Objected-Oriented Chess in C++

Christopher Ryder
10185761
Department of Physics and Astronomy
University of Manchester
(Dated: May 11, 2020)

The main objective of this project was to develop a ‘full’ implementation of a standard game of Chess. Object oriented principles in the form of polymorphism and inheritance were used to implement the Chess pieces and a more advanced object-oriented design pattern was successfully implemented to better abstract and describe the concept of a move. The combination of these elements resulted in the production of a comprehensive and correctly operating Chess game.

I. INTRODUCTION

Chess is in an ancient, two-player, strategy board game with origins thought to be from the Persian empire around 500 BC [1]. However, the game has changed considerably since then, mutating strongly based on the region and time-period in which it was played. It was not until around 1475 when the first recognisable variants of ‘modern chess’ appeared in western Europe. The emergence of competitive play and tournaments in the 19th century necessitated the need for a standardised rule book, which is currently managed and revised by the FIDE.

Throughout recent history, Chess has been used as a benchmark in computer development, with research on digital chess-playing machines beginning around the 1950s. These initial machines had limited computing power and had to perform ‘selective searches’ through the search space of Chess. This search space can be likened to a decision tree in which a computer will consider a move, evaluate how strong the move is, and then consider the possible responses to this move up to a certain ‘depth’. Early Chess computers could typically search to a depth of about 3 half-moves; a half-move describes a turn taken by one player and a full move represents each player having taken a half move. These early computers were not particularly strong and could still be easily beaten by human chess players. It was not until 1997 that the infamous ‘Deep Blue’ computer beat the then current reigning chess champion, Garry Kasparov[2], under tournament conditions. Unlike its predecessors Deep Blue could perform a brute-force approach to determining the optimum move by enumerating all possible moves 7-8 half-moves deep. The size of search space at a depth of 7 half-moves is 3,195,901,860 nodes. Deep Blue could perform this search at a rate of 500,000,000 nodes per second.

A standard game of chess is composed of two teams, each with 16 pieces arranged on a 8x8 board, as shown in figure 1. Each type of piece, of which there are 6, can move in a unique way and some even have unique abilities. Consider the Pawn, it may only move forward one position, relative to its faction, except for on its first move where it can move forward two positions - this is called a double push. If a pawn does decide to move forward two positions on its first move, then it becomes vulnerable to enpassant capture which is a capture that can only be performed by a horizontally adjacent enemy pawn immediately after the double push. In addition to enpassant captures, Pawns may also capture one position diagonally forwards from their current position. Pawns are also unique in that they are the only piece capable of promotion to another type of piece (excluding the King). Promotion is realised if the pawn manages to move to the opposite end of the board to where it began. The other pieces have much simpler rules, with the exception of the King: A Knight may move or capture in a crook or ‘L’ shaped pattern in any direction from its position, jumping over other pieces if necessary; a Bishop may move an indefinite number of positions or capture in any diagonal direction from its origin but cannot move through other pieces; a Rook is similar to a bishop but constrained to only horizontal and vertical directions; a Queen can be considered to be a composite of both a Bishop and a Rook. Finally, the King may only move or capture one position in any horizontal, vertical or diagonal direction but in doing so it cannot expose itself to attack via any enemy piece. Similarly, if the enemy moves a piece in

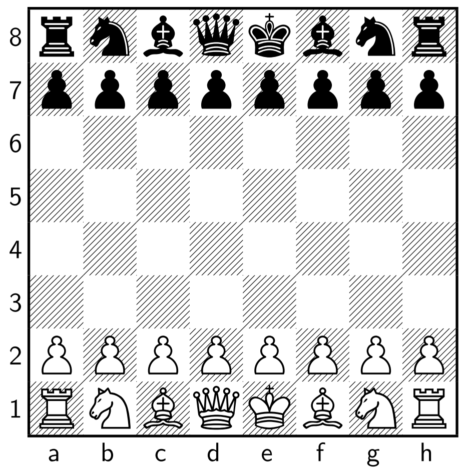
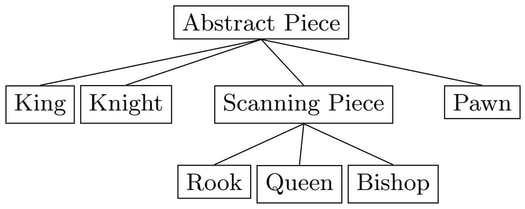


Figure 1. The entirety of rows 7 are occupied by Pawns. The symbols on rows 1 and 8 read from left to right as: Rook, Knight, Bishop, Queen, King



```
/*...*/
virtual ~Piece() = 0;
virtual void move(const int position);
virtual MoveList generate_move_list(...);
/*...*/
```

Figure 2. An abridged class hierarchy representing the game pieces and some important methods and properties belonging to the interface.

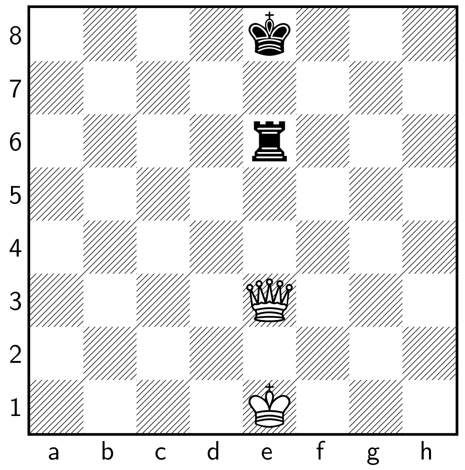
such a way that it would enable them to 'capture' the King the player is said to be in Check and their next move must remove this threat of capture. If a player cannot do this, then the game is lost and they have been 'checkmated'. The King is also capable of 'castling' with a friendly Rook, provided that neither piece has moved previously. Castling can be best described as the king moving to the left or right by 2 positions and then the Rook moving to the inside position of the King. Checkmate is the only way to win a Chess game, but the game can also end in a stalemate, in which neither player wins. Stalemates are reached if the player cannot make a legal move (a move which does not put their king in check) but is otherwise not in check. A game may also reach stalemate if 50 half-moves have passed since the last Pawn move – this rule is known as the 50-move draw [3].

II. PROJECT DESIGN

Chess is a complex game and contains quite a few exceptional rules but the use of object-oriented programming allows us to abstract the game into smaller concepts, for example, each game can be broken down into the interplay of a board and a set of pieces. Although each piece appears to be quite different, they all have common functionality allowing us to write an abstract interface common to all types.

This interface can then be inherited and overridden if necessary for each derived piece. The hierarchy used in this program is shown in figure 2 and defines a subgrouping called 'scanning pieces'; these pieces have special implications for move generation as they are capable of 'pinning'. In Chess, a pin is defined as a piece restricting the movement of another, for example see figure 3.

Before moves are generated, pins are calculated by determining if the scanning-piece is aligned with the enemy king, and if so, it then tries to determine how many enemy pieces lie on the vector between them. If only one enemy piece lies on that vector then the piece is pinned



```
struct Pin {
        int vector{ 0 };
        int position{ 0 };
};
```

Figure 3. In the following example, the black Rook is pinned by the white Queen and is only able to move in the vertical direction. Horizontal movements would expose check. The code snippet serves to demonstrate that you can represent any pin with two pieces of information, the location of the pin and the vector in which the restriction applies.

and receives a Pin object containing the restriction.

In order to generate a move, we must first define what is meant by a move. This program defines moves as command objects, utilising a design pattern called The Command Pattern which at its simplest encapsulates a command as an object by defining an abstract interface for performing the command [4].

```
    virtual ~CommandInterface() = 0;
virtual void execute(Context& context) = 0;
virtual void undo(Context& context) = 0;
```

Figure 4. The abstract interface required to implement the Command Pattern.

This interface is to be inherited by a 'concrete' command class which will implement its methods and hold any state relevant to those methods; commands can also be given a context. For example, consider the implementation of a standard move command shown in figure 5.

The move command has implemented the interface and requires a board as context to be able to execute the move. In addition, it has defined the function `virtual bool validate(...);` which will determine the legality of the move. Each command holds only the necessary information required to execute that command: "What you don't use, you don't pay for" [5]. Each piece is capable of being able to generate some of the commands from the hierarchy in fig 6 using its `virtual MoveList generate_moves(...);` function.

Chess programs tend to either utilise a 'legal move-generator', which will validate every move, or a 'pseudo-

```
/*...*/
virtual ~MoveCommand() {}
virtual void execute(Board& board) override;
virtual void undo(Board& board) override;
virtual bool validate(Board& board) override;
/*...*/
```

Figure 5. A concrete command will override and implement the following interface methods. In the case of a Chess move command, and extra method has been added to the concrete-command to enable the move to validate itself.

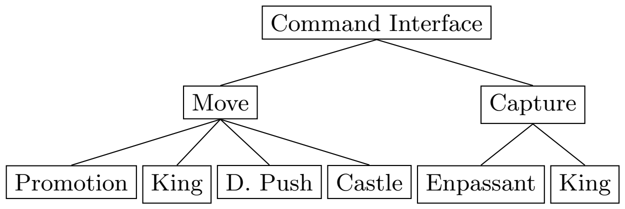


Figure 6. The class hierarchy representing the command pattern in the context of this program. Double Push has been shortened to 'D. Push' and King represents either a King Move or King Capture respectively - this distinction is made as King moves always require validation.

legal move-generator', which will only validate moves capable of exposing check. This program uses a psuedo-legal generator and validates explicitly: King moves, King captures, Castles and Enpassant Captures. The other moves are only validated if the player is in check. In order to generate the moves each piece needs context of the greater board to determine where the other pieces lie. This is given to the piece via the board class which holds a vector of 'tile' objects. A tile object is a collection of two enumerations: a tile-type and a faction-type. This allows for the easy comparison of tile objects with one another to deduce whether a position can be moved to or captured. When generating a move, a piece will iterate through their allowed directions of movement and traverse the board in that direction, adding empty tiles to their list of moves and enemy-occupied tiles as captures - provided they satisfy their commands verification criteria. Validation is usually performed by simply making the move, checking whether the player is in check and then un-making the move.

The Board class contains a lot more than just the representation of the board, however. It is directly responsible for owning, via a `std::unique_ptr<T>`, the pieces on the board as well as holding a list of all the possible Move-commands. The board is able to check if a player has been mated by calculating the size of the 'move-commands' vector – If there are no possible moves and the player is in check they have been checkmated, otherwise if they are not in check the game is a stalemate. Notably, the only way to instantiate a board class is through a string describing 'Forsyth Edwards Notation' (FEN). Deciphering the content of FEN is beyond the scope of this paper but it is important to know that a FEN describes the entire game state of Chess. Regular

```
    std::smatch matches;
    std::regex_search(config, matches, fen);
```

Figure 7. The regex will search the string, denoted as 'config' and compare it to a regular expression, denoted 'fen', any matches will be appended to 'matches' which acts as a vector of strings.

```
while(/*...*/) {
    m_board->rotate_players();
    m_board->determine_current_check_state();
    m_board->assign_piece_pins();
    m_board->generate_composite_movelist();
    m_board->pretty_print_board();

    m_handler.listen(std::cin);
    }
```

Figure 8. The main game loop performs the following commands repeatedly until either the game is lost or a player quits or resigns.

Expressions were used to parse the FEN string and split into sub groups as shown in figure 7.

The game class serves to dictate the flow of the program by executing the code found in figure 8 until either a player quits or wins.

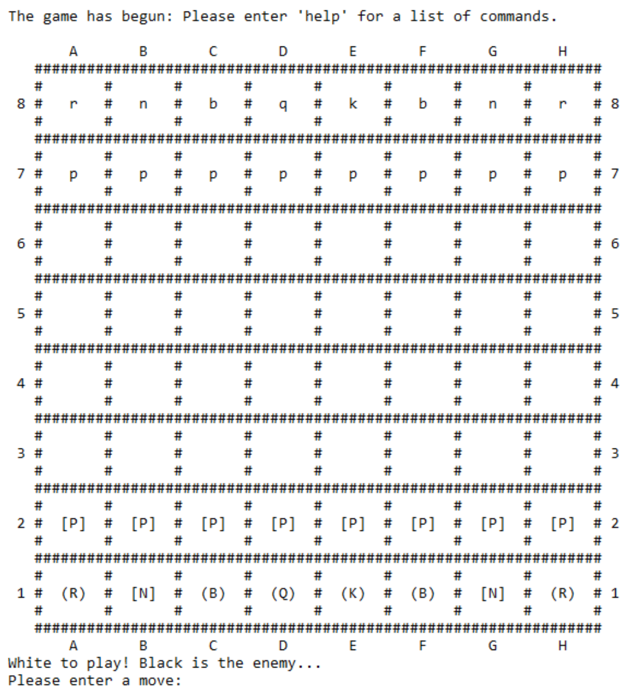
The input handler was designed to keep user input separate from game-logic and has been mostly successful in that function. The input handler attempts to match a user's command against all possible commands and upon finding a match is able to execute any of the following:

- a1;a2 (Move from a1 to a2)
- a1? (Prints moves for a1)
- help (Prints help)
- resign (Resigns, player loses.)
- quit (Quits, neither player wins.)

If given chess coordinates they are ran through an `std::unordered_map<std::string, int>`[6] which contains valid algebraic chess coordinates as a key and the corresponding board position as the value. The find algorithm can be used on the map to look up and determine if the input corresponds to a board coordinate.

III. RESULTS AND ANALYSIS

The program executes as expected and all the commands described previously work as intended. Included in the program is a 'debug suite' designed to check some of the more complicated scenarios which may arise in a game of chess, primarily related to move-generation. The program, in its current state, passes all of these tests successfully.



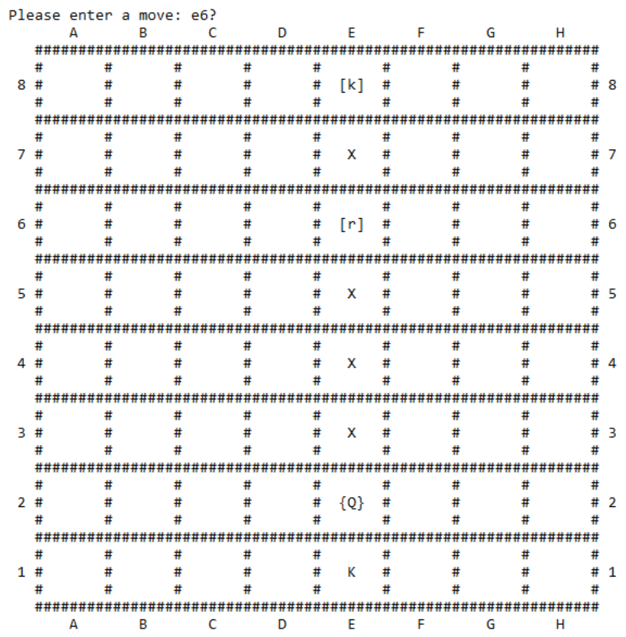
```
The game has begun: Please enter 'help' for a list of commands.

     A       B       C       D       E       F       G       H
  #################################################################
  #       #       #       #       #       #       #       #       #
8 #   r   #   n   #   b   #   q   #   k   #   b   #   n   #   r   # 8
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
7 #   p   #   p   #   p   #   p   #   p   #   p   #   p   #   p   # 7
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
6 #       #       #       #       #       #       #       #       # 6
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
5 #       #       #       #       #       #       #       #       # 5
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
4 #       #       #       #       #       #       #       #       # 4
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
3 #       #       #       #       #       #       #       #       # 3
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
2 #  [P]  #  [P]  #  [P]  #  [P]  #  [P]  #  [P]  #  [P]  #  [P]  # 2
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
1 #  (R)  #  [N]  #  (B)  #  (Q)  #  (K)  #  (B)  #  [N]  #  (R)  # 1
  #       #       #       #       #       #       #       #       #
  #################################################################
     A       B       C       D       E       F       G       H
White to play! Black is the enemy...
Please enter a move:
```

Figure 9. The board at the start of a new game. Pieces belonging to the current player are surrounded by brackets. Square brackets indicate a piece can make moves and circular brackets indicate that a piece cannot.

Profiling of the program using the visual studio profiler determined that there were no bottlenecks as a result of my code. The most frequently called function not related to input or output was the access operator for the board object. None of the functions critical to the algorithm of the Chess game played a part on the hot-path of the program, the path responsible for taking up the most time during execution. This goes to show that the main bottle-neck of the program is related to input and output and not move-generation or execution.

IV. CONCLUSION

Although the objectives of the project were met with success there are still a lot of improvements which could be made. Currently, there is no definitive way to check if the move-generator behaves correctly in all possible situations, the debug-suite tests some more complicated scenarios but it is not exhaustive. There exists a method of verifying move generators called 'Perft' (performance test, move path enumeration). This involves enumerating all possible moves for each piece up to a certain depth to get the total number of moves available. These numbers can then be compared to accepted values generated by established chess engines to determine the accuracy of the move generator. Similarly, more trivial extensions could be made through the introduction of a 'turn-timer' which would only allow each player a certain amount of time to make each move; this would probably require the use of multi-threading, however. Finally, the program would benefit massively from a GUI to display the given board and the ability to take mouse input.



```
Please enter a move: e6?
     A       B       C       D       E       F       G       H
  #################################################################
  #       #       #       #       #       #       #       #       #
8 #       #       #       #       #  [k]  #       #       #       # 8
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
7 #       #       #       #       #   X   #       #       #       # 7
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
6 #       #       #       #       #  [r]  #       #       #       # 6
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
5 #       #       #       #       #   X   #       #       #       # 5
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
4 #       #       #       #       #   X   #       #       #       # 4
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
3 #       #       #       #       #   X   #       #       #       # 3
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
2 #       #       #       #       #  {Q}  #       #       #       # 2
  #       #       #       #       #       #       #       #       #
  #################################################################
  #       #       #       #       #       #       #       #       #
1 #       #       #       #       #   K   #       #       #       # 1
  #       #       #       #       #       #       #       #       #
  #################################################################
     A       B       C       D       E       F       G       H
```

Figure 10. Example of 'a1?' functionality. Captures of enemy pieces are highlighted by curly-brackets and moves are indicated by an 'X' in the empty box.

[1] T. Daryaee, Mind, body, and the cosmos: chess and backgammon in ancient persia, Iranian Studies **35**, 281 (2002), www.tandfonline.com/doi/pdf/10.1080/00210860208702022.

[2] M. Campbell, 20 years after deep blue, a new era in human-machine collaboration, THINK Blog (2017), https://www.ibm.com/blogs/think/2017/05/deep-blue/.

[3] P. Wolff, *The complete idiot's guide to chess*, 2nd ed. (Alpha, Indianapolis).

[4] A. B. Singer, The command dispatcher, in *Practical C++ Design: From Programming to Architecture* (Apress, Berkeley, CA, 2017) pp. 57–96.

[5] B. Stroustrup, *The Design and Evolution of C++*, Programming languages/C+ (Addison-Wesley, 1994).

[6] B. Stroustrup, *A Tour of C++*, C++ in-depth series (Addison-Wesley, 2014).

Appendix A: Impact of COVID19 on the work completed and report

- Please describe the effect of the COVID19 under the following headings. If there were difficulties of a personal nature, such as in your home situation, that you would rather communicate directly to prof. Walet or physics.support@manchester.ac.uk, please do so!

- Time lost travelling home: Some of you may have had substantial difficulties travelling home, and lost a lot of time on the way. Please give details here: N/A

- Lack of access to computers and the internet: We understand that not all students have access to sufficient IT resources. Please specify what the limitation of resources is: N/A

- Limitation in the outcome: Please describe how this has impacted both the extent of the results presented, and the quality of the report:

  I am however concerned that this appendix is going to push me over the word count!