design.pdf by ec2li, h25lu, and s23hu

1. Introduction

This is our Fall 2023 CS246 final project: Chess. This Chess Game Project in C++ aims to implement the classic chess game where two players compete on the classic 8x8 board. It serves as a demonstration of fundamental C++ programming concepts, including object-oriented principles, modular design, design patterns, and algorithmic thinking.

Here are some key features of our project:

- Full implementation of standard chess rules, including pawn promotion, castling, and en passant.
- Simple console-based and window-based user interface for easy interaction.
- Object-oriented design with classes for pieces, players, and the game board.
- Modular code structure promotes maintainability and extensibility.

How to Play:

- Players take turns making moves by specifying starting and destination coordinates.
- The game checks the validity of the move and updates the board accordingly.
- The game continues until one player achieves checkmate or a draw is declared.

Summary of Implementations:

- The chess game is implemented using C++ classes for pieces, cells, players, and the game board.
- Validation of moves, checkmate conditions, and other game rules are enforced through carefully designed algorithms.
- The modular structure allows for easy extension and modification of game features.

2. Overview

In terms of structuring, our project is mainly separated into 4 parts.

Part 1: Pieces

- Files included: piece.(h|cc), bishop.(h|cc), king.(h|cc), knight.(h|cc), pawn.(h|cc), queen.(h|cc), rook.(h|cc).
- PieceType and TeamColor classes (piece.(h|cc)) are enum classes to better style the representation of types of pieces, and the piece belongs to which player.
- Piece class (piece.(h|cc)) is an abstract superclass for all the different kinds of pieces. It also contains some helpful public methods that work on every type of piece.

- Every piece has its position in the grid, its team colour, and a vector of Piece pointers to all other Pieces on the grid. The vector of Piece pointers is protected so that every subclass of Piece can access it.
- Methods provided in Piece and every method does exactly what the name suggests:
    - Accessors: getTeam(), getPosition(), getPieceType(), getPieces()
    - Mutators: setTeam(TeamColor), setUpPieces(vector<Piece *>), removePiece(Piece *), move(int, int)
    - Methods to calculate the current status of the piece: **getAllMoves()** which returns all the possible legal moves the piece can make, **getAllCheckMoves()** which returns all the moves that will check the enemy king, **getAllCaptureMoves()** which returns all the moves that would capture an enemy piece, **isAttackEnemyKing()** returns true if the piece is currently checking the enemy king and false otherwise, and **enemyKingPos()** which returns the position of the enemy king. Those methods support the implementation of Cell, Board, and computer players.
    - Methods: getAllMoves and getAllCaptureMoves() are fully virtual, and getPieceType(), getAllCheckMoves(), isAttackEnemyKing(), enemyKingPos(), isChecked() are virtual methods with an implementation in piece.cc
- All subclasses of Piece (*.(h|cc) except for piece.(h|cc) stated above) represent one type of piece.
- Methods of each kind of piece are overriding the virtual methods in Piece based on the logistics of that kind of piece. Only Pawn has extra public methods (to support en passant and double move).
- We can get the Piece type using getPieceType() override which returns the type of the Piece.
- Every piece has its different implementation of :
    - getAllMoves() which finds all the possible moves that a piece can make depending on all the other pieces on the board. The implementation of these methods is completely different from piece to piece.
    - getAllCaptureMoves() which finds all the moves that capture an enemy piece.
    - getPieceType() which returns the type of the piece in question.
    - Ctor
- Every piece also has an implementation of other virtual methods where it would be more efficient to do so or the default implementation in piece.cc does not work for that piece. This does not change the behaviour of the method, only the implementation.
- Some pieces may also have more fields like for example a pawn that has a bool doubleMove that checks if that pawn can move forward 2 spots and a bool canBeEnPassant that checks if a pawn can be eaten by En Passant. Pawn would also have an accessor and a mutator for each of its extra fields.

Part 2: Players

- Files included: player.(h|cc), human.(h|cc), computer.(h|cc), level1.(h|cc), level2.(h|cc), level3.(h|cc), level4.(h|cc).
- Player class (player. (h|cc)) is an abstract superclass for different kinds of players and it has methods:

    - Accessors: getTurn(), getTeam()
    - Mutators: setTurn(bool), setTeam(TeamColor)
    - Virtual abstract method turn(), which is called during a game to represent a turn of a player. For computers, turn() moves a piece. It has to be a valid move. Every level of computer has a different algorithm to determine the move it will do.
- Human class (human. (h|cc)) is a subclass of Player that represents a human player. This class has no functionality now, but it is here to organize the code better and to facilitate resilience to changes.
- Level1 - Level4 classes are subclasses of Player that represent 4 levels of computers. Each Level only overrides the turn() method to make a computer move based on the difficulty clarified in the documentation of project Chess. To do this, the Piece class has added many interface functions, such as all possible moves of a Piece. It uses getAllMoves(), getAllCaptureMoves(), getAllCheckMoves() to determine a valid move to make. All of those methods are in Pieces.
- Level4 added priority capturing: prefer capturing the queen over the rook, which is over the bishop, which is over the knight, which is over pawn.

Part 3: Displays

- Files included: observer.h, graphicsDisplay.(h|cc), textDisplay.(h|cc), window.(h|cc).
- This part uses the observer design pattern with the Observer class (observer.h) being an abstract superclass for observers. Observer provides the Virtual abstract method notify(Cell &). The classes being observed can call it.
- XWindow class (window. (h|cc) is a utility class for the X11 library. It is the same as given in Assignment 4: a window is created when an Xwindow object is created, and we can add strings and fill rectangles with colours within the window.

- GraphicsDisplay (graphicsDisplay.(h|cc) is an interface observer class for displaying this chess game using XWindow.
- Methods provided in GraphicsDisplay:
    - constructor GraphicsDisplay(int): It takes in the window size we want the display window to be. It initializes the default chessboard and displays it.
    - notify(Cell &): It overrides the superclass notify method and redisplays that part according to the content of Cell.
- TextDisplay (textDisplay.(h|cc) is an observer class designed for displaying the current game status as text to the terminal. Its methods do similar things as GraphicsDisplay, but display it in the terminal with char instead of as a graphical interface.
- Methods provided in TextDisplay:
    - constructor TextDisplay(): It initializes the default board as a 2D vector of chars.

    - notify(Cell &): It overrides the superclass notify method, and modifies the 2D vector of chars to contain the correct status of the current game.

Part 4: The game itself

- Files included: board. (h|cc), cell. (h|cc), main.cc.
- GridColor class (cell. (h|cc)) is an enum class to better style the colour of grids in a board.
- Cell class (cell. (h|cc)) is a class that represents each cell in a board.
- It contains a pointer to the Piece in this cell, its row and column number, its GridColor, and a vector of observers of this cell.
- Methods provided in Cell:
    - Constructor Cell(int, int): It initializes a cell with parameters being row and col to contain a nullptr to a Piece and an empty vector of pointer to observers.
    - Destructor ~Cell(): It deletes the piece contained in this Cell.
    - Accessors: getGridColor(), getRow(), getCol(), getPiece(), getPieceType().
    - Mutators: setGridColor(GridColor), setPiece(Piece *), detachPiece().
    - Methods to support observer pattern: attach(Observer *), notifyObservers().
    - promote(PieceType) to promote the Pawn in this Cell to PieceType.
- Board class (board. (h|cc)) is a class that represents the chess board.
- It contains an 8x8 vector of Cells, TextDisplay and GraphicsDisplay of this game, and the current turn of this game.
- It uses a controller pattern to connect Cells(Pieces), Players, Displays, and the game execution.
- Methods:
    - Constructor Board(int): It initializes the TextDisplay and GraphicsDisplay with window size as the parameter of this constructor.
    - Destructor ~Board(): It destroys all Cells and displays.
    - Accessor getPiece(int, int).
    - Mutators: toggleTurn(), setPiece(int, int, Piece *).
    - Methods to initiate the board: init() to initiate all the Cells to make an empty board, and initDefault() to initiate all the Cells to make a default chess board.
    - Methods to get the current status of the Board: isCheck(), isCheckmate(), isStalemate(), getWinner(), getCheckedPlayer(), getAllWhitePieces(), getAllBlackPieces(), getAllKingPieces(), getAllPieces().
    - Methods to modify the status of the board: move(vector<int>, vector<int>) which takes in the starting/ending position vectors (of size 2, representing row and col) and moves the piece from the starting position to the ending position, resign() to output resign message, and promote(int, int, char) which promotes the pawn at the given position to the kind of piece represented by the char parameter.

    - operator<< to support the TextDisplay of the Board.
- Main procedure (main.cc) which handles input and output by the user. It will receive the input from the user and act accordingly. It is responsible for calling each class's corresponding method depending on the input of the user. It starts by creating a board and User Interface with the 2 display class (part 3). It also runs the games using the classes mentioned above (Part 1 and Part 4). It also manages the AI using methods from (part 2).

3. Design

Problem 1: How to get the current game status and display it efficiently?
Solution: Adopting the observer pattern, we designed TextDisplay and GraphicsDisplay classes to be observers, observing all the Cells on a Board. Thus, every time a Cell in the Board is modified, it notifies both Displays so that both Displays can make changes to display the current game status properly and efficiently.

Problem 2: How to handle different types of Pieces in a game, without inefficiently enumerating all cases?
Solution: We adopted Polymorphism when designing Pieces. We designed Piece to be an abstract superclass which contains multiple virtual methods. Then, for each type of Piece, we created a new class inheriting Piece and overrode those virtual methods to be compatible with this Piece type. When creating a new Piece, we allocate heap memory for it and store the pointer to it as a pointer to a Piece. Thus, when using those Pieces, we could just call the piece->doSomething() method without having to know the type of the Pieces.

Problem 3: How to interact with users (input/output) in a manner that facilitates resilience of code?
Solution: Initially, we were planning to fully use the MVC(Model-View-Controller) pattern: handle inputs and outputs in Player classes, process data (interacting Pieces) in Board/Cell/Piece classes, and display data in Observer classes. However, when we were trying to handle the parameters being passed to this program, we found it messy and inefficient to handle inputs and outputs in the Player classes. Thus, we modified our program to handle inputs and most outputs in main.cc and use the Player classes to handle moves (especially for computer Players). Because of the possibility of promoting and casting, it would also be messy and inefficient to implement a human move in the Human class, we are handling human moves in main.cc. However, the Human class is still there to support the resilience to new commands (especially when we need to make an auto move for humans, such as hint or "move for me"). All in all, we partly adopted the MVC pattern: we used Players to handle auto moves, Board to handle data(Pieces), and TextDisplay and GraphicsDisplay to handle views.

4. Resilience to Change

Low coupling:

Low coupling is achieved in our program. For example, Piece and its subclasses do not rely on other classes. This gives us flexibility and so changes to the pieces affect the rest in minimal amounts and vice versa.

High cohesion:

High cohesion is achieved in our program. We only have one class per file and that class has a unique goal. Each class represents a crucial component that builds up the chess game, e.g. piece, cell, board, displays, etc. High cohesion makes change more localized and leads to better maintainability and reusability.

If move rules for a piece change:

We only need to change the getAllMoves() function in the class for that piece so that it supports the new rules. No changes would be made to codes in other classes.

If adding new types of Pieces (changing how some Pieces would behave):

We have adopted polymorphism when processing interactions between pieces. Thus, if we want to support new types of Pieces, we could just add a new class of that Piece type to inherit Piece, override all virtual methods in Piece in that class, add this enum possibility to PieceType, and then support setting up this type of Piece. Similarly, no changes would be made to codes in other classes.

If the algorithm for computer move changes (or another level):

Since we have Levels as Players and use polymorphism (call Player->turn()) when it is a computer move, we can simply create a Leveln class which only needs to override the turn() function to make a computer move. Then, we add the initialization option for computer[n] and we are done!

Adding supports for Players (supporting different commands)

We can add support for Players such as opening book sequences and displaying all possible moves. We can add methods on the board to display those supports for Players if Players indicate so through input, and add only if-else's processing commands in main to handle those inputs.

Adding different Display methods (changing ways of Displays)

Since we are using the observer pattern for Displays, we can simply add a new class for this new display method, implement a notify(Cell &) method to modify the state of this display, attach the object of this Display to each Cell as an observer, and then add the display command whenever needed. In this way, we can display the correct information in this new method anytime we want.

5. Answers to Questions

Question 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.
See for example: https://www.chess.com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

We would implement a book of standard opening move sequences using a tree-like class. Each node contains the move, the win rate of the move in this situation, and an array of pointers to other nodes. The root node would be the starting position. The children of the root node are the best possible moves that can be played by White in the starting position. The children of those nodes are the best moves Black can play after its parent has been played. The tree would continue this way until it ends. The tree ends when we reach a leaf node which is a node where the opening we were playing ends or when one player plays a move that is not optimal, meaning that it is not in the children of the move before it. To implement this book of standard opening move sequences, we would need to get data from online sources. We would also need to create a program that can convert the data we have to the structure we want it to be.

We can store such a tree-like class in the game Board. The way our book of standard opening move sequences would work is that we start at the starting position. Our program would show all the children of the root node with its win rate. When we do one move that is included in the best moves, our current node moves to that node and the program would show the children of the current node. When we reach the end of the node, our current node becomes a null pointer so we no longer show any best moves, but rather have a free board.

Since we would have the data, we can also add it to our AI so that it can play the move with the highest win rate in a given position assuming that the position is in the tree. So we would have a current node and after each move, the current node would move to the next node corresponding to the move played or null pointer which would result in the AI stopping to resort to the book of moves.

Question 2:

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

We would make a game (main.cc) containing a stack of boards instead of a single board. Starting from pushing the initial board to the stack, each time a player makes a move, we create a new corresponding board and then push it to the top of the stack. Thus, the board at the top of the stack represents the current state of the game.

If a player wants to make an undo, then we pop the top of the stack. Thus, the current state of the game is represented by the board that the player hadn't made the move yet, and thus a redo could be done. Multiple undos can also be done by popping multiple boards from the stack until the initial board is reached.

The above method is easy to understand but is quite inefficient in terms of both time and memory. Thus, we could also create a new object class, called Move. A Move stores the index of the piece that was moved (which means that we also need to modify the piece class), the x, y values of the move, and also a pieceType which may have been captured in this move. This way, we can make a board containing a stack of Moves. Each time a player makes a move, we store the information of the move to the Move object and then push it on top of the stack.

If a player wants to undo their last move, we pop the move on top of the stack and then resume the board according to the move. A multiple time of undos can also be done by popping multiple moves from the stack until the stack is empty.

Question 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.

First, we need to change the board size to fit a four-handed chess game (change Board init()) and would need two more players for a game(change enum class TeamColor). Promotion mechanisms would also change accordingly. A player can be eliminated by being checkmated, stalemated, or resigning, and the game ends when three players are eliminated(e.g. move isCheckmate() to player class), or, in a special case when two players remain and one is leading by at least 21 points. Then, it is necessary to keep track of the point for each player (a new variable) and the player with the highest point would win the game (change isWon()).

You can earn points by:

- By checkmating an opponent (+20);
- By stalemating oneself (+20);
- By stalemating an opponent (+10 for each player still in the game);
- By checking more than one king simultaneously with a queen (+1 for two kings, +5 for three);
- By checking more than one king simultaneously with a piece other than a queen (+5 for two kings, +20 for three);
- By capturing active pieces (+1 for a pawn or promoted queen, +3 for a knight, +5 for a bishop, +5 for a rook, and +9 for a queen).
- If the game is drawn by threefold repetition, insufficient material, or the 50-move rule, all active players receive 10 points each.

This requires extensive alterations to the Player class to check for those things. Once a player is checkmate or stalemate, all his pieces become dead and capturing does not give any points (change turn() sequence and also points algorithm in Player class). If a player resigns, his king starts moving randomly (e.g. make Player a Computer object and remove all its other pieces) when it's his turn. If we are to allow a team setting, a team class would need to be created with players as its subclasses, and the game status-checking algorithms of the Board are to be updated as well. Note that there are TONS of other minor changes that need to be made to facilitate four-player chess, and it is impossible to list them all here.

6. Extra Credit Features.

Default Board

We can run the chess program with a parameter: ./chess -defaultBoard. If we use this parameter, every time the board is initialized or reinitialized (after a game), the board would be set to a default chess board (each team has 16 pieces)

7. Final Questions

Question 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?

Firstly, it is extremely important to clarify the specifications of work assigned to each group member. During our development process, because we were not clear about how each function should behave, some behaviours of our functions were not what we expected them to do. We would use a certain incorrectly which leads to a large amount of bugs. This has largely decreased our working efficiency as well as the cleanliness of our code because we have to reimplement our classes and methods. If we had specified the requirements of each function in detail before we started implementing it, we should have avoided those deficiencies. This could be done by writing a good description of all of our functions above its declaration.

Secondly, we learned that communication during the development process is also crucial for the team's success. No matter how well we did the specification process at the start of this project, there should still be confusion since it is almost impossible for us to get the specification perfect on the first try. For all the confusion, if we had better communication with each other, we would ensure that our code is in the right way as it should go. As the first point, communication would also reduce useless work and make the integration of the whole project easier. This is also why we implemented the end of the project when we were always on the same Discord call.

Thirdly, we should always accommodate the time in case unexpected things happen. In this group teaming process, we were unaware that the penalty for missing in-detailed specifications was so severe that we had to work all night a few days right before this project's due. This is probably partially due to the scale of this project being larger than anything we've worked on before and the fact that working as a team is slower than expected since we were often in situations where we were fixed code written by someone else. If we had finished our implementations earlier, we could have made a more reasonable and healthy working schedule during the integration process.

Last but not least, we should test our code separately before merging all of our implementations. We do that to make sure our code is functioning before the integration process since it is harder to test the whole program than a small part of it. During this project, we did not do any debugging by ourselves before we integrated our program. This has made bugs harder to spot during the debug process after the integration of our program. We have spent more than 30 hours in total debugging our integrated program. If we debug our code before the integration process, the final debugging process would be way shorter and easier.

Question 2:
What would you have done differently if you had the chance to start over?

Firstly, we would have spent a lot more time discussing how we were going to implement the project and adding more specifications before starting so that everyone was on the same page when talking about the implementation. we would add comments as specifications of each method before we start implementing. This would make our project work similar to what we have done in former assignments, and files would be much easier and clearer to implement. Since we did not do that for this project, the integration process became way more troublesome because the specific designs of some classes do not support the actual functionalities those classes are supposed to implement.

Secondly, we should have communicated with other group members more often, ideally, we could have communicated on every possible confusion. This is more efficient than trying to find out how every function of every class should work on ourselves and assuming that the implementation of a certain function can do something not planned. We did not communicate often and we were all figuring out how each class should work by ourselves. This has not only made the implementation process more inefficient but also caused many errors during the implementation that we had to find and debug which took a very long time.

Thirdly, we should have started the project earlier to accommodate unexpected events. Since we were busy during the first week of the project (due to other courses), we did not work on this project that much. However, the amount of work needed for this project is a lot more than what we anticipated. This caused all of us to work on the project for more than 10 hours for the last 4 days. If we started the project earlier, we could have had a more reasonable working schedule for the last 4 days, and time to add extra features which we unfortunately do not have.

Additionally, we should all have created tests and test suites for our code before the final integration process. For example, we should have a test suite that tests every getallMoves(). We did not test our code before the integration process. Thus, after we wrote main.cc together, we found that there are plenty of errors that make the code not compilable. After hours of working, we finally compiled our code, but there were still a huge amount of logic errors in our code. As a result, the time we spent debugging is much longer than the time we spent implementing the code.

Finally, we could have created a better order of implementation instead of assigning all the work at one time. For example, we could have implemented main.cc first, so that each member could have used main.cc to facilitate testing our implementation without having to finish the project.