Chess Game with Object-Oriented Programming

Flavio Ryu Ciciriello
10367400

School of Physics and Astronomy

University of Manchester

PHYS30762 OOP in C++ Final Project Report

May 2021

Abstract

A chess game program was written in C++ making use of the advantages of Object-Oriented Programming such as encapsulation, inheritance and polymorphism. It is designed to be played by two players that interact with the game using the terminal (or console if the .exe file is launched), where the chess board is displayed after each player move. The code successfully implements all the major rules of the game including pawn promotion and castling. The game is terminated when a checkmate or a stalemate is detected.

1 Introduction

Chess is a classic board game with a substantial amount of history behind it. Despite different schools of thought exist, it is widely believed that it first originated in India around the 7th century A.D. and slowly arrived in Europe around the 15th century in a form very close to the modern one [1]. Today it is the most popular board game in the world, with millions of players, especially online. In the recent years it has gained an increasing popularity due to the rise of online websites such as Chess.com (2005) or Lichess (2010). These websites make use of sophisticated chess engines to allow online players to play together.

A chess game is undoubtedly the best demonstration for a project in Object-Oriented Programming since it requires a deep understanding of the language to most efficiently code the complex structure of the game.

This game was written using the VS Code program and compiles with Mingw-w64 (g++ ver. 8+). The entire game is terminal-based so that it is played locally by two users that interact with the terminal by typing on the keyboard. After each move, the updated chess board is displayed using appropriate functions.

1.1 Basic Rules

The board contains 64 squares ordered in a 8×8 square grid labelled with letters (files) and numbers (ranks), a system known as "algebraic notation". An image of the board is shown in Figure 1.

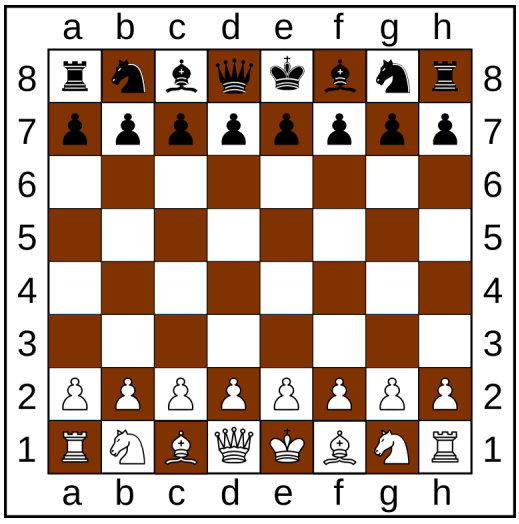


Figure 1: Standard western chessboard [2]. Each square can be identified by its file and rank.

Each player has a colour (white or black) and starts with an identical set of 16 pieces: 8 pawns, 2 rooks, 2 knights, 2 bishops a queen and a king. The objective for both players is to put the enemy's king in checkmate. A king is in checkmate if it is both in check and has no available moves. Being in check means that there is at least one enemy piece that can capture the king

in the next turn. A game can also end with a stalemate, which happens when a player does *not* have their king in check but they have no other moves available.

A piece is characterised by its own unique type of movement, and pieces of different colours have same allowed movements except for the pawns. In addition, a piece cannot skip other pieces while moving, except for the knights. Specific piece movements are described below.

- Pawn: Can only move one square forward, but if in their starting position, they are also allowed to move two square forward. If in the forward diagonal squares there is an enemy piece, they can move in that position by capturing. Movement is different between black and white pieces since black pawns move in decreasing rank direction, while white pawns move in increasing rank direction.
- King: Can only move one square around their current position, with the condition that those squares are not threatened by enemy pieces.
- Queen: Can move an indefinite distance both in diagonal and veritcal/horizontal directions.
- Bishop: Can move only in diagonal directions.
- Knight: Can move one square on the side after going two squares forward in any direction, skipping any piece encountered on the way.
- Rook: Can move only in vertical/horizontal directions.

2 Code Structure

To create a structure that can handle such a complex game, it is essential to abstract the concept of the pieces and the board. The most intuitive way to do this is to create a `board` and `piece` class, each with different class variables and member functions. This is where the concept of encapsulation becomes essential: the `piece` class will contain variables to identify its owner (either white or black) and its type (pawn, king, queen, etc.) together with instructions on how it can move. The `board` class will then contain an 8×8 dynamical array of `piece` class pointers that represents the chess board. Inheritance is also featured in the `piece` class since there will be a different class for each piece type but these will all be sub-classes of a generic abstract `piece` class. Polymorphism is also implemented through the usage of base class pointers: these allow to call the pure virtual `is_move_allowed()` function, obviously different for each piece type, that is overridden in the sub-classes (more details in Section 2.2). One of the most obvious advantage of using base class pointers to create the board is that when a piece is moved in a particular square, it is sufficient to assign the new pointer value to that square and delete the old one. For this reason, this approach is adopted by many programmers, as it can be seen in Reference [3]. A schematic diagram of the code is shown in Figure 2.

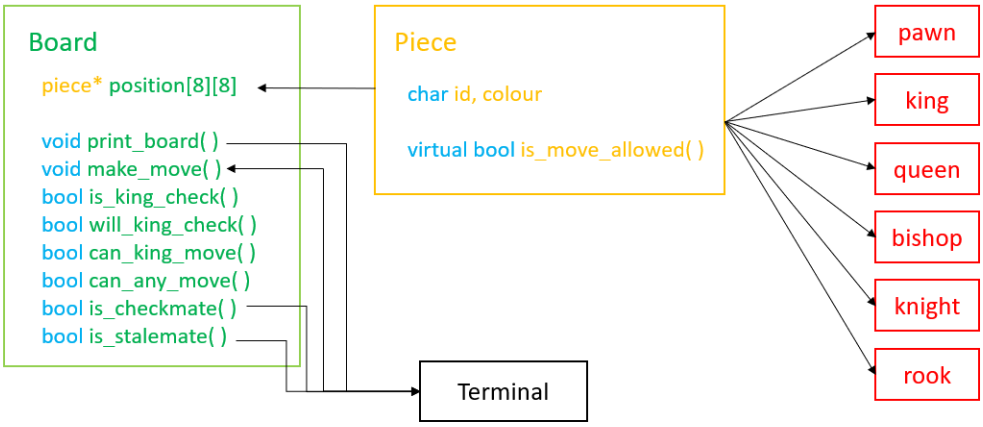


Figure 2: Schematic structure of the code. As it can be seen, the `board` class plays a central role by interacting with the terminal.

The code is divided into 3 header files (`board.h`, `piece.h`, `subpieces.h`), each with its own associated `.cpp` file, and a `main.cpp` file which contains only the `main()` function. In order to avoid any possible duplicate definitions, header guards are implemented.

In this paper, the `main()` function and the `.h` files will be illustrated in detail, together with how the user interacts with the terminal to play the game.

2.1 `main()`

The `main()` function is rather simple but all the major steps of the game are contained in it:

```
int main()
{
    board chessboard;
    bool is_draw{false};
    chessboard.start_message();
    chessboard.print_board();
    while (true) {
        chessboard.make_move();
        chessboard.alternate_turn();
        if (chessboard.is_checkmate()) {
            chessboard.print_board();
            std::string winning_player;
            winning_player = (chessboard.get_turncolour() == 'W') ? "Black" : "White";
            chessboard.end_message_win(winning_player);
            break;
        } else if (chessboard.is_stalemate()) {
            chessboard.print_board();
            chessboard.end_message_draw();
            break;
        }
```

```
            chessboard.print_board();
        }
        return 0;
    }
```

First, a `board` object called `chessboard` is created. This initialises the `board` class default constructor that create the starting board. Then a starting message is printed in the terminal and the initial board is displayed using the function `print_board()`. The game itself consist of a `while` loop that is broken when either `is_checkmate()` or `is_stalemate()` return true. Inside the loop, the `make_move()` function takes in the user input and moves the pieces accordingly (more in Section 2.3). If the move is legal, which means it is an allowed move within the rules of the game, the piece is moved and the turn is switched to the other player by the `alternate_turn()` function. After the move has been completed, or after a checkmate/stalemate has been detected, the updated board is printed again in the terminal. After exiting the loop, in case of a checkmate, the winning player is determined and an ending message is displayed before the program terminates.

2.2 Class `piece` and its Sub-classes

Since there is no need to create at any point instances of the `piece` class, this is set to be an abstract class by making `is_move_allowed()` a pure virtual function. This function will then be overridden in all the sub-classes, wich are: `pawn, king, queen, bishop, knight, rook`.

- **Attributes**
  The class variables of `piece` are set to the `protected` access specifier so that specific piece classes can access them:

  ```
  char id, colour;
  bool status{false};
  ```

  - `id` and `colour`:
    The characteristics (owner and piece type, *i.e.* the id) of each piece are stored as `char` variables. The ids of each piece types are: `'P'` (pawn), `'K'` (king), `'Q'` (queen), `'B'` (bishop), `'N'` (knight), `'R'` (rook).
  - `status`:
    Is initialised as `false` and holds the information of whether the piece has moved or not from the beginning of the game. It is used when applying constraints to the pawn's double move or the castling.

- **Methods**
  The (`public`) methods are:

  ```
  piece(char c, char i): colour{c}, id{i} {};
  char get_id();
  ```

```
char get_colour();
bool has_moved();
void set_has_moved();
virtual bool is_move_allowed(piece* temp[8][8], int irow, int icol, int frow,
                                                                    int fcol) = 0;
```

- `piece(char c, int i)`:
  The parametrised constructor is used in the sub-class constructors. For example, in the `pawn` class the parametrised constructors looks like

  ```
  pawn(char c): piece{c, 'P'} {};
  ```

  so that a `piece` class with `id = 'P'` is created. The colour is unspecified so that both players' pieces are created when allocating the initial pieces in the `board` class constructor (see Section 2.3).
- `has_moved()` and `set_has_moved()`:
  Are respectively the getter and setter for the `status` variable.
- `is_move_allowed()`:
  Is the virtual pure function that is overridden to all the sub-classes. For each piece class, it contains constraints on the movements, checking if the initial square and final square satisfy them; if they do, the function returns `true`. The parameter `temp` is passed so that the function has access to the board and can put constraints to pieces that cannot skip other pieces while moving (which is all the pieces except for the knight, as explained in Section 1.1). An example of the function for the `knight` class is shown below:

  ```
  bool knight::is_move_allowed(piece* temp[8][8], int irow, int icol, int frow,
                                                                    int fcol)
  {
      if ((abs(fcol - icol) == 1 && abs(frow - irow) == 2) ||
          (abs(frow - irow) == 1 && abs(fcol - icol) == 2)) {
          return true;
      } else {
          return false;
      }
  }
  ```

2.3 Class `board`

The `board` class is the central part of the code since it contains not only the information of the current state of all the pieces but also all the functions that regulate the game such as the constraints on the king movement or special rules like castling and pawn promotion. It also contains other functions to print statements on the terminal.

- **Attributes**
  The (`private`) class variables are:

```
char turncolour{'W'};
int ranks[8] = {8, 7, 6, 5, 4, 3, 2, 1};
char files[8] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};
int wking_row, wking_col, bking_row, bking_col;
piece* position[8][8]{nullptr};
```

- `turncolour`:
  Stores which player's turn is currently; it is a `char` variable and it is either `'W'` or `'B'`. It is initialised as `'W'` since the white player moves first.
- `position`:
  Represents the 8×8 board grid; all 64 values are initialised as `nullptr` but they are set to `piece` pointers when pieces are allocated in the constructor.

- **Methods**

The (`public`) methods of the class are:

```
board();
~board();
// miscellaneous functions
void print_board();
char get_turncolour();
void alternate_turn();
void start_message();
void end_message_win(std::string& player);
void end_message_draw();
// move-related functions
void make_move();
void input_checker(std::string& input);
int input_converter(char& i, int mode);
// king-related functions
void find_kings(piece* temp[8][8]);
bool is_king_check(piece* temp[8][8], char& king_colour);
bool will_king_check(int irow, int icol, int frow, int fcol, char king_colour);
bool can_king_move(char king_colour);
bool can_any_move();
bool is_checkmate();
bool is_stalemate();
// special moves
void pawn_promotion(int row, int col);
bool castling(char king_colour, int fcol);
```

Below, the most important functions are examined:

- `board()` and ∼`board()`:
  The constructor creates the initial pieces of the board and allocates them in the right positions. This is done by dynamical array allocation using the keyword `new` since each square is essentially a `piece` pointer. The piece allocation is extremely flexible

and one can start with any configuration by changing only few lines in the constructor (see *e.g.* Figure 4b). When the destructor gets called, it will delete all the squares by implementing the `delete[]` keyword inside a for loop.

– `make_move()`:
This is the biggest function of the class since it takes in the user input, validates it by calling `input_checker()`, converts it into coordinates in the board grid using `input_converter()`, checks that the move is legal and *makes the move*:

```
position[frow][fcol] = position[irow][icol];
position[irow][icol] = nullptr;
```

where `irow, icol, frow, fcol` are the initial and final row and columns of the move. A move is legal if:

* there is a piece on the initial square (*e.g.* `position[irow][frow] != nullptr`);
* the piece selected is of the same `colour` as the `turncolour`;
* the final square does not contain a piece with the same `colour`;
* `is_move_allowed()` for that move returns `true`;
* `will_king_check()` returns `false` (more on this below).

Before making the move, the function checks if the piece that is being moved is a king (`id = 'K'`), if this is the case and the initial and final squares correspond to that of a castling, it calls `castling()`. This function makes sure that both the king and the rook involved have never moved (`status = false`) and that the king is never in check in any square it travels through by making sure `will_king_check` returns `false`.
If the piece moved is instead a pawn (`id = 'P'`), it checks if it reached the final file. If this is the case, `pawn_promotion()` is called and the user is given the possibility to insert in the terminal one character from `'Q', 'R', 'B', 'N'`, and the pawn is then replaced by a new piece of the selected type.

– `is_king_check()`:
For a king of given colour (`king_colour`), it runs through all the squares on the board and checks if for any enemy piece the `is_move_allowed()`, evaluated from its position to the king position, returns true. If it does, it means that the piece can move to where the king is and `is_king_check()` is set to `true`.

– `will_king_check()`:
Creates a "virtual" board which is a temporary board identical to `position`, and makes a virtual move. After that, `is_king_check()` is called and if this returns `true`, it means that the king will be in check after the hypothetical move and therefore the function returns `true`.

– `can_king_move()`:
This function checks all the 8 squares around the king of given colour, and checks if the king would be in check if it moved there by calling `will_king_check()` on each

square. If there is at least one allowed square, the function returns `true`. Maps are used to iterate through all the 8 squares.

- `is_checkmate()`:
  This function simply returns `true` if `is_king_check()` returns `true` and `can_king_move()` returns `false`. Returns `false` otherwise.
- `is_stalemate()`:
  Returns `true` if `is_king_check()`, `can_king_move()` and `can_any_move()` are all `false`.

3 User Interaction

The game is played by interacting with the terminal. When the game starts, `start_message()` prints the initial message with the instructions of how to play, and when it ends, `end_message_win()` or `end_message_draw()` are called accordingly. The `print_board()` takes information from the `position` variable and heavily relies on `for` loops to print each segment of the board correctly. Symbols are used to distinguish white ([ ]) and black (<>) pieces. Examples of the printed board are shown in Figure 3.

The user inputs are all validated, meaning that if an input of exactly [letter]+[number] is not entered, errors will be risen depending on the error type. This is handled in the `input_checker()` function of `board` class. After the user successfully entered the file and rank of the initial and final squares, these are translated into coordinates of `position` in the `input_converter()` function.

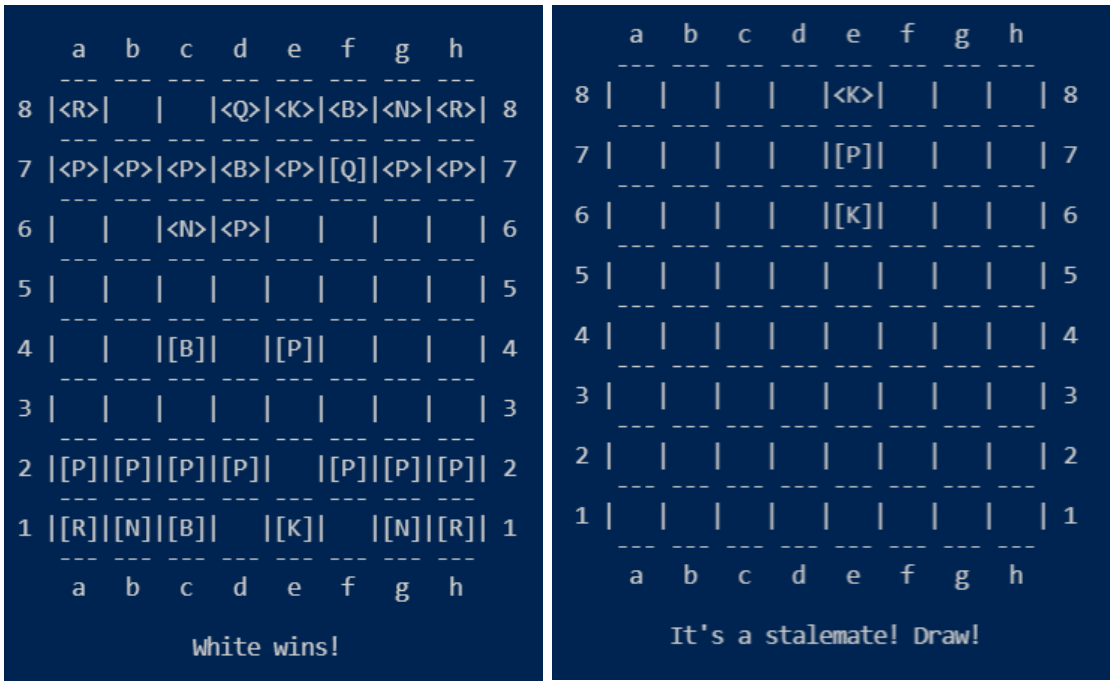
4 Results and Discussions

A chess game was written in C++ to demonstrate the main features of Object-Oriented Programming, which are encapsulation, inheritance and polymorphism. The pieces and the board are coded into `piece` and `board` classes and the various pieces classes are all inherited classes of the former. The `board` class is a core part of the code as it not only contains the information of the pieces but also all the rules of the game and functions to display outputs in the terminal.

The game runs successfully without any errors nor warnings, and it reproduces a real-life chess board game. An example of an end game is shown in Figure 4. Possible improvements to enhance the quality of the game can be the use of some external Graphical User Interface and to add time limits to each move. Adding a possibility to switch game mode between a user-user mode to a user-computer mode might also be interesting as it can show how a simple AI could be implemented (see for example Reference [4]).

(a) (b)

Figure 3: The board is organised such that squares that do not contain any piece are empty. Squares are easily identified by files and ranks printed on the sides. (a) Initial display of the game, showing the start message and the starting configuration of the chess board. (b) Example of a typical mid-game board configuration.



(a) (b)

Figure 4: Two examples of an end game: (a) the game is terminated because the black king is in check but cannot move elsewhere (if it captures the queen, it will still be in check by the bishop); (b) a simple case of a stalemate.

References

[1] R. G. Eales, *Chess: The History of a Game*. Facts on File, Inc., 1 ed., 1985.

[2] Wikipedia, "Chessboard and chess pieces." `https://commons.wikimedia.org/wiki/File:AAA_SVG_Chessboard_and_chess_pieces_02.svg`. Accessed on 24/05/21.

[3] XoaX.net, "C++: Chess." `https://xoax.net/cpp/crs/console/lessons/Lesson43/`. Accessed on 24/05/21.

[4] freeCodeCamp.org, "A step-by-step guide to building a simple chess AI." `https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/`. Accessed on 24/05/21.