Two-Players Chess Game

Vlad-Marius Griguta

Final Report

Object-Oriented Programming in C++

School of Physics and Astronomy

The University of Manchester

May 2019

Abstract

A two-player chess game application was developed in the C++11 environment using an Object-Oriented code architecture. An abstract base class called ‘piece’ was used to define the functionalities of each chess piece, from which the chess pieces inherited their private attributes (colour, current square and number of moves) as well as the public methods used to act on the pieces. A second class called ‘square’ was used to store the information retained in each square on the board. A cyclic relationship dependency was used to connect the ‘piece’ and ‘square’ objects and ensure continuous communication between these objects.

A second layer of interconnected classes was devised to implement the player and board related features, named ‘player’ and ‘board’ respectively. The class ‘player’ was used to retain all information collected from each player, namely its colour, name, remaining and captured pieces, and the methods required to act on these attributes. The class ‘board’ was used to store the information related to the current structure of the board in the form of a static self-attribute, to allow universal access for all classes.

The third hierarchical layer of the architecture contained the class ‘session’. This class was used to initialise one session the chess game and run the game through the methods implemented in the previous layers. A main function was used to control the hierarchical implementation of the game.

A terminal-based user interface was developed for the game. The successful operation is demonstrated throughout this report and is made available for testing through the source code attached.

1. Introduction

Chess is one of the oldest and most popular strategy games played throughout history. The game is thought to have derived from the Indian game called 'chaturanga' sometime before the 7th century AC [1]. Two players are required to play the game, and they both begin the game with a set 16 pieces each, differentiated by their colour (Figure 1). The player with white pieces moves one piece first, according to the allowed moves of each piece. Following on, the players alternate turns, moving one piece at once, with the objective to force the principal piece of their opponent (the King) into a checkmate[1].

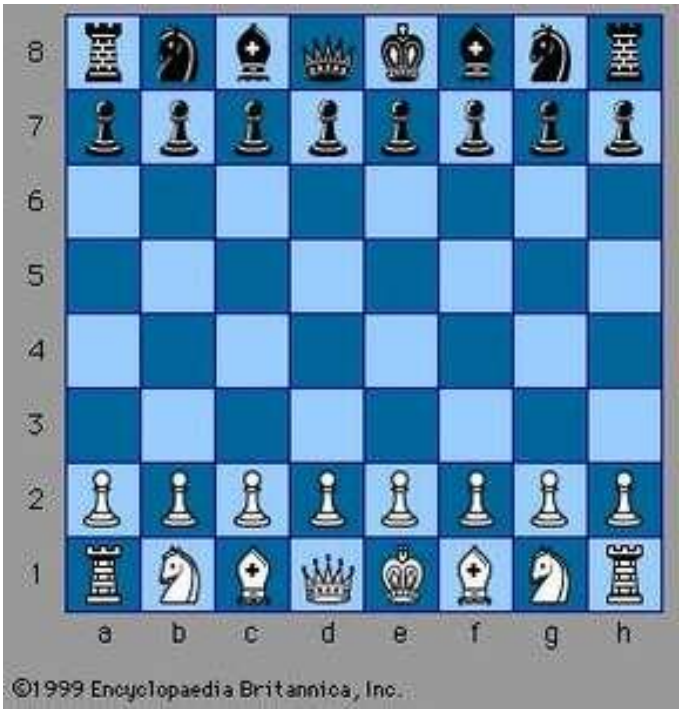

Figure 1 Layout of the Chess Board. [2]

Part of the reason why Chess has been popular over so many centuries is due to the complexity and the specialisation of each piece on the table. A lower-bound estimation of the game-tree complexity yields about 10^{120} possible games, derived from the average number of 10^3 possibilities for a pair of moves [3]. Besides the number of possible moves of each piece, additional complexity is introduced by the number of position-dependent rules that exist in the game (e.g. castling, pawn capturing, pawn promotion, etc). These rules will be exemplified in the following lines, where the exact mobility of each piece will be given.

A. Rooks

[1] A board layout in which it is impossible for the player to defend its king from being captured next turn.

The rooks are in the four corners of the chess table and each player starts with two rooks. They can move horizontally and vertically over an unrestricted number of squares, if their path is unobstructed. Rooks are allowed to capture an opposite piece that must be situated on an eligible square.

B. Knight

There are two knights allocated to each player and their initial positions are next to the rooks (i.e. *b1*, *g1*, *b8*, *g8*). Knights can move in an L-shape summing three squares, one in a direction and two in a perpendicular direction to the initial move. For example, in the initial format of the board, the knight on *b1* would only be allowed to move to *a3* and *c3*. Additionally, knights are the only pieces on the board that can move over obstructed paths. Knights can capture opposite pieces found on their destination square

C. Bishop

Each player starts with two bishops which are situated next to the knights, towards the centre of the board (i.e. *c1*, *f1*, *c8*, *f8*). Bishops can move on the unobstructed diagonals that intersect their current squares. Similar to the rooks, bishops can capture opposite pieces situated on eligible destination squares.

D. Queen

The queen is the most flexible piece on the board, and it combines the powers of a rook and a bishop. Queens begin the game on the same *d* vertical, namely on *d1* (White Queen) and *d8* (Black Queen).

E. King

The kings of both players are situated on the same vertical. They are allowed to move one square in any direction, if in the new position they are not threatened by any of the opposite pieces. If the king and one of the rooks have not previously moved and the path between them is clear, castling is allowed. When castling, the king moves two squares towards the rook and the rook moves on the opposite side of the king.

F. Pawn

In the initial board layout, the pawns populate all squares in the second and second to last horizontals. A pawn can only move one square towards the opposite side of the board. The exception is on the first move, when each pawn can move two squares. If an opposite piece is placed on a square diagonally in front of the pawn, in can be captured. However, no capture is allowed on a vertical move of the pawn.

2. Code Architecture

The architecture of the code follows closely the Object-Oriented programming (OOP) paradigm, a paradigm that uses objects as fundamental representation of data structures. In contrast to defining a data structure as a mere piece of information (e.g. database), OOP defines data structure as an instance (object) of a class which contains both data as well as the operations that can be applied on that data. The main features of the Object-Oriented programming paradigm are *abstraction* (picking up common features of data), *encapsulation* (join data with the allowed methods), *inheritance* (derive a class by inheriting all features from a base class) and *polymorphism* (multiple derived classes that inherit from one base class). These four features were used extensively within the code.

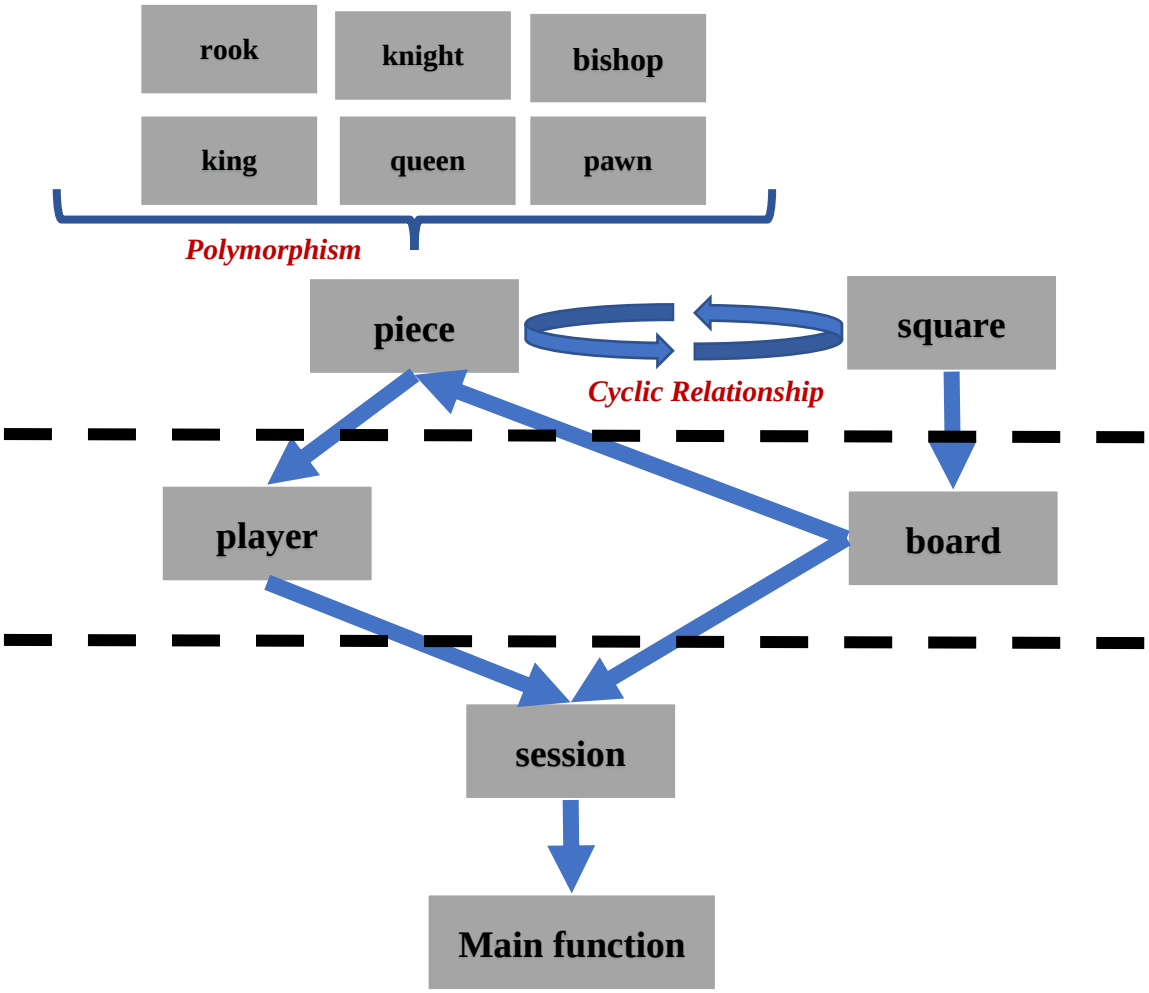

Figure 2 Schematic representation of the code architecture.

A schematic representation of the code architecture is shown in Figure 2. By making use of the OOP paradigm the workflow was divided in three stages, each containing a development and a testing session. The first stage was devised to constructing the building blocks of the chess game, the pieces. Each piece was constructed by inheritance from the base class 'piece'. To ensure consistency in the methods declared in each derived class, virtual and pure virtual functions were used. For example, to avoid declaring a separate class for the pawn, a generic virtual function for piece promotion was declared in the base class 'piece'. This function was set to return the same derived object it acted on and was only overridden in the pawn ('pon') class.

The second stage of the workflow implemented the rest of the data structure required to run a chess game, namely the board retaining the current stage of the game and the player data structure. To allow each object declared in the code to read the current stage of the game, the 'board' class was designed to contain a static pointer to itself that can be access through the static method 'access_board()'. The 'player' class was designed to store all information retained by a player in the game. The main functionality of this class is the method 'move()', which reads a player's move from the keyboard and implements it, checking for input inconsistencies.

The third and last stage in the workflow was designed to initialize and run the chess game with minimum functionality, using the classes developed at previous stages.

3 Code Implementation

This section is devised to describing in finer detail the implementation choices that were made in each of the three stages of the development process. To provide a more structured approach, each class design is treated separately.

Stage I

A. piece

The class piece is an abstract base class that is the building block for all pieces within the game. It takes as arguments the *colour* of the piece, a pointer to the *square* on which the piece is place at an instance in time and the *number of moves* that the pieces conducted. Functionalities for accessing and updating these attributes are implemented in the class. As a generic principle, all functions that were only devised for accessing the arguments were declared as constants.

Besides the access and update functions, three additional methods were implemented. The first method, in order of importance, was the method that checks whether the piece is allowed to move from the current square to a destination square, read as input. To account for the different rules of moving pieces on the board, this method was implemented as a pure virtual function and overridden in each of the derived pieces. The second method was a virtual function used for piece promotion. Since only pawns can be promoted, this method was only overridden in the ‘pon’ derived class. The third method was a pure virtual function that returns a string code for each unique piece. This was used for building the user interface in the ‘board’ class.

B. square

The class ‘square’ is the second fundamental data structure required in the chess game. The relation between squares and pieces data is a one-to-one relationship, meaning that each square on the board can retain at most one piece and piece in the game can be on at most one square. The attributes of the class are *x* and *y* position and a pointer to the occupying *piece*. The methods implemented on this class were the access and update methods that act on its attributes. Because the functionality that provides access to the occupying piece was also used for updating the piece later in the development, this access method was not declared as constant.

Stage II

C. Board

The ‘board’ class is the data structure that retains the positions of all pieces in the game at a given instance. The two attributes of the class are a pointer to a static instance of the *board* and a pointer to an 8x8 *array of squares*. To provide universal access to the board structure, a static method was implemented for accessing the static board instance. Additionally, access functionality was implemented for the array of squares.

Besides the access methods, the functionalities required to check if the path between two squares is clear were implemented. These functionalities are essential for checking if moves are allowed, and are used both in the class ‘player’ and in the class ‘piece’ from **Stage I**. A method implementing a graphical user interface was also developed within this class.

D. player

The ‘player’ class implements all methods available to a player during the chess game. The class takes as attributes the *colour*, *name*, *remaining pieces* and *captured pieces* of one player. Besides the accessing and updating functions, this class implements the method that reads moves from the keyboard, checks for input, board arrangement and rule-based inconsistencies and implements the moves. To better organize the development flow, the different checks required were implemented separate functionalities, in hierarchical coding structure. For example, the method used to flag a checking position was incorporated in the method used to flag checkmate, which was further incorporated in the methods used to try conducting one move. Iterators were used to iterate through the vector structures and the try – catch syntax was used to manage erroneous user inputs.

Stage III

E. session

The class ‘session’ is a high-level data structure that deals with the aggregate of all previous classes declared in the development process. It stores as attributes the two players and all their initial pieces and implements the method that initializes the chess game by arranging the pieces on the board and connecting each instance from the previous hierarchical layers to its attributes.

Functionalities used for storing instances of the game were implemented, however the development stage was not finalised for these methods. Further work should be focused on implementing these features in the code pipeline.

F. main function

The main function is the controlling body that implements the loops required to run the chess game for as many times as the user requests.

4 Conclusions and Future Work

The potential of using the Object-Oriented programming paradigm for the development of complex board games was demonstrated. An overview of the current functionalities of the chess game developed was presented. These include functionalities for reading the user input from the keyboard, drawing the current state of the game in terminal, moving pieces according to the rules, checking whether check and checkmate conditions are fulfilled, as well as functionalities implementing the more advanced rules of the game such as pawn promotion and castling.

To improve the C++ application, future work should focus on the implementation of a more attractive user interface. In that regard, third party software toolkits such as Qt [4] might facilitate the development of a high standard graphical interface. Additionally, certain functionalities that allow the user to reverse moves should be integrated in the code. Implementing *en-passant* rule, although not universally used, could be added to the scope of future development.

References

[1] C. Hugh, "Chess," *Encyclopedia Britannica*, vol. 6, pp. 93-106, 1911.

[2] A. E. Soltis, "Chess," [Online]. Available: https://www.britannica.com/topic/chess. [Accessed May 2019].

[3] C. Shannon, "Programming a Computer for Playing Chess," *Philosophical Magazine*, vol. 41, p. 314, 1950.

[4] "Qt Software," Qt, [Online]. Available: https://www.qt.io/developers/. [Accessed May 2019].