CS 246 - Object-Oriented Software Development
Assignment 5: Chess Project
Documentation
July 26th 2024
Group Members:

Patricia Yuan
August Tan
Elaine Ge

Introduction:

The primary task of this fifth and last assignment of CS246 - Object-Oriented Software Development course is to implement a game of chess using C++ with an emphasis of using the principle of object-oriented design techniques. This approach promotes modularity, reusability and maintainability during the development process of our fully functional chess game. In this document, we will provide a detailed outline of our design and implementation, highlighting our use of object-oriented techniques.

Overview:

- **Board**: Represents the chessboard and contains logic for moving pieces, checking conditions, and board state
- **Game**: Manages the overall game state, players, and scoring..
- **Player**: Abstract base class representing a player, with derived classes for human and computer players.
- **HumanPlayer**: Derived from Player, this class is in charge of handling moves from a human player.
- **ComputerPlayer**: Derived from Player, this class implements various levels of AI, including move generation and decision-making.
    - Level 1 is the simplest chess AI. It only generates random moves by picking random pieces according to the color it is given, and a random coordinate to move the targeted piece. If the move is valid based on the random coordinate and random piece, then it will be displayed on the board. Otherwise, AI will keep picking up the coordinate until the move is valid.
    - Level 2 focuses on the behavior of capture. The program will check all possible moves from the pieces according to the given color. If there is any piece that could be captured, the program will move to that coordinates. If there is no piece to be captured, it will use a level 1 program to generate random moves.
    - The Level 3 program is similar to the Level 2 program. It instead focuses on avoiding the capture. The program finds all possible moves that checks whether the opponent piece could capture the piece, then moves the piece to that

coordinate as the move is valid. Otherwise, the program will use a level 1 program to generate random moves.

- **Piece**: Abstract Base class for all chess pieces, with derived classes for specific pieces.
- **King, Queen, Rook, Knight, Bishop, Pawn**: Inherited from Piece, each of these classes is in charge of validating their moves and any special cases involving them

In a standard run of the program, a *Game* object is created which creates a *Board* object and two *Player* objects. Once the *Board* is created, the setup will create the *Piece* objects.

Design:

As mentioned, this project uses an object-oriented design. Many features of the object-oriented design are explained in other sections of this document. These features include

- **Encapsulation**: the program is largely modularised, see resilience to change
- **Inheritance**: the program uses inheritance to create objects of similar nature, see resilience to change
- **Polymorphism**: the program utilizes polymorphism to treat instances of objects to be treated as instances of their parent class. In the piece class, all of the *King*, *Queen*, *Rook*, *Knight*, *Bishop* and *Pawn* are treated as a *Piece* in other classes. This makes it significantly easier to control the pieces of the board.
- **Abstraction**: in the program, only methods that are used outside of the class itself are made public, otherwise, fields and methods are left private to prevent confusion and potential bugs.

Another significant point of our implementation is the use of unique pointers to manage dynamic memory, see extra credit features.

The structure of our program separates and groups common concerns and functions. The *Game* class handles game logic, acting as a referee in a real-person scenario. The *Board* class acts as the physical board, including moving the pieces, checking validity of the moves and setting up the board. The *Player* objects act as players, in charge of making the decisions for moves. The *Piece* objects act as pieces and will check over their moves.

We used vector, map and pair of the C++ Standard Template Library (STL), as containers as it is easier to keep information together without worrying about dynamic memory errors.

Resilience to Change:

<u>Encapsulation:</u>

Our program modularized the aspects of a chess game to make the program more resilient to change. In the case that something needs to be added or new features need to be added, it is easy to implement and test.

When changes are needed, this feature ensures that any unwanted change to the internal implementation is minimized. Through isolating change into its class and controlling its relation with external code, we can prevent changes from causing bugs to ripple through the code.

By modularizing the code, it makes it easier to understand and maintain the code by clearly separating classes and functions. By separating the interface from the implementation, we can reduce confusion from understanding implementation to understanding the functions' affect and return value.

Further, this makes it easier to debug and test. Since an object can only be changed through its public functions, it's easier to track where the state of it changes, which helps significantly in pinpointing bugs and verifying behavior.

This feature also promotes reusability. By having functions grouped together, it reduces the need to write separate functions at each instance, ensuring that we use already written methods over writing new methods.

Inheritance:

In this program, inheritance is used in *Player* and *Piece*, where *HumanPlayer* and *ComputerPlayer* are inherited from *Player*, and *King*, *Queen*, *Rook*, *Knight*, *Bishop* and *Pawn* are inherited from *Piece*.

By using inheritance, it makes it easier to add new players. By having a basic *Player* class, adding a new player only entails inheriting from *Player* and implementing the player logic from there. This reduces the need to change syntax with every implementation, and more focus can be put on the implementation. In the case that a new *Piece* needs to be added, a similar process can be followed, making it significantly easier to seamlessly implement any needed functions.

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. 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.

To implement a program that displays recommended moves for the first dozen moves of games, we can create a class that stores the piece being used, the coordinate being moved to, an int indicating the number of times that player used this move, three ints that represent the number of win, draw, and loss after the game is ended.

By doing so, we can have a vector of this object to record the white player's move and black player's move each time. After a few games are being played, we will be able to analyze through this vector of objects and sort them according to the win, draw, and lose rates. Then we are able to display those recommended moves to the players.

2. 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 use an object-oriented approach by creating an object that keeps track of the piece that the user uses in each move, an initial coordinate that indicates a piece's original coordinates before the move, and an ending coordinate after the move. A bool is set to check if a promotion is possible. If the promotion is indeed possible, then the promoted piece would be updated to the project. We also consider the situation where if one piece captures another piece, this will also be recorded in the string representing the captured piece.

As a result, we could create a vector of Move objects that records each move from the users. If the user wants to undo their last move, then the last move would be popped off. If the user keeps undoing, the process will be repeated until there are no further moves and can be deleted. Then, the user will be forced to stop the game but move instead. This could be done in our board.h and game.h, so we can directly use them in main.cc.

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. (If it's important to your answer, state whether you're assuming free-for-all or team rules and then answer the question. You don't need to get too specific into the rule set changes in answering the question though; your focus should be more on what would need to be altered at the high level of the design?

Based on [https://www.chess.com/terms/4-player-chess], it is clear that the main alteration would be to accommodate the way board.h interacts with 4 people versus 2 people, as well as the handling of players. In the initial two-player game, we are using bool to represent white and black players since it is binary. 4-player game disallowed this approach, and necessitates the usage of string to represent each player.

In particular, board.h needs to be altered slightly to accommodate a 14x14 grid instead of 8x8 for a 4-hands chess game. Handling this new layout includes initializing the board with the correct dimensions and updating the methods that manage piece placement and movement.

For player.h, the class should be able to handle multiple players instead of just 2. The rotation logic has to be changed to accommodate four players as well, since we are currently using boolean flags to determine the turn of two players.

For pieces.h and game.h, most underlying logic remains the same, and the only changes that should be made is for the movement rules for interaction between 4 players to be accommodated, such as check, checkmate and capturing work.

Extra Credit Features:

Unique Pointer

In this project, we used a *unique_ptr* to manage dynamic allocation of the objects *Board* and *Players*. This helps to prevent commonly-seen memory errors and ensures efficient memory management.

```
Game::Game(){
  board = make_unique<Board>();
  scoreBoard["white"] = 0;
  scoreBoard["black"] = 0;
}
```

When the program starts, a *Game* object is created which creates a *unique_ptr* to a *Board* object. This means that the *Board* is dynamically allocated and exclusively owned by the pointer. Therefore, the *Board* object gets automatically deleted when the *Game* is destroyed, preventing memory leaks.

```
void Game::start(unique_ptr<Player> white, unique_ptr<Player> black) {
  player1 = move(white);
  player2 = move(black);
  gameStart = true;
  display();
}
```

The *start(unique_ptr, unique_ptr)* method takes in a *unique_ptr* as players, and transfers the ownership of the players to *player1* and *player2* in *Game*. By using a *unique_ptr* transferring ownership can be done with the *move(unique_ptr)* command.

```
void Game::end() {
  cout << "The game has ended." << endl;
    printFinalScore();
    // Reset the game state if necessary
    board = make_unique<Board>();
    player1.reset();
    player2.reset();
    white = true;
}
```

At the end of the game, the *end()* method is called to reset the game state, *board* is reinitialized using *make_unique<Board>()*. The player pointers are reset using the *reset()* command. This ensures that the board and player objects are properly deleted, preventing memory leaks.

```
Board* temp = dynamic_cast<Board*>(board.get());
map<string, map<string, int>> allMoves = temp->allMoves(isWhite);
```

level2(unique_ptr&, bool) and *level3(unique_ptr&, bool)* methods receive a *unique_ptr* reference to a *Board* object. The *get()* method is then used to retrieve the object pointed to, and cast to a *Board* pointer for temporary use within the method. This does not change the ownership of the *Board* object.

By using *unique_ptr* to manage the *Board* object, the *ComputerPlayer* can ensure that it will not cause any memory errors as the *unique_ptr* ensures that memory is safely and automatically managed, preventing potential leaks or dangling pointers. Hence, the use of *unique_ptr* is used to ensure automatic memory management, exclusive ownership and the safe transfer of ownership.

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?

The main thing learned through this project is the importance of communication. While working on the project, one of our biggest errors was miscommunication about the main framework. Not all members were fully aware of how the implementation was supposed to work and the interfaces were not documented properly. Because of this, when we met up to put all our pieces together, we found that a lot of our parts did not fit together the way it should have. After this, we had a more in depth discussion about our design, opting to change parts to flow together better and changing our parts, pushing connecting all our parts for the next day. This caused our timeline to be pushed back a day.

Another thing learned was the use of shared repositories to store code. We used a github repository for this project and some members did not have any prior experience with this technology. Through working on this project, we were able to practice using this technology and learn new skills. This also transferred into learning to use version control and collaboration tools, our main communication was done through discord, though it may not be considered a professional technology, it was still a learning experience for members that did not have prior experience with this technology. This project also refreshed and enhanced our skills in object oriented programming and other programming skills learned in this class and previous classes.

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

If we could start again, the main thing would be to create a more robust plan, and make sure that every single member is on the same page before diving into execution. As mentioned

above, many of our issues arose from a lack of communication and planning. This ambiguity leads to vastly different implementations of different components of the project. During the initial planning part of the process, we would need to clarify and write better documentation for our interfaces, and throughout the execution, consistent communication on the changes made should be made and the overall direction should always be aligned to prevent immense changes and debugging towards the deadline.

Another thing we would have done differently is time management. Due to miscommunication, the bulk of our project was done in the last three days, which meant we did not have the ideal time to compile and debug. If we were to start a week earlier, our project would have been considerably refined and many features would have run more smoothly.

Conclusion:

In the Chess Project, through using an object oriented design, we were able to address many issues that commonly arise in procedural programs. Through programming, we become more familiar and grasp the concept of Object-Oriented Programming better. Particularly, features such as inheritance, abstraction and many other features were understood more clearly with practical examples. Looking forward, if there is still a chance to improve the implementation of the program, we would like to incorporate additional features, such as a level 4 computer player, graphical interface or a book of standard openings.

In developing this project, we learned a lot about collaborative programming in a team and the importance of communication and planning when many people are involved. We experienced the significance of teamwork and time management, and how these contribute to the successful completion of a project. Looking forward, with the experience of creating this project as a group, we believe we will perform even better in the future, as better programmers.