Object-Oriented Programming in C++
Final Project - Board Game - Chess

Thomas Ronayne

10305048

Department of Physics and Astronomy

The University of Manchester

Final Project Report

May 2021

Abstract

In this paper I describe how I implemented chess using an object-oriented approach in C++. It runs in the terminal and allows two players to play against each other. It demonstrates key object-oriented characteristics such as inheritance, encapsulation, and polymorphism. The program also makes use of advanced C++ features such as: unique pointers, standard library arrays, static data, namespaces, dynamics memory allocation, exception handling, header files and multiple *.cpp* files.

1. Introduction

For this project I have decided to code the classic game of chess. I believe it possesses the necessary complexity to make a good final project; it has pieces with vastly different possible moves, such as the knight's ability to jump over other pieces, or the pawn's ability to take diagonally, and it also has two players competing against each other. However, due to time constraints, I could not code a perfect recreation of chess, and have therefore omitted more special features such as: castling, en passant, pawn promotion, and draw conditions (such as stalemate).

2. Code design and implementation

2.1 Overview of code structure

The program is made up of 5 files: *chess_game.cpp*, *board.h*, *board.cpp*, *pieces.h*, and *pieces.cpp*. The file *chess_game.cpp* contains the main code where the game is played. The other files define the `board` class, and the `piece` class and its derivatives.

The chess pieces themselves are all derived from the abstract base class `piece`, and override the pure virtual `moves` function defined in the `piece` class, which determines which valid moves a piece can make.

The `board` class contains all the pieces that are currently in play in a standard library array of unique pointers. When a piece needs to be accessed, the board returns the relevant unique pointer by reference using the `board::get_piece` function. Following the principle of encapsulation, the board contains the pieces and their locations, which the pieces do not have access to. The pieces themselves are responsible only for calculating what moves they can make. When a piece needs to do this, the board is passed by reference, so the piece can check whether certain squares are accessible or not.

The game itself is played in the *chess_game.cpp* file, which creates an instance of `board` and creates the pieces in the standard chess arrangement. The board can be seen in Figure 1. The squares are referred to with standard algebraic notation (AN), e.g. ”a4” or ”b6”, as labelled on the chess board. The pieces are represented by their team (w or b for white or black respectively) followed by the first letter of their name – except for knight which is represented by an ”N” so as to be distinguished from the king; so a white bishop will be shown as ”wB”.

2.2 The use of unique pointers

The `board::add_piece` function has two parameters: a unique pointer to the `piece` base class, and a position. When a piece is added, this unique pointer is moved into the relevant position in the array that stores the pieces using `std::move`. In doing this we are giving control of the piece to the board, which then becomes the only way we can access the piece. When we want to move a piece from one square to another, we use the `board::move_piece` function, which also uses `std::move`. The advantage of *moving* pointers rather than copying them is we never lose track of a piece, whilst also making our code more intuitive (since moving a pointer corresponds to moving that piece).

Another advantage of unique pointers is that it is not necessary to track down and delete all of the pieces at the end of the game, because when a `std::unique_ptr` goes out of scope, the object it manages will be deleted automatically. Finally, using an array of pointers to store the pieces, rather than storing the pieces directly in the array, allows us to use a `nullptr` to represent an empty square, rather than having to create some `empty` derived class.

2.3 Piece movement

Since the pieces move in different ways, the way their moves are calculated is also different. The `pawn` class, for example, can only move forwards and must therefore check what team it is on, so it knows which direction to travel. It also needs to check whether there is an enemy piece diagonally forward one square to the left or right, before it can move there. Finally it must check whether it is still on its starting line, as that will allow it to move forwards 2 squares instead of 1. Contrast this with the `queen` class, which can travel any number of squares in 1 of 8 directions. To calculate *these* moves, the squares along each of these directions are checked one-by-one from the queen outwards until the edge of the board or another piece is reached.

These move calculations are performed by passing the `board` by reference to the `piece`, allowing it to check the state of certain squares using three `board` member functions: `is_valid_square`, `is_empty`, and `contains_enemy`, which should all be self-explanatory.

When the user wants to move a piece, the move they input is checked against all the valid moves that piece can make, and then the `board::move_piece` function is used to actually move the piece (provided the move is valid).

2.4 The main game loop

As mentioned earlier, the game is played in the *chess_game.cpp* file. The game is controlled by two `while` loops: one that repeats for each turn, and one that repeats after every user command. The former loop is terminated whenever the user inputs ”end” (indicating the end of the game), and the latter when the user inputs ”pass” (indicating the end of their turn). The turns alternate between white and black (white has the first turn), and the game ends when one of the kings dies.

Exception handling is used in these loops to catch instances of `std::length_error`, which occurs when the user has entered insufficient parameters for the chosen command (for example typing only ”move a2”, when ”move” requires two coordinates: an initial and a final). These commands will be discussed in detail in the results section.

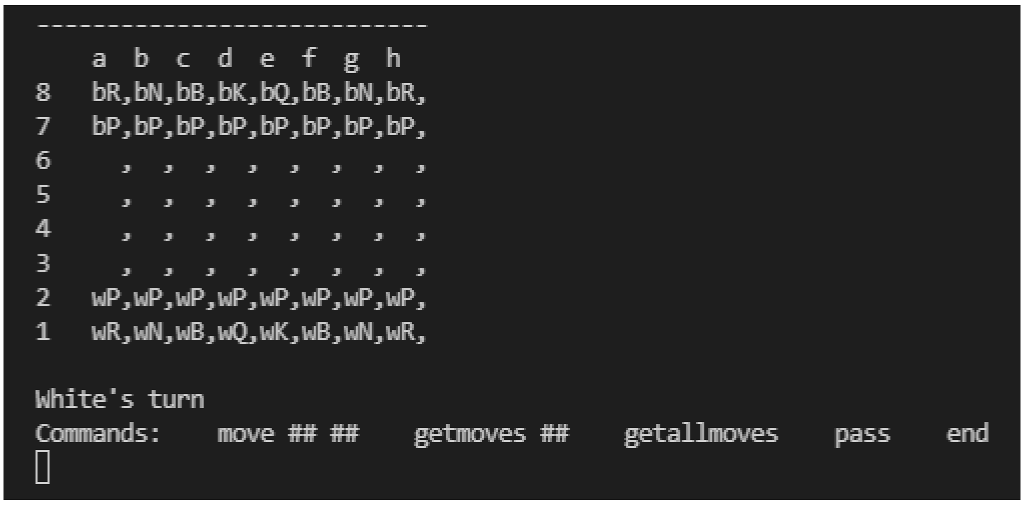
2.5 Other advanced features

The win condition for this chess game is the death of the opposing player’s king. The status of each players king is stored in two static boolean variables: `king::black_king_alive` and `king::white_king_alive`.

All the code for this project is wrapped in a `chess` namespace, allowing this code to be easily used within other programs if need be.

Coordinates are stored in a single `std::pair<size_t,size_t>`, since the x and y positions are so closely related, and are rarely used separately.

Using a `std::array` to store the pieces is just as fast as using a C-style array, but also comes with added benefits such as knowing its own size, being able to use STL-style iterator-based algorithms, and its ability to be passed to functions by value. These features could prove useful if this program was developed further (for example if varying the dimensions of the board was added as a feature).



```
--------------------------
    a  b  c  d  e  f  g  h
8   bR,bN,bB,bK,bQ,bB,bN,bR,
7   bP,bP,bP,bP,bP,bP,bP,bP,
6     ,  ,  ,  ,  ,  ,  ,  ,
5     ,  ,  ,  ,  ,  ,  ,  ,
4     ,  ,  ,  ,  ,  ,  ,  ,
3     ,  ,  ,  ,  ,  ,  ,  ,
2   wP,wP,wP,wP,wP,wP,wP,wP,
1   wR,wN,wB,wQ,wK,wB,wN,wR,

White's turn
Commands:    move ## ##    getmoves ##    getallmoves    pass    end
```

Figure 1. Screenshot of what the user sees at the beginning of each turn.

3. Results

The user has 5 commands at their disposal, which can be seen in Figure 1. These commands are interpreted by the program and executed if valid. The ”pass” and ”end” commands will end the current player’s turn (with the ”end” command also terminating the program), but the ”getmoves” and ”getallmoves” commands can be used multiple times without ending the current turn. The ”move” command, if successful, will also end the current player’s turn.

The ”move” command takes two positions as arguments and moves the piece at the first position to the second; for example ”move a2 a4” will move the piece at a2 to a4. After the user has inputted the desired move, the board will check it against the possible moves that that piece can make, and if the move is valid it will move the piece, ending the current player’s turn.

The ”getmoves” command takes a single position as an argument, and returns all squares that the piece on that square can move to; for example in Figure 1, ”getmoves a2” will return ”[a3 a4]” indicating that the pawn at a2 can move to a3 or a4. ”getallmoves” returns all moves that can be made by the player whose turn it is, an example of which can be seen in Figure 2. ”pass” will end the current turn, and ”end” will end the program.

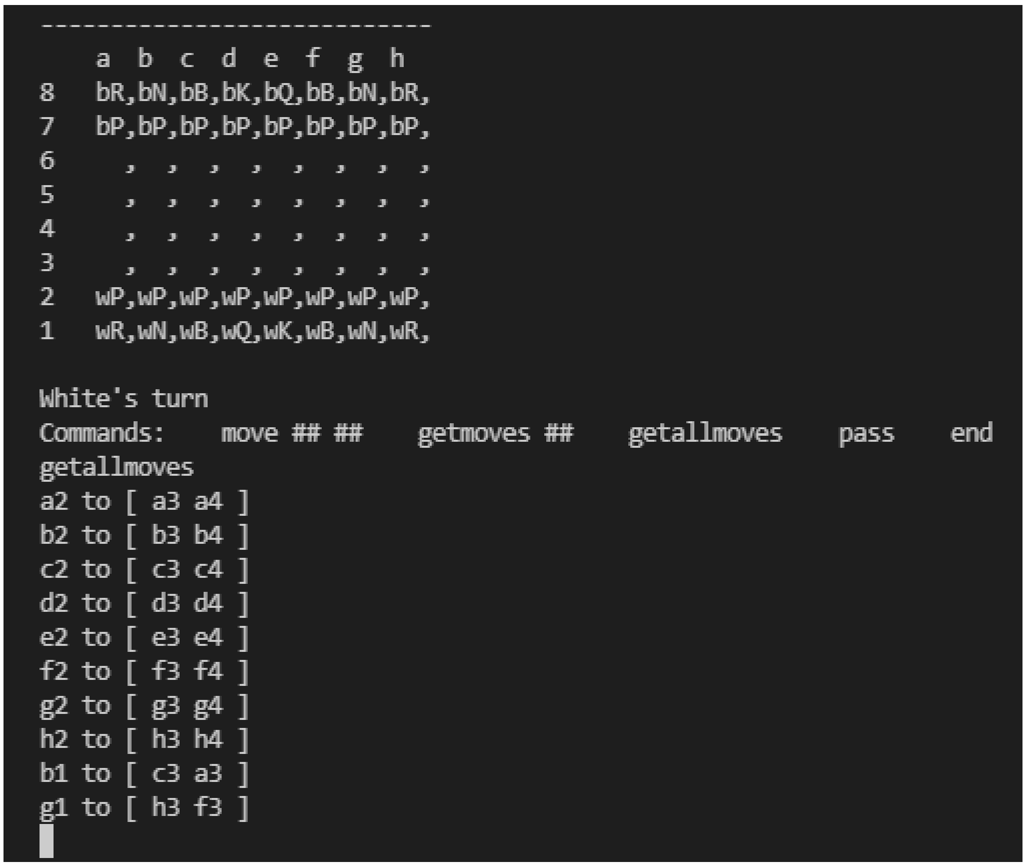
Specific error messages will be printed in the console if the user attempts to:

- move an empty square,
- move an enemy piece,
- move a piece to a square it cannot move to,
- get moves from an empty square,
- use invalid coordinates,
- enter less coordinates than required,
- use an unrecognised command.

If either player successfully takes the enemy king, the game will end and produce a corresponding victory message.

4. Discussion

The list of ways to improve this code is nearly endless, so I will start by highlighting issues in the code that I would change if I coded this again. Firstly, many member functions have



```
---------------------------
    a  b  c  d  e  f  g  h
8   bR,bN,bB,bK,bQ,bB,bN,bR,
7   bP,bP,bP,bP,bP,bP,bP,bP,
6     ,  ,  ,  ,  ,  ,  ,  ,
5     ,  ,  ,  ,  ,  ,  ,  ,
4     ,  ,  ,  ,  ,  ,  ,  ,
3     ,  ,  ,  ,  ,  ,  ,  ,
2   wP,wP,wP,wP,wP,wP,wP,wP,
1   wR,wN,wB,wQ,wK,wB,wN,wR,

White's turn
Commands:    move ## ##    getmoves ##    getallmoves    pass    end
getallmoves
a2 to [ a3 a4 ]
b2 to [ b3 b4 ]
c2 to [ c3 c4 ]
d2 to [ d3 d4 ]
e2 to [ e3 e4 ]
f2 to [ f3 f4 ]
g2 to [ g3 g4 ]
h2 to [ h3 h4 ]
b1 to [ c3 a3 ]
g1 to [ h3 f3 ]
```

Figure 2. Screenshot of the output of the ”getallmoves” command.

been defined with a `void` return type, such as the `board::move_piece` function. While this is perfectly valid, it might be better to give more functions like this an `int` return type, where the return value could be 0 if the function was successful, and -1 if not, which would help catch potential errors in the program more easily. For example, the `board::AN_to_coord` function *does* return a pair of -1s if the input is invalid, which is used in the main *chess_game.cpp* file to determine if the user has entered a bad coordinate.

Also, in some places I have used so-called ”magic numbers” such as 8 when converting between algebraic coordinates and number coordinates, or 64 when defining the length of the array to store the pieces. Ideally these numbers would be replaced by variables, which would make the code more readable, and help the code generalise.

The main *chess_game.cpp* file uses large `while` loops and a series of `if else` statements to interpret user input, making the file difficult to read. These responsibilities would more suitably carried out by some `game` class that could handle user commands, input validation, the turn system, and the win condition (all of which is currently done in *chess_game.cpp*). This would make the code easier to read and understand, and would also allow for more advanced functionality, such as playing the game multiple times without terminating the program.

As for missing functionality, advanced moves such as en passant or castling could be included, but the more important missing feature is check and checkmate calculation, which is an important part of chess (normally a game of chess ends when the king is checkmate, rather than when the king is taken). A more user-friendly GUI might also be desirable, and maybe even a computer player to play against the user, although this would be a large undertaking.

5. Conclusion

For my final project I coded the board game chess, using an object-oriented approach. The game runs in the terminal and consists of two players, black and white, who play against each other until one player successfully takes the other's king. The program includes a `board` class that stores the pieces and controls their movement, demonstrating encapsulation. The program also includes an abstract `piece` base class, derived classes for each of the chess pieces, and valid move calculation. The pieces derived from the `piece` base class demonstrate the C++ concept of inheritance and polymorphism, particularly in the overridden `moves` function.

The advanced features used include but are not limited to: unique pointers, standard library arrays, static data, namespaces, dynamics memory allocation, exception handling, header files and multiple .cpp files.

There are 2025 words in this document.