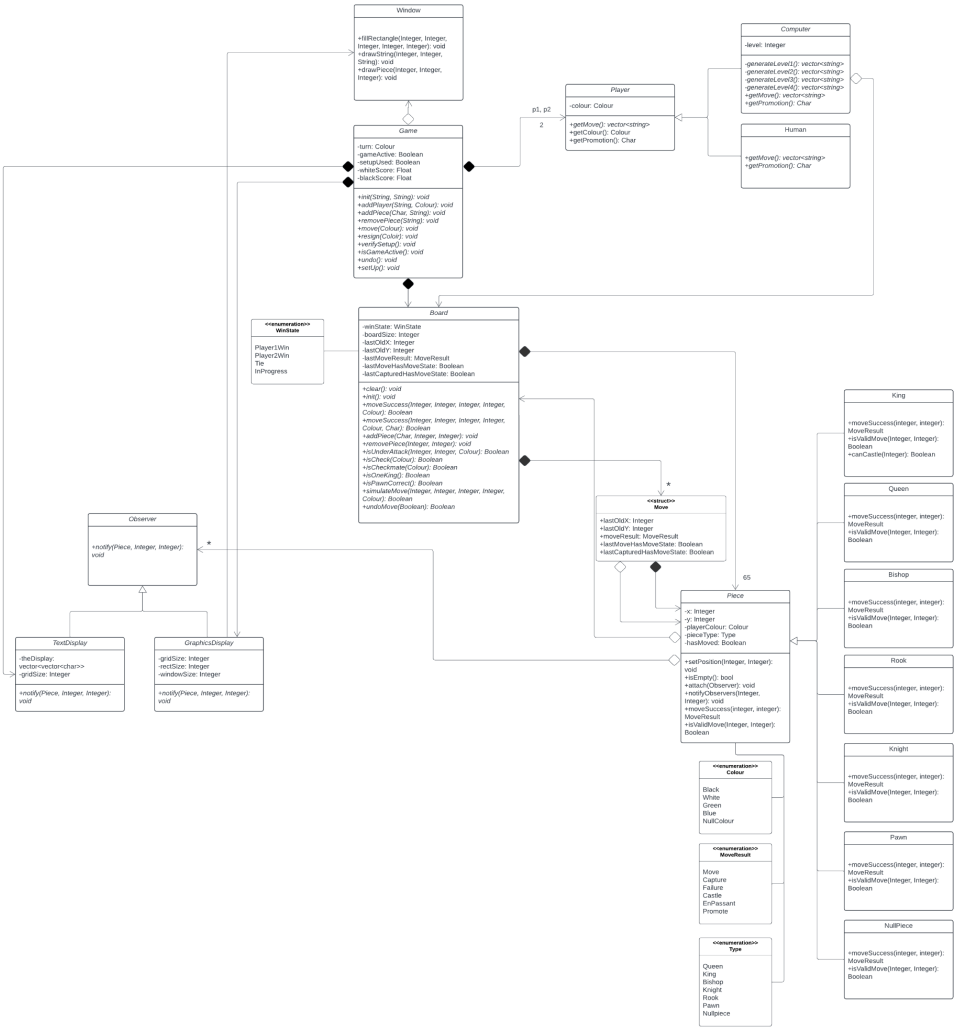
At its core, this chess game allows players to engage in the traditional chess experience with the added flexibility of choosing between human and computer opponents. The computer opponents themselves can range in difficulty from levels 1 to 4, providing a scalable challenge for players. Additionally, the project stands out for its dual-display capability, featuring both a Text display for command line outputs and a Graphics Display using X11 graphics for a visual representation of the chessboard.

The design philosophy of the project is rooted in key software engineering principles emphasizing the use of the observer pattern for efficient updates to the game displays, and a clear distinction between abstract and concrete classes to maintain an organized and modular structure. The implementation also showcases smart memory management and a focus on low coupling and high cohesion within the codebase.

Overview

Main: Takes input from the user and interacts with a Game object.
Game: Determines what to do with user input. Creates new Board objects whenever a new game of chess starts.
Board: Handles a single game of Chess. Owns all the pieces on the board, handles checks for checkmate, check, stalemate.
Piece: Abstract Piece class. Also acts as a Subject (there is no abstract Subject class as Piece is the only type of subject).
King, Queen, Bishop, Rook, Knight, Pawn, NullPiece: Inherits Piece, concrete classes for each type of chess piece (NullPiece is for blank spaces on the board).
Observer: Abstract observer class for the observer pattern
TextDisplay: Inherits from Observer. Handles the command line outputs for the program. Will output error messages, the current chessboard, and any other text the user needs to see.
GraphicsDisplay: Inherits from Observer. Displays current state of Chessboard using X11 graphics.
Window: Simplifies using X11 graphics. Main will create an Xwindow and pass a reference to other parts of the program like GraphicsDisplay.
Player: Abstract class for players of the game. Player objects will get the next move for the game.
Human: Inherits from Player. Gets the next move from stdin.

Computer: Inherits from Player. When the user specifies it wants a computer (of level 1-4) to play in the chess game, this object will be used. Based on the level, it calls a private method to generate the next move.



Design

Firstly, to handle updating the text and graphic displays, we used the observer pattern, where the different displays were subclasses of an abstract observer class, which attaches to the pieces on our board. When a piece is moved, it notifies its observers so that only the changed parts of the game are updated. This is especially convenient for the graphical display, as it minimizes the time needed to redraw the board with the new state of the game.

In addition, the notion of abstract and concrete classes was also an important component of our design. In our game, there are different types of observers, pieces, and players. The cleanest way to implement the different types of each was to make the Observer, Piece, and Player classes be abstract, and then make each type a concrete subclass.

There were also many times in development where we needed arbitrary length arrays. Whether it be in our Board class, or for parts of our implementation of the Computer class. Using vectors was essential to emulate that effect.

Next, encapsulation is enforced across our program. Each class has its own private/protected members, with public methods that can be used to access and/or modify them. One example of this is in our Computer class. When a Computer object is initialized, its level is specified, which is a private integer of Computer. Then, when the public method getMove() is called, it will call a private method based on what the level was initialized as, to generate the correct level of move.

Dynamic memory is used throughout our program. One of the most common uses is for our vector of vector of Piece pointers, which changes with every move that is made. In all cases of pointer ownership, we were able to leverage STL smart pointers. This made freeing memory very straightforward. Some other classes often needed certain pointers, but didn't have ownership of the object, so for this we had to use raw pointers.

Finally, our program was designed with low coupling and high cohesion in mind. To ensure low coupling, friends were only used for overloading the output operator, and for all other communication between classes, we made sure to use the public methods of the object a class wants to communicate with. To ensure high cohesion, we made sure that the components of each class were all related to one responsibility, thus heeding to the Single Responsibility Principle (SRP). For instance, Piece classes handle the state of a single individual Piece, Player classes handle getting the moves for a given player, Game handles the state of an entire session of playing chess, and Board handles the state of one instance of chess.

Design Differences Compared to DD1

Now, as for design differences compared to DD1, one of the major changes done to most classes throughout the development process was the addition of many more methods than we initially thought would be required. For example, initially our board was only used to add/remove pieces, check if a move was successful, and check if the game was in a state to finish. However, by DD2, there are now many more methods including methods to determine if the game is in a checkmated state, a check state, and a stalemate state, among many other methods.

Furthermore, another major change we made is combining the 4 separate Computer classes into one class, and using a level field inside the class to determine what type of move to generate. This made more sense because they were all computers, and we could differentiate them with one field instead of 4 different classes.

Also, we moved the enum declaration for Colour into Piece, and added an enum for both the Type of piece and the MoveResult of a move to accommodate different move types. This did not affect much during our development, since we included enum declarations in headers, and thus if we wanted to use them we only had to include the header for piece.

Resilience to Change

Our program supports changes in many areas:

1. Adding new pieces, since we developed pieces with the idea of using Piece as an abstract class, and having individual pieces (e.g: a knight) be a subclass of Piece. The game logic makes use of Piece's virtual methods through pointers. Thus, when adding a new piece, all it takes is to change the logic of the piece's moveSuccess() and isValidMove() methods, as well as any fields particular to the new piece one wants to add. Furthermore, these pieces have lower coupling with the rest of the program. When modifying a piece, the only file that needs to be recompiled is the file of that piece itself, and the linking process. The pieces only pass results with functions. However, at times they do affect the control flow of the Board (for example, if a piece's getMove() method returns a failed move).
2. Adding new player types, since we also made Player an abstract class, with virtual methods getMove() and getPromotion() (in the case where extra input/output was needed due to pawn promotion). Thus, when adding a new player, one can just make a new class and implement those two methods along with any other fields or private methods that it might need. Furthermore, if one were to implement computer levels 5+, they would only need to add a function (e.g: generateLevel1(), generateLevel2(), etc…) to generate a vector of strings (the start move and the end move) with whatever algorithm their computer level uses, as well as modifying getPromotion() (in case they do not want to auto promote their pawns to queens).
3. Modifying the TextDisplay and/or GraphicsDisplay. These two classes utilize the Observer pattern, and have low coupling with Board and Game (The two classes that control most of our game), since they do not depend on Board and Game, and Board and Game do not depend on them. Furthermore, data is passed between the displays and the game controllers using basic parameters (results), through the notify() function, where the Piece to be updated is passed, meaning low coupling. Thus, since they have low coupling, the way the graphics are drawn and the text is displayed can be easily changed without affecting the rest of the program (when changing TextDisplay or GraphicsDisplay, only their respective .cc files need to be recompiled, with linking being the only other step.
4. Instead of hardcoding the number of Colours and Players, we have the Colours in an enum that can be easily modified to accommodate more colours. The players are also in a vector, in case one wanted to accommodate 4-player Chess. The use of vectors allows many features to be resized (since a vector is essentially a resizable array). In addition, we tried to limit the amount of times we hardcoded the constant for board size (the only cases where this was necessary is if a board had not been declared yet or a class did not have access to board), but this could be fixable with a global constant variable. Thus, our program can accommodate different board sizes, though it would require some changes to areas where the value is hardcoded. Furthermore, along with different board sizes, we wrote a "convertCoordinates()" function for both int, int -> vector<string> and vice-versa, allowing the system to accommodate different coordinate systems by just changing two functions.

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.

DD1 Response:
Data structure for storing openings
The first step is to decide how to store the opening sequences. An n-ary tree or graph structure is ideal because chess openings are essentially a branching set of possibilities. Each node in the tree represents a board state, and each edge represents a move. For example:
The root node represents the initial board state.
The children of the root represent all possible moves for the first turn (each node contains a pointer to a next node, a string representing the next move, and an int representing the win percentage).
Each subsequent layer represents the board states after each subsequent move.
Integrating with Game Logic
The opening book needs to be integrated into the game's logic:
When a game starts, the program should first check the opening book.
As moves are made, the program follows the path down the tree corresponding to those moves.
If the current game state matches a node in the tree, the program can use the children of that node to suggest or make the next move.
If the game deviates from the book (i.e., reaches a state not in the tree), the program switches to its standard move-making algorithm and creates a new node as a child of the previous node with the new move.
Choosing openings
For choosing the best opening, the program can:
Randomly select an opening to add variety.
Use statistics (win percentage) to choose the opening with the best historical success rate
User Interface
To accommodate the UI, we can extend the XWindow display and display the list of moves on the right side of the game board. The list of moves will be recalculated after every move through by traversing down the tree and obtaining its list of children nodes in the next level.

Changes:
This feature would be implemented the same way if we were to do it with our final code.

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

DD1 Response:

To undo one move, a struct could be implemented as a field for a Player, that stores two parallel vectors, one that holds the old state of Pieces that were changed by the move (Piece type, old coordinate), one that holds the new state of Pieces that were changed by the move, as well as integer to represent who's turn it is. A new user input can then be implemented that accepts "undo", and when it is used, the program will check if that player has a previous move defined. If the player does, it will go through the vector of Pieces that changed after the move, and replace them with NullPieces, and then add the pieces in the old state vector to the board.
For unlimited undos, the above idea can be extended to a vector that stores these structures. This vector will work like a stack. Every time undo is called, it will pop from the back of it, and every time a new move is done, it will emplace to the back of it.

Changes:
While we did implement a struct as a field, instead of using parallel vectors, we directly used a stack to store the struct, the struct is called Move and contains a list of private fields to store the last move's old x y coordinates, new x y coordinates, a pointer to the piece that was moved, a unique pointer to store the piece that was captured (for move, the captured piece would be the null piece), the type of move (Move, Capture, Castling, etc…), as well as their hasMoved states before the move.
The stack approach is particularly advantageous for several reasons:

1. Ease of Access: The most recent move is always at the top of the stack, making it straightforward to access and reverse the last action.
2. Efficiency in Undo Operations: Undoing a move simply involves popping the top element from the stack and reverting the game state accordingly. This method is efficient and clean, as it avoids the need to search through a collection of moves.
3. Support for Unlimited Undos: The stack can store an indefinite number of moves, enabling the feature of unlimited undos. This is achieved by continually pushing new moves onto the stack and popping them off for each undo action.

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.

DD1 Response:
To extend the game to four players, the Game class would now have 4 Player objects instead of 2. Similarly, when taking input from the user, it will now loop through all four players. Since our Piece class already has a variety of colours for each player, we can choose two colours other than black and white for the new players. To accommodate for the larger board size, the vector of vectors of Pieces will be increased accordingly. The blank corners will be initialized as NullPieces, but to ensure that no real Pieces move there, the calculation of whether a move is valid or not will now consider the additional boundaries (there will be a vector of vectors of Booleans, that is of the same size as the Board. If a Piece can move there, the boolean will be true in that spot). When a round of chess concludes, the Board will now return the Colour of the player who won, so that Game knows which Player to give the point to.

In terms of keeping track of the score, since 4 player chess is based on points earned, the score in the Player class can be used to represent the points the player earned instead. Since our current move function returns a boolean on whether the move was successful, we could modify it in a way such that it returns a state instead (the different states would be in an enumeration class) representing the amount of pointers a player gets for their moves, if any (capture, checkmate, stalemate, etc).

Changes:
Much of the approach remains the same, though there would be a few changes. Firstly, score is no longer kept in Player classes, as different Players can be set up for each new game. Thus, we would instead have to add more variables to represent the scores of the additional players. Some code would also need to be added to accommodate for the new boundaries, and also to print and display the board correctly.

Extra Credit Features

Firstly, we completed the project with no memory leaks and without explicitly managing memory. We used STL smart pointers wherever possible, only using raw pointers when needed for non-ownership, and thus there are no delete statements anywhere in the program. From the start of development, our team had the goal of implementing this feature. With the UML made for due date 1, we had a good idea of where to use smart pointers and where raw pointers had to be used, making this feature fairly straightforward to implement.

Another feature that we implemented is the ability to undo as many moves as possible. This feature was fairly complicated to implement due to the different types of moves that can be made. We started with implementing a single undo, that just kept track of the most recent move. We added a list of private fields to store the last move's old x y coordinates, new x y coordinates, a pointer to the piece that was moved, a unique pointer to store the piece that was captured (for move, the captured piece would be the null piece), the type of move (Move, Capture, Castling, etc…), as well as their hasMoved states before the move. Then to implement the undo, after every move, we first updated the fields and moved the captured piece using std::move since it is a unique pointer. Afterwards, the undo function will check the type of move and revert the changes as needed. Then to implement unlimited undos, we defined a struck called Move using all the fields needed for a single undo move and a stack of <Move>, the stack will record the history of moves, and the last move fields will be for just the last move, every time when an undo is called, we will revert the changes, and then update the fields by called stack.top() and then popping it from the stack. We also added an optimization such that every time a player makes a move, we will check if the last move is Failure or not, if it is a Failure, then we will not push it in the stack and set last move's type to Failure. This way the stack will only contain successful moves. To accommodate this optimization, in the undo, we wil first check the last move's type, if it is Failure, then we will retrieve the last successful move from the stack (since this will guarantee the move is success), otherwise if the stack is empty, we will return

false since there is no successful moves made. This will also guarantee no memory errors, as if a move is unsuccessful, then we will not be transferring the ownership of a piece (which is a unique pointer) from board to the move history stack.

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 was a great opportunity for the team to get familiar with working in Git with a small group. We quickly learned that the best way for us to develop was to assign specific methods/classes to each group member to develop separately, and commit our individual changes to build up the program together. For example, for the different subclasses of Piece, each group member developed a different one, and then we merged our code together to have all the completed Piece subclasses.
Along with this was the importance of communicating. As development ramped up and got more complicated, the best way for everyone to stay on top of changes was to talk about the logic behind our code, and exchange ideas to help each other solve problems. One example of this was in developing the different levels of Computers. We originally had issues with our algorithm modifying the display, but with the help of the entire team, we were able to pinpoint the issue and solve it. It turned out that looking at each other's code was very helpful.

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've better defined the interfaces for our classes at the beginning, including adding more helper functions for code that we would potentially reuse at a later time, as this would tidy up the code much more.

On top of that, we would've begun by developing the testing harness, beginning with the game, setup, and move commands, as those were the most integral to testing our program, instead of developing the pieces first. We thought this would be a good idea since pieces were an isolated part of the program, but it would've been much more efficient to have a working board and testing harness, and seeing if a single piece worked properly before implementing the others, as most of the pieces were able to reuse code from the other pieces.

Next, we would've converted some of our manual testing to automated testing (or close enough to it) as we were developing. A lot of our testing was done using our setup/move commands manually, but we could've added those into .in files to automate the process of typing in the commands every time we wanted to test a specific feature. Though there was no sample executable to generate proper outputs, having tests that could write to our game's standard input would've made the testing process much smoother.

Finally, we would've set a style guide that all of us would follow, including specifications on tab size, comments, indentation, etc. Since we didn't establish a style guide at the start of the project, we spent more time than necessary cleaning up and refactoring code at the end that could've been spent implementing extra features.

Conclusion

Our chess project has made use of different design patterns learned throughout the CS246 course, particularly the observer pattern. It also includes solid object-oriented design, utilizing encapsulation, single-responsibility principle, and other techniques as discussed beforehand. Though there were many challenges to working on a project of this scope, it was a good introduction to developing large software projects in a team and a helpful learning experience on all aspects of C++.