# Development of a game of object-oriented chess in C++

*Elliot Goodwin*

May 2019

## Abstract

This report discusses the development of the popular board-game Chess using the C++ standard library in an object-oriented framework, using VisualStudio 2013. The decisions made in the game design and implementation are discussed with reference to important features of object-oriented programming. Improvements that could be made to the game design with both the C++ standard library and beyond are briefly presented.

## 1 Introduction

This document introduces the game of chess written in an object-oriented (OO) framework in C++ . Object-oriented programming (OOP) is a style of programming that often makes a parallel with physical objects to describe objects by their defining features. Objects are able to interact between themselves and with other objects and are often reuse-able. The result of OOP is incredibly flexible software that can respond dynamically to the needs of an application at run time [1]. Chess is a complex game with many game pieces, each with many different legal moves and possible configurations, making it an interesting (and challenging) demonstration of the power of OOP.

C++ is a *class-based* programming language. Class-based programming relies on identifying important features of an object and storing them in a data structure known as a *class*. An instance of a class is referred to as an *object*. Objects in C++ have three important features: *encapsulation*, *inheritance* and *polymorphism*. Encapsulation refers to hiding data from view. For example, an object can have *private* and *public* members: the former are hidden from view, whilst the latter are available to the user at run-time through some interface. This adds a layer of safety to the data-structure. Inheritance is a term used when an object is defined in terms of some other object. Such an object is an instance of a *derived class*. Derived classes inherit the features of the base class that they have been given permission to access. The term "polymorphism" means "having many forms". In the context of OOP, this means that polymorphism is the property of a procedure being able to handle more than one type of object [1].

Inheritance has sometimes been used inappropriately, to share code between classes. However, the resulting *class hierarchy* means that derived classes can inherit unwanted properties from distant classes. Therefore inheritance is generally used if an object *is an* instance of another class. Generally, classes can be composed of other classes if they *have a* property defined by that type of object. An accepted use of inheritance in OOP is when defining objects in terms of an *abstract base class*. The abstract base class itself is never instantiated, but polymorphism can be exploited to write flexible code for a number of derived objects.

The outline of the remainder of this document is as follows. Section 2.1 introduces the notation specific to the game of chess, while Secs. 2.2 and 2.3 give an overview of the game design and implementation, including the class structure and user interface. Section 2.4 highlights some advanced features of C++ used in the program. A summary and discussion follows in Sec. 3. Explicit examples of the features of OOP introduced above will be highlighted throughout the document in *italics*.

Table 2.1: The correspondence between chess algebraic notation and chess piece. The character is used as the first character in the string used to input a player move.

| Algebraic notation | Chess piece |
|---|---|
| P | Pawn |
| C | Castle |
| N | Knight |
| B | Bishop |
| Q | Queen |
| K | King |

## 2 Game design

### 2.1 Chess algebraic notation

Chess algebraic notation is often used to label the movement of pieces across the chess board. It has many variations, and the version used in this program is therefore reported here for clarity. More information about chess algebraic notation can be found in Ref. [2].

This program accepts a player move in a 5 or 6 character string format. The first character indicates the piece a player is trying to move. The correspondence between algebraic notation and chess piece is given in Tab. 2.1. The following two characters are character $[A, H]$ and integer $[1, 8]$ coordinates indicating the square the chess piece is moving from. Similarly, the final two characters indicate the square being moved to. If a capture is being made during the player move, an "x" is used between the coordinate pairs. It should be noted that even moves with a valid capture will be deemed illegal if the "x" notation is not used in this program. An example of valid algebraic input is C$A5A8$. This indicates a castle owned by the player moving from the square $A5 \rightarrow A8$.

### 2.2 Program structure

The program is structured to allow instances of a number of classes to interact with each-other. The relationship between each class can be deduced by considering the game structure in plain English:

- Pieces: There are six distinct pieces in the game, each with different legal moves. Although each piece is distinct, they share the common property of being a chess piece. For example a Pawn *is a* piece. Each piece *has a* distinct set of legal moves.
- Players: The game *has* two players (e.g Human, Computer). Each player *has* 16 pieces and a colour to distinguish them from the opposing player.
- Squares: The game pieces are located on board squares. Squares can either be occupied or empty. The square therefore *has a* piece and the corresponding team colour that it represents.

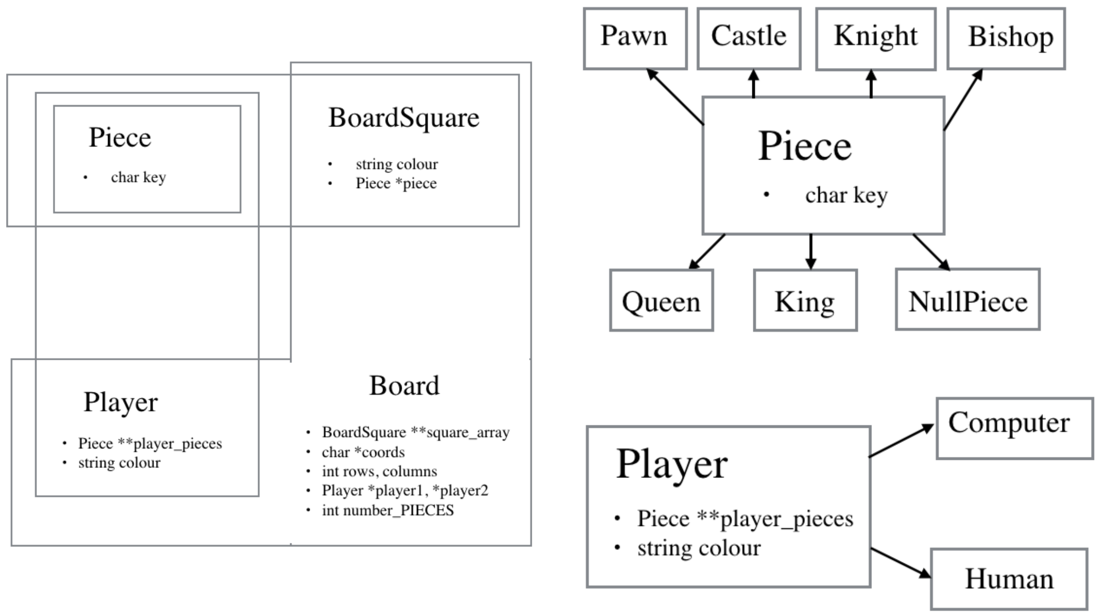


Figure 2.1: The class composition *(left)* and hierarchy (*right*), indicating the classes (boxed) and private members (bullets). Overlapping boxes indicate class *composition*. The arrows may be read as "has a derived class". Private members initialised as a pointer, for example to a base class object, are indicated with *. An array of pointers to base class objects are indicated with **.

- Board: The board *is an* array of $8 \times 8$ squares. The board *has* two players and their pieces that occupy these squares. Game progression can be thought of as decreasing the number of pieces that occupy squares on the board.

With reference to the introductory discussion of OOP in Sec. 1, these features can easily be converted in to the *class hierarchy* shown in Fig 2.1. It should be noted that in the program all classes in the hierarchy are separated in to separate header ( `.h` ) and source ( `.cpp` ) files. Additionally, all code written for the game of chess is grouped in a namespace " `Chess` ". This organisation of code greatly improves code readability and makes development more straightforward.

The design in Fig. 2.1 is chosen for several reasons. Firstly, since a "piece" itself is never instantiated (only distinct types of piece are), an *abstract base class* `Piece` can be defined, to which all distinct types of piece belong. All chess pieces are *derived* from this abstract base class. This means that flexible code can be written to handle a generic piece on the board, using *polymorphism*. Legal moves for each distinct player can be assigned by over-riding a virtual function written for the base class. A similar approach is used to allow derived classes `Human` and `Computer` from the base class `Player` . A `BoardSquare` class is defined to contain `Piece` s and `Player` s on the board.

The `Board` class contains all of these classes in a structure that enables a game of chess to be played. Additionally, the board provides an interface through which the user can interact with the program at run-time. A user interface is an important step in developing the game of chess; it ensures that all subsequent development can be visualised (and validated) immediately, and much of the complexity in the structure of the game can be hidden in the board design. The `Board` class contains a polymorphic array of pointers to 64 `BoardSquare`s. This design is chosen such that:

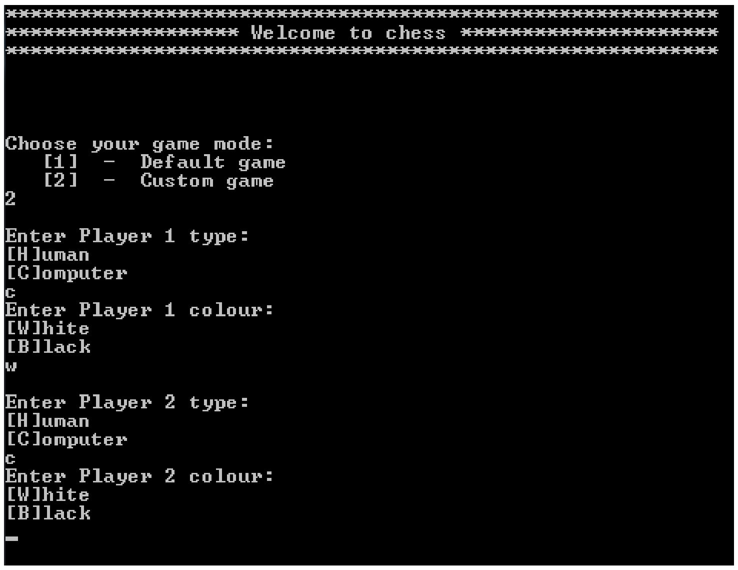
- The complexity of labelling co-ordinates within the `BoardSquare` array can be hidden in the board access functions. This aims to make programming the game rules at a later stage of development much more intuitive;
- Members of each `BoardSquare` can be easily manipulated, for example to set the board with pieces and allow them to move around its surface;
- *Polymorphism* can be exploited to display the properties of a `BoardSquare` in a dynamic way;
- The idea of direction on the board can be programmed, to help define legal moves for each piece;
- An array of letter co-ordinates desribing the $x$-axis of the board can be used to label squares; this feature makes the board output compatible with the chess algebraic notation defined in Sec. 2.1.

## 2.3 Game-engine

The implementation of the game of chess is powered by the game-engine. The engine stores a library of rules and has the ability to call game functions that makes use of the classes defined prior. The game-engine also uses an extensive error-handling interface to handle user input safely, by either re-prompting for user input or providing a graceful exit. It should be noted that since the game-engine is not an object, a stand-alone `.cpp` file that is able to instantiate the various classes needed for game-play is used, rather than a class structure.

Rules are implemented through a number of Boolean functions that assess the legality of a move; the functions take a string input of the form "C$D$6x$D$2" (see Sec. 2.1), and uses a stringstream to parse the $(x, y)$ co-ordinates of the initial and final positions on the board. This method is computationally inefficient, as each Boolean function constructs the stringstream and parses input individuallly. A less wasteful method would be parse the stringstream once and store each piece in a vector; vector elements could then be assigned to informatively named variables and passed to functions as parameters. Some examples of the jobs performed by the Boolean functions are to check for valid user input, valid piece moves (by calling `Piece` virtual functions), and assessing the occupancy of destination and intermediate squares.

If the move is permitted by chess rules, the game-engine calls `Board` access functions and utilises *move semantics* to swap elements in the array of `BoardSquares`, effectively moving pieces across the surface of the board. If a capture is indicated (using "x" notation), the


```
************************************************************
******************** Welcome to chess *********************
************************************************************



Choose your game mode:
   [1]  -  Default game
   [2]  -  Custom game
2

Enter Player 1 type:
[H]uman
[C]omputer
c
Enter Player 1 colour:
[W]hite
[B]lack
w

Enter Player 2 type:
[H]uman
[C]omputer
c
Enter Player 2 colour:
[W]hite
[B]lack
_
```

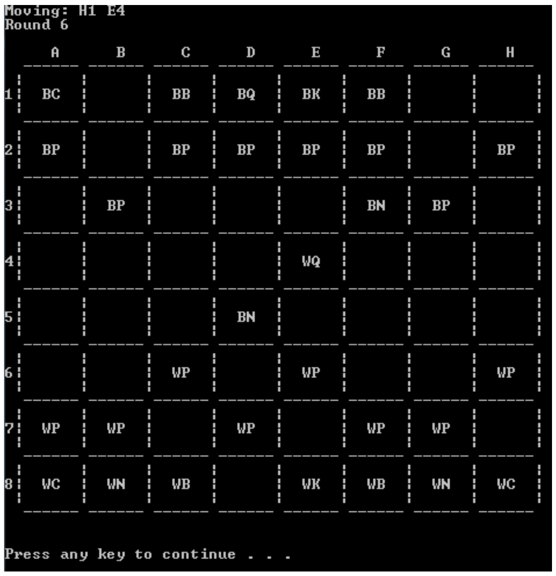
Figure 2.2: The game initialisation screen. This game is a computer-computer game.

function first decrements the number of soon-to-be captured pieces on the board, before setting the `BoardSquare` private member `piece` to type `NullPiece`.

## 2.4 Advanced features

A number of advanced C++ features have been used in the program development. For example the use of a thorough error-handling interface and advanced features of OOP such as inheritance and polymorphism have been discussed earlier in Sec. 2. Other examples include the use of dynamic memory allocation, which allows objects no longer needed by the application to be deleted, freeing up memory. An alternative approach would be to use smart pointers, which automatically free up memory; however since all memory is handled appropriately in this program, either approach is valid. Additionally, a help interface has been included to aid the user whilst the application is running. Upon inputting ‘h’ or ‘help’ to the program, the help interface is able to suggest moves available to the player. This feature computes all accessible moves to each piece owned by the player, and appends each move to a vector of strings, before printing the vector to the console. The vector is printed using a lambda function rather than a vector iterator, to make the print function aesthetically comparable to a traditional for loop.

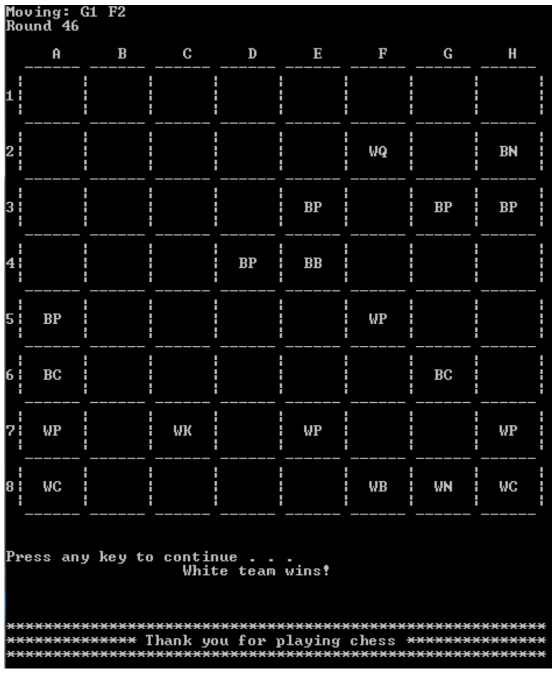
The help interface makes additional use of lambdas through its search engine feature. The search engine takes user input (for example, a type of piece or a board coordinate), and only displays moves accessible to the player that contain their search query. This feature is made possible by allowing a function to accept a ‘function-type’ parameter; the function-type parameter is declared using a template. A generic function can then be designed to cycle through vector elements (using iterators) and return the element if it matches the function criteria. The search engine function prompts the user to input their search criteria; the generic function described above then matches the search criteria to text in the elements of the vector containing all accessible



```
Moving: H1 E4
Round 6

     A      B      C      D      E      F      G      H
   ------ ------ ------ ------ ------ ------ ------ ------
1¦  BC  ¦      ¦  BB  ¦  BQ  ¦  BK  ¦  BB  ¦      ¦      ¦
   ------ ------ ------ ------ ------ ------ ------ ------
2¦  BP  ¦      ¦  BP  ¦  BP  ¦  BP  ¦  BP  ¦      ¦  BP  ¦
   ------ ------ ------ ------ ------ ------ ------ ------
3¦      ¦  BP  ¦      ¦      ¦      ¦  BN  ¦  BP  ¦      ¦
   ------ ------ ------ ------ ------ ------ ------ ------
4¦      ¦      ¦      ¦      ¦  WQ  ¦      ¦      ¦      ¦
   ------ ------ ------ ------ ------ ------ ------ ------
5¦      ¦      ¦      ¦  BN  ¦      ¦      ¦      ¦      ¦
   ------ ------ ------ ------ ------ ------ ------ ------
6¦      ¦      ¦  WP  ¦      ¦  WP  ¦      ¦      ¦  WP  ¦
   ------ ------ ------ ------ ------ ------ ------ ------
7¦  WP  ¦  WP  ¦      ¦  WP  ¦      ¦  WP  ¦  WP  ¦      ¦
   ------ ------ ------ ------ ------ ------ ------ ------
8¦  WC  ¦  WN  ¦  WB  ¦      ¦  WK  ¦  WB  ¦  WN  ¦  WC  ¦
   ------ ------ ------ ------ ------ ------ ------ ------

Press any key to continue . . .
```



```
Moving: G1 F2
Round 46

     A      B      C      D      E      F      G      H
   ------ ------ ------ ------ ------ ------ ------ ------
1¦      ¦      ¦      ¦      ¦      ¦      ¦      ¦      ¦
   ------ ------ ------ ------ ------ ------ ------ ------
2¦      ¦      ¦      ¦      ¦      ¦  WQ  ¦      ¦  BN  ¦
   ------ ------ ------ ------ ------ ------ ------ ------
3¦      ¦      ¦      ¦      ¦  BP  ¦      ¦  BP  ¦  BP  ¦
   ------ ------ ------ ------ ------ ------ ------ ------
4¦      ¦      ¦      ¦  BP  ¦  BB  ¦      ¦      ¦      ¦
   ------ ------ ------ ------ ------ ------ ------ ------
5¦  BP  ¦      ¦      ¦      ¦      ¦  WP  ¦      ¦      ¦
   ------ ------ ------ ------ ------ ------ ------ ------
6¦  BC  ¦      ¦      ¦      ¦      ¦      ¦  BC  ¦      ¦
   ------ ------ ------ ------ ------ ------ ------ ------
7¦  WP  ¦      ¦  WK  ¦      ¦  WP  ¦      ¦      ¦  WP  ¦
   ------ ------ ------ ------ ------ ------ ------ ------
8¦  WC  ¦      ¦      ¦      ¦      ¦  WB  ¦  WN  ¦  WC  ¦
   ------ ------ ------ ------ ------ ------ ------ ------

Press any key to continue . . .
                    White team wins!

*********************************************************
************** Thank you for playing chess ***************
*********************************************************
```

Figure 2.3: Representations of the board at two stages in the game. Game pieces are represented by their team-colour and key (see Tab. 2.1).

player moves, before printing them to the console. The search engine eliminates the need for multiple sort functions in the program, as they can instead be defined by the user at run-time.

Polymorphic `Player` classes are exploited to provide a computer opponent. The computer opponent is able to play chess by accessing random elements of the vector containing all available moves, before executing one; the pseudo-random number generator is seeded with a pointer to `TIME` to ensure the choice is actually random. A computer-computer game mode can be initialised in the application, which provides a powerful method of assessing the game capabilities.

Examples of the console output at different stages of the game are shown in Figs. 2.2 and 2.3.

## 3 Conclusion

This report has followed the development of a game of object-oriented chess in C++ by referencing the features of OOP used in decision making throughout. The code is structured in a logical manner, making use of header and source code files, in addition to use of the namespace “`Chess`” to group code used in game development together. The game is set up by instantiating four distinct classes related by a complex class hierarchy.

Several important features of OOP are demonstrated in these classes, for example: a complex class hierarchy, making use of the idea of composition, is used effectively; move semantics are used to efficiently update player moves on the board; abstract base classes with derived classes for each piece and player are implemented for flexible game initialisation; dynamic memory allocation is used in all constructors, destructors and abstract base classes to allow increased program flexibility and efficiency.

A substantial input handling interface that strictly checks for player input formats (including moves) is implemented through the game-engine. Input format errors are detected and prompted to be re-input with the aid of informative error messages, whereas more severe errors are handled with a graceful exit from the program. An aesthetic ASCII representation of the board is output to the console during game-play. Several advanced features of C++ are used through, for example, the combination of lambda functions and template for a generic function type to create a search engine within the game.

Improvements to the game using the C++ standard library could be possible with more efficient coding; the game-engine contains many instances of repeated code, for example when parsing move information from player input. Additionally, external libraries could be used to animate piece movement and provide a graphical user interface; geometric machine learning algorithms could be used to build a better computer opponent.

## References

[1] I. Craig, *The Interpretation of Object-Oriented Programming Languages.* Springer, 2nd ed., 2001.

[2] Wikipedia, "Algebraic notation (Chess)." `https://en.wikipedia.org/wiki/Algebraic_notation_(chess)`. Accessed: 18/04/19.