”

An Object-Oriented Chess Game in C++

10127484
University of Manchester
School of Physics and Astronomy
Object Oriented Programming in C++
(Dated: May 24, 2021)

A chess game incorporating most standard chess rules was designed and implemented in C++, demonstrating the use of the three fundamental principles of object-oriented programming: encapsulation, inheritance, and polymorphism. The user interacts with the engine through a terminal shell, where moves can be inputted in long algebraic notation. The board and menus are printed to the terminal. Additional functionality includes the ability to load games from and save games to a text file, undo moves, and a tool for enumerating the number of moves to an arbitrary depth.

I. INTRODUCTION

For the past 20 years, chess engines have far surpassed human grandmasters in skill. The first engines were built in the 1950s, and were beatable by novices. Their limited processing power meant that they could only evaluate moves 3 turns in the future, whilst it is generally agreed that evaluation to a depth of 10 turns in the future is required for good play [1]. The number of positions increases exponentially with the number of moves, and so brute force searches require massive computing power - there are estimated to be a total of 10^{120} possible positions [2]. Developments in the manufacture of more powerful processors therefore resulted in increasingly powerful chess engines.

In order to evaluate many positions quickly, the game must be represented in a computationally inexpensive way. One such representation is the 0x88 representation, where each square is represented by a hexadecimal digit, allowing many computations such as determining if a piece is off the board to be carried out using bitwise operations rather than integer comparisons [3]. State of the art chess engines will have other optimizations to maximise the computation power from multi-core CPU architectures

If a computer opponent is not required, however, computation speed is not an important factor, since the ability to enumerate and evaluate positions is not necessary. Thus a more intuitive square list representation of the board is used in this project, where an 8×8 array of pointers to pieces describes the state of the board.

This project aims to create a command line based chess game for two players demonstrating the use of modern object-oriented programming principles. To this end, the Resource Acquisition is Initialization design pattern is followed, making use of `unique_ptr` and STL containers such as vectors to handle dynamic memory allocation. All chess rules are implemented, apart from *en passant* capture, castling, and pawn promotion.

II. CODE DESIGN

The program contains four classes: the board, the pieces, the menu, and the game. Their relationships are illustrated in Figure 1. The functionality of each is discussed below.

A. Piece

The six piece types all derive from an abstract base class `Piece`. The three sliding pieces (rook, bishop, and queen) are derived from an intermediate abstract class `Sliding_piece`. `Piece` has a pure virtual method `get_possible_moves()` which takes as an argument a pointer to the 8×8 array that represents the game board, which is a member of the class `Board`, as well as the current location of the piece on the game board. This method is overridden in each of the derived classes in order to implement each pieces' movement rules. The `Sliding_piece` class introduces the method `get_sliding_moves()`, which generates the possible moves for a piece sliding in a given direction. For example, to generate all the moves for a rook, `get_sliding_moves()` is called in the $[0, \pm 1]$ and the $[\pm 1, 0]$ directions. Figure 2 illustrates the implementation of the method.

The other pieces (king, knight, pawn) have unique implementations for `get_possible_moves()`. The pawn, for example, behaves differently if it is black or white, since the direction that is “forward” is different for black and white pawns. Moreover, a pawn may move two spaces forward if it is its first turn, and so must keep track of how many moves it has made. The knight searches all squares within two units of itself, and ensures that, for a move i units in the file direction and j units in the rank direction, $|i| + |j| = 3$.

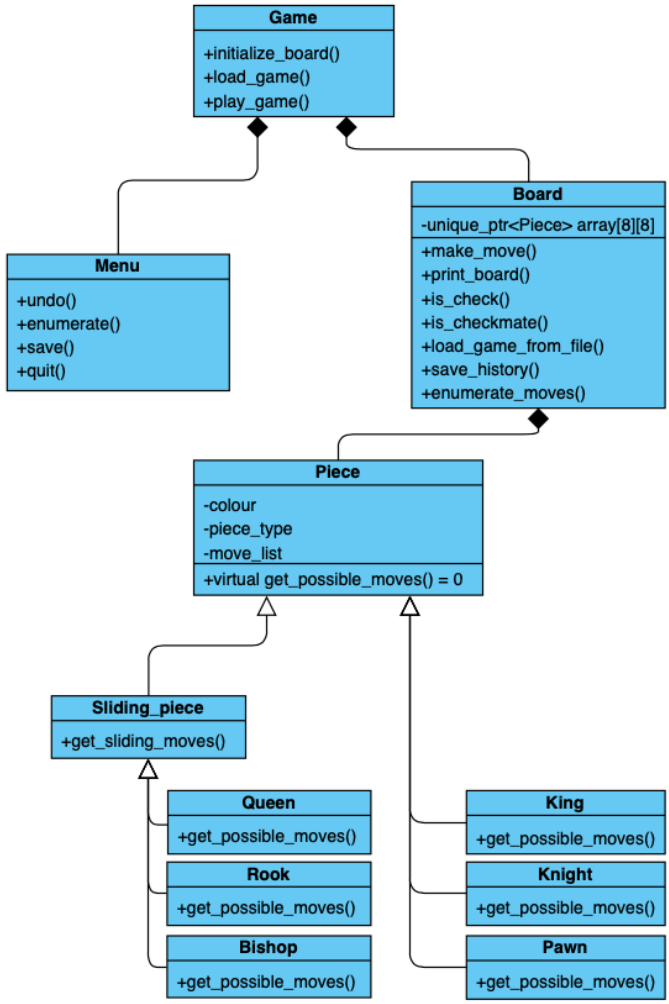

Figure 1. UML class diagram for the chess engine. Some attributes and methods have been omitted for conciseness.

B. Board

The `Board` class is responsible for checking the validity of moves, executing them, and checking for checkmate. It also has a method to enumerate the number of possible moves after any number of half-turns (plies), as well as saving and loading games to and from files. The class also includes methods to facilitate the conversion from long algebraic notation (e.g. "e2 e4" to move from e2 to e4) to a row/column format. In algebraic notation the bottom left corner of the board is denoted $a1$, with the file increasing from a to h rightwards, and the rank increasing from 1 to 8 upwards. The array storing pointers to the pieces, however, is indexed from $[0, 0]$ in the top left corner to $[7, 7]$ in the bottom right corner. For example, the method `file_to_col()` uses a map to convert the user input of type `char` to an integer.

The array representing the game board is an array of `unique_ptr` to pieces, rather than normal pointers. This has the advantage that the pieces cannot be owned by more than one object - that is, they cannot be copied, and can only be moved. It is therefore impossible, for example, to accidentally have a piece occupy two spaces on the board at once. Moreover, memory cleanup is handled automatically when the board goes out of scope at the end of the program.

The method `is_move_valid()` accepts a move in the long algebraic form as a string, and converts it into the row/column format. It then compares the move against the vector containing the possible moves for that piece, and if the move is found in the vector, then the move is valid and the method returns `true`. Note that `is_move_valid()` only checks a move is *quasi-legal* - it does not test whether the move would put the player into check (and therefore would be an illegal move).

In order to execute a move, the method `make_move()`

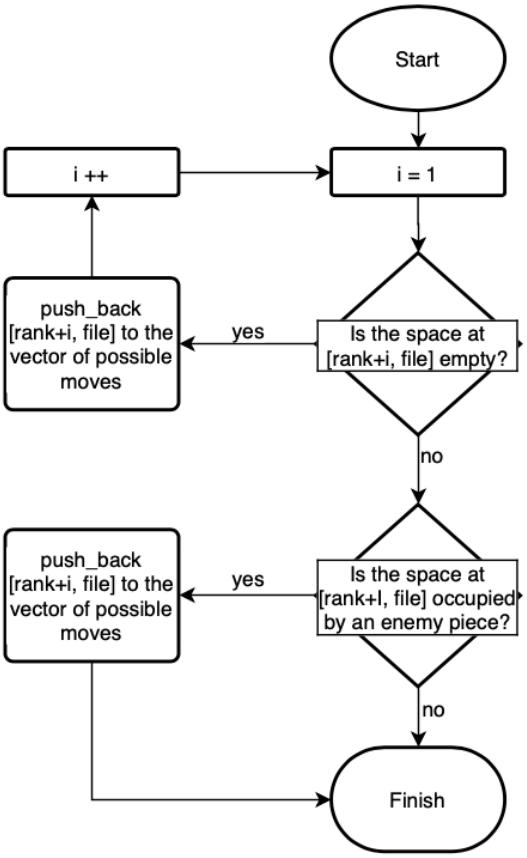

Figure 2. Flowchart representation of the method `get_sliding_moves()` for a rook searching in the increasing rank direction. The method `get_sliding_moves()` is called four times in different directions by `get_possible_moves()`.

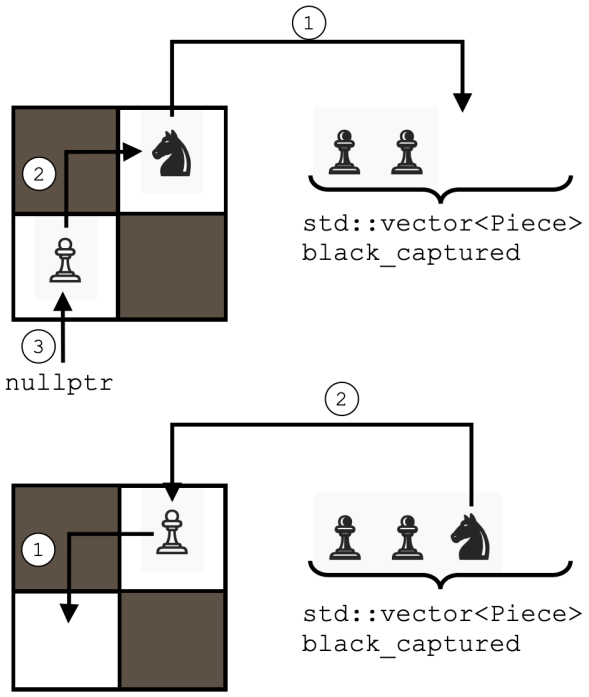

Figure 3. *(Upper)* Visual representation of the method `make_move()` for a white pawn taking a black knight. The order the operations are carried out is denoted by the numbers in the circles. *(Lower)* Visual representation of the method `undo_move()`.Note that the vector `black_captured` is actually a vector of unique pointers to a `Piece`.

```
e2 e4 | e7 e5
g1 f3 | b8 c6
f1 b5 |
```

Figure 4. Example of a saved game `.dat` file. Shown is the Ruy Lopez (Spanish game) opening.

is called. This is illustrated in Figure 3. First, the method checks if the move is a capture. If it is, it moves the piece being captured into a vector of captured pieces, using `push_back(std::move(Piece))` ((1) in Figure 3, *upper*). Then the piece is moved from its "source" location to the "target" location (2) using `std::move()`. If the move is not a capture, the first step is skipped. The "source" location is then set to `nullptr` (3). Then the string representation of the move is added to a vector of moves, `move_history`. If the move was a capture, the space between coordinates is replaced with a colon, for example "e4:d5".

The board also has a method to save the move history to a file. An example file is shown in Figure 4. The method asks the user for a filename to save to, opens the file and loops over the vector `move_history`, printing the contents to the file. The method `load_game_from_file()` can be called before the game starts. It asks the user for a filename, checks it can be opened, and then reads the moves in one by one, executing them with `make_move()`. Control is then returned to the user.

A move may be undone by calling `undo_move()`. The process is similar to that of `make_move()`, but it happens in reverse. First, the piece at the "target" square is moved to the "source" square. If the move was a capture, the captured piece is moved out of the captured pieces vector, and placed at the "target" location. If it was not a capture, the "target" location is set to `nullptr`. This is illustrated in Figure 3, *lower*.

In order to determine if the king is in check, the method `is_check()` loops over all the opponent's pieces on the board, and checks their possible move lists. If any move in the move list is the same square the king is on, the king is in check. The `is_checkmate()` method checks whether the king can escape check by looping through all the possible moves, making them one by one, and calling `is_check()` each time. If

the move results in the king no longer being in check, `is_checkmate()` returns `false`.

The recursive method `enumerate_moves()` counts the number of half moves available to a given depth. For example, at the start of the game, there are 20 moves for a depth of 1 ply. This means there are 20 moves available to white. To a depth of 2 ply, there are 400 moves, since for each move white can make, there are 20 black can make. The moves are enumerated by first looping through all the pieces on the board belonging to whichever player's turn it is. For each piece, the move list is generated with `get_possible_moves()`. Then each move is made in turn and checked if it is legal. If it is, `enumerate_moves()` is called again, but this time enumerating the opposite colour's moves, with a depth one less than before. This process is repeated until the depth reaches zero, where the method returns 1. The number of moves in each iteration is fed upwards until every possible version of the board has been counted.

C. Menu

The menu may be called at any time instead of entering a move by typing "menu". The menu offers five options to the user: *return to game*, *undo move*, *enumerate moves*, *save game*, and *quit*. The menu accepts user input and checks that it matches one of the options, and then calls the appropriate method on the `Board`.

If the user selects *enumerate moves*, the number of moves up to a depth of four ply is printed, along with the time taken to enumerate moves. This is achieved using the `chrono` library.

D. Game

The class `Game` is responsible for controlling the flow of the game. The method `play_game()` contains a loop that begins by checking that the player whose turn it is is not in checkmate. If they are, the game is over, and the user is instructed to use the menu. If they are not in checkmate, the user may input a move or a call to the menu. If the move is quasi-legal and does not put the player into check, the move is made, the turn is over, and the board is printed again. The loop then returns to the beginning.

`Game` also contains the method `initialise_board()`, which is responsible for dynamically allocating the memory for the pieces. This function is called within `main()` inside a `try` / `catch` block, so that if the memory allocation is unsuccessful, an error message can be printed and the game can terminate.

```
   a  b  c  d  e  f  g  h
  ========================
8 [r]   [b] q [k] b [n] r      White to move
7    [p] p [ ]   [p] p [p]     Type [menu] to open the menu
6 [p]   [p]   [ ]   [ ]
5    [ ]   [ ] p [ ]   [ ]     Captured white pieces:
4 [ ]   [ ]   [P]   [ ]             B
3    [ ]   [ ]   [N]   [ ]
2 [P] P [P] P [ ] P [P] P      Captured black pieces:
1  R [N] B [Q] K [ ]   [R]          n
  ========================
```

Figure 5. Screenshot of the terminal halfway through a game, demonstrating the user interface. The game shown is a continuation of the Ruy Lopez shown in Figure 4, with black playing the Morphy defence (3... a6), followed by white playing the Exchange Variation. The game is waiting for white to input a move.

```
  ========================
     Main menu:
       - [r]eturn to game
       - [u]ndo move
       - [e]numerate moves
       - [s]ave game
       - [q]uit
  ========================
e
Enumerating moves
This may take a few seconds...
1 ply: 20 nodes          1 ms
2 ply: 400 nodes         30 ms
3 ply: 8902 nodes        639 ms
4 ply: 197281 nodes      14312 ms
```

Figure 6. Screenshot of the terminal after selecting *enumerate moves* from the menu.

III. RESULTS

Figure 5 shows how the game board is displayed to the user. White pieces are represented with a uppercase letters, and black pieces are lowercase. Note that the knight is represented by an "N", in order to avoid confusion with the king, "K". Although there exist the unicode characters to print black and white chess pieces, since terminal colour schemes across platforms may not be consistent, it is safer to differentiate white and black pieces in a way that will be consistent across platforms. In addition to showing the game board, additional information such as the vectors of captured white and black pieces are printed on the right hand side.

Figure 6 shows the menu, and the resulting display after the user inputs "e" to enumerate moves. The number of nodes found can be compared to a modern chess engine such as Stockfish 13 [4] to check that the rules of the game have been implemented correctly. In Figure 6, `enumerate_moves()` was called at the start of the game, so all pieces were in the starting position. To a depth of 4 ply, the number of nodes in Figure 6 is congruent with the results from Stockfish. At 5 ply, however, the C++ engine counts 4 865 495 nodes, whilst Stock-

fish counts 4 865 609. The discrepancy is due to the fact that the C++ engine does not include *en passant* capture or castling.

Note that with a dual core 2.7 GHz Intel Core i5 processor, it takes over 14 seconds to enumerate moves up to a depth of four ply. Modern chess engines, on the other hand, can not only find but also analyse the strength of millions of nodes per second - the same test using Stockfish takes less than a tenth of a second.

IV. CONCLUSION

A simple chess game for two players implementing most of the standard chess rules was created. To check that pieces' movement rules and rules for check were implemented properly, the Stockfish engine was used for comparison. It was demonstrated that Stockfish and the C++ engine both predict the same number of positions, until *en passant*, castling and pawn promotion are possible, at which point the C++ engine underestimates the number of possible moves, as expected. It was also demonstrated that the Stockfish engine was faster at enumerating the number of moves by up to a factor of 100.

Beyond including the full chess rule set, to extend the project, a graphical interface could be designed and implemented, or the engine could be modified to interface with open source chess GUIs such as ChessX or Arena. The players could therefore interact with the game in a more intuitive manner, by dragging and dropping pieces rather than typing moves in algebraic notation.

[1] A. Kotok, "A chess playing program for the ibm 7090 computer," Master's thesis, Massachusetts Institute of Technology, 1962.

[2] C. E. Shannon, "Programming a computer for playing chess," *Philosophical Magazine*, vol. 41, 1950.

[3] F. M. H. Reul, *New Architectures in Computer Chess*. PhD thesis, Tilburg University, 2009.

[4] "Stockfish 13," 2021. `https://stockfishchess.org`.