An implementation of Chess in C++ using Object Orientated Programming

Karan Mukhi
Student ID: 9655106

May 2019

Abstract

The board game chess was implemented in C++ using object-orientate programming. All rules of the game were implemented including, castling, pawn-promotion and en-passant, however the checkmate function has a bug. Advanced features in C++ 11, including lambda functions, exception handling and smart pointers, were used to produce efficient code.

1 Introduction

Chess is one of the classic board games, believed to be derived from the East Indian game "chaturanga" devised circa 500 CE, with the modern rules formalised in the 19th century. There are two players, each controlling sixteen pieces: a king, a queen, two rooks, two bishops, two knights and eight pawns. Players move the pieces around a board comprising of 64 squares in an 8 x 8 layout. Pieces move within constraints with distinct pieces having different constraints, or 'legal moves', pieces are captured when an opponent's piece moves to the square the piece is occupying. The aim of the game is to position one's pieces such that the opposition's king cannot escape capture on the next move, this is known as 'checkmate'. The structure of the game lends itself well to demonstrating the capabilities of object-orientated programming. All rules of the game were fully implemented, including castling, pawn promotion, en passant, check and checkmate. A method of displaying the board in an aesthetic manner was also implemented.

2 Code Design and Implementation

The program is comprised of four classes: game, player, board and square, and an abstract base class: piece, each of the distinct pieces are derived from this abstract base class. One of the most important features of the program was the movement functionality, this involved checking not just if the piece was moving under its constraints but also preventing movements to a square occupied by a piece of the same colour and preventing a player to move into check. Therefore the structure of the program was developed so that all objects in each of the classes involving game play, i.e. the board, square and piece classes, could access data from each other. This was done not only having container objects point to the objects that were held within them, e.g. the board holding pointers to each of the squares and the squares holding a pointer to the piece that occupies them; but also having pointers in the other direction, e.g. each piece held a pointer to the square it occupied and each square held a pointer to the board it was contained in. Thus each of the objects in these classes could access data from each of the other objects. The program architecture is shown in Figure 1. Each of the classes are described in more detail below, with a brief description of members and key member functions.

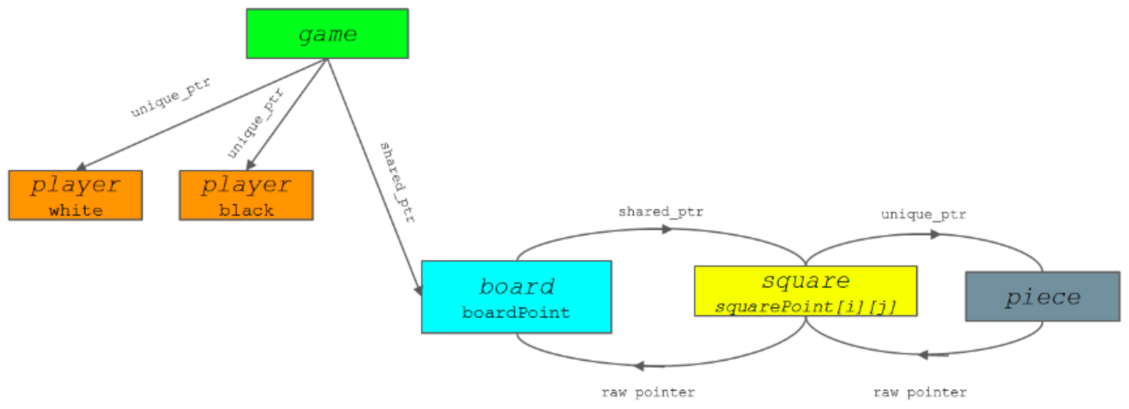


Figure 1: The program architecture, showing the classes can access data from each other.

2.1 Class Structure

2.1.1 `game`

This class is an overarching structure that contains the board and two player objects. It is the interface between the `player` objects and the `board` object. Its members include: `boardPoint` - a shared pointer to the board, `white` - a unique pointer to the white player and `black` - a unique pointer to the black player.

Member Functions:

- `game()` - constructs the game object by initialising the `board` shared pointer `boardPoint`, then filling it with squares and pieces. It then asks for input names of each of the players. One of these players is picked randomly to become white, that player's name is used to make the `player` unique pointer `white` while the other is used to make `black`
- `move(bool colour, moveCode move)` - colour defines the side attempting to make the move, move defines an intended move. These arguments are passed to the board to complete the move.
- `void play()` - this prompts users to input moves in alternating order. Exceptions are used to catch invalid input moves, both if the formatting is incorrect and if the move is illegal. This sequence continues while the `board` function `checkmate` returns false, if the `checkmate` function returns true the sequence breaks. The function then prints the name of the player who is not in checkmate as the winner, and returns. The protocol is shown in Figure 2.

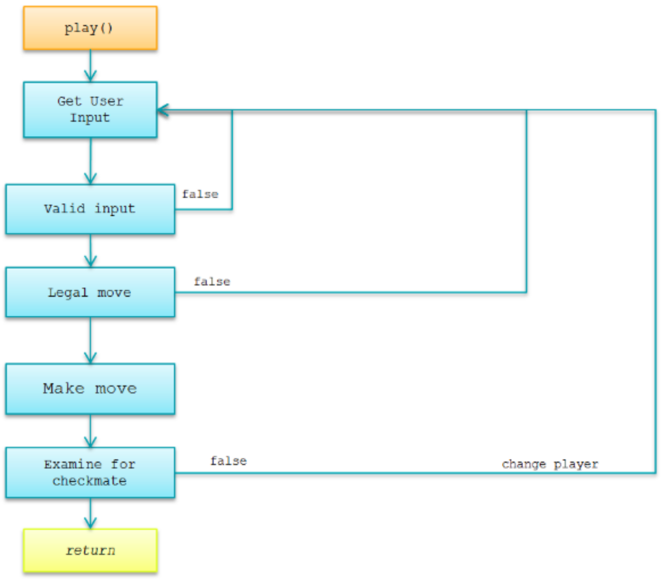


Figure 2: The `play()` function protocol

2.1.2 `player`

This class is used to interface with the users. Its members include: `colour` - a boolean representing what colour the player is and `name` - a string representing the name of the player. For interfacing with the users two other structures were defined: `squareCode` and `moveCode`. `squareCode` contains two integer members `col` and `row`, which are used to define the position of a square on the board. `moveCode` has two members of type `squareCode`: `initialSquare` and `finalSquare`, it represents the movement of a piece from `initialSquare` to `finalSquare`.

Member Functions:

- `getMove()` - prompts the player to input a move, the formatting of the move is validated, if there are errors an exception is thrown.

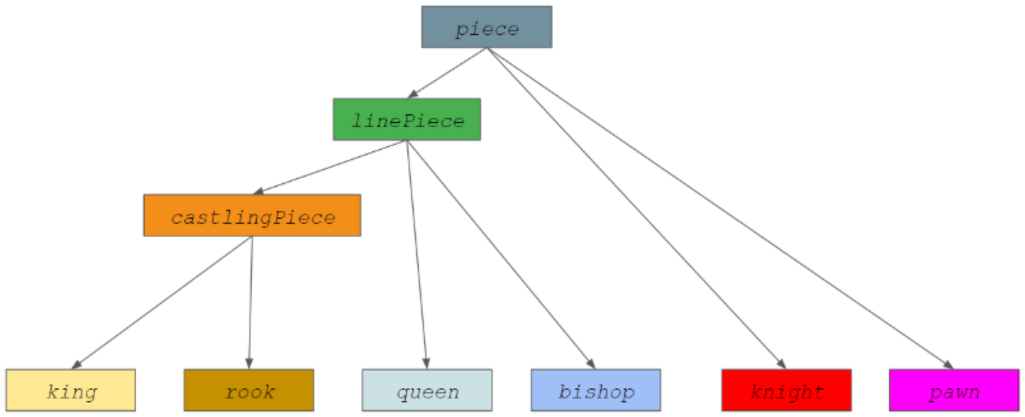


Figure 3: Polymorphism of `piece` abstract base class

2.1.3 `piece`

`piece` is the abstract base class from which all pieces on the board are derived. It was important to have all pieces derived from the same base class so that methods in the program could be written generically using base class pointers. Figure 3 shows the structure of how each piece is derived from the `piece` abstract base class. The `knight` and `pawn` piece classes were derived directly from this class. The `pawn` class had an extra member added, `bool justMoved`, and it also overrode the virtual functions: `switchMoved`, `getMoved` and `resetMoved`, which were all necessary when implementing en passant. A base class, `linePiece`, was derived for all pieces that can move in straight lines: the king, queen, rook and bishop. This meant `legalMove()` only needed to be implemented once for all these pieces; to differentiate the constraints between pieces `linePiece` had extra members all of boolean type: `straight`, `diagonal` and `distance`, defining whether the piece could move in straight lines, diagonal lines and the distance it could move, respectively. For example, a rook would have `straight` and `distance` set to `true` and `diagonal` set to false. These members were used by `legalMove()`, to tailor the function to that specific piece. One further base class was derived from `linePiece`, called `castlingPiece` from which only the `king` and `rook`, the pieces involved in castling, were derived. This class only contains one further variable a boolean `hasMoved` which keeps track of whether the piece has moved, for determining if castling is legal. The `castlingPiece` class also overrides the `switchMoved()` and `getMoved()` virtual functions. Its members include: `colour` - the colour of the piece, `side` - which side of the board the piece resides eg. 'K' or 'Q' (king or queen side) for non-pawn pieces, or a number between 1 to 8 for pawns, `type` - a code for the type of piece eg. 'K' for king and `homeSquare` - a raw pointer to the square the piece is occupying.

Member Functions:

- `virtual bool legalMove(moveCode move, bool noThrow)` - this is a pure virtual function. The function returns a boolean value depending on whether the piece can legally make a move defined by `move`, it was implemented separately for each derived class.

2.1.4 `board`

The `board` class is a container class for the `square` class. It stores all the squares that make up the board in a vector of vectors of squares. Its methods can access and manipulate data from each of the squares. It only includes one member `squarePoint` - a shared pointer a vector of vectors of squares, essentially a way of accessing all of the squares that make up the board.

Member Functions:

- Construction - this had to be done in two stages, as the board object needs to be initialised before a pointer to it can be shared with the squares. First the `board` constructor is called, this makes the shared `squarePoint` pointer. Once the board is initialised `void fillBoard()` is called, this function inputs eight vectors of eight squares, each vector of eight squares shall be referred to as a row, into `squarePoint` in the following order: a row containing white non-pawn pieces (formed by the `backRow()` function), a row containing white pawns (formed with `frontRow()`), four empty rows (`emptyRow()`), a row containing black pawns (again with `frontRow()`) and finally a row containing the black non-pawn pieces (`backRow()`). Lastly, it accesses each of the pieces on the board and sets their `homeSquare` member to a pointer to the square that the piece is occupying.

- `emptyRow(bool colour)` - this forms a row, by calling the `square` construct eight times, and using the `vector push_back()` function to insert them into the result `vector`. The square constructor argument `colour` is alternated between `true` and `false`, setting the square's colour to white or black respectively. The `pieceName` arguments are all set to "-" implying no piece is occupying the square. Finally `homeBoardInput` is set to `this` giving the square a pointer to the board it is held in. Other functions for forming rows with pieces on the squares were also written.

- `void print()` - this prints the squares out row by row, with a border around it. A lambda function is used to cycle through each vector in `squarePoint`, it captures the `int rowNumb` by address so it can be modified and hence the correct row number can be printed in the border. The `board` pointer, `this`, is also captured so the member function `printRow()` can be called on each row within the lambda's scope.

- `void printRow(const vector <square> printSquare, int rowNumb)` - the row `printSquare` is printed, with the `int rowNumb` printed in the border on either side. Each square is represented by a 14×7 grid of characters. To print a row the first line of characters in each square in the row is printed, the line is ended and the second line is printed. This continues until all seven lines that make up the row are printed.

2.1.5 `square`

This is the class that makes up the locations on the board. It is used to store data about these specific locations, and check if moves can be made from it. Its members include: `squareColour` - the colour of the square, `occupation` - a unique pointer to the piece that is occupying the square and `homeBoard` - a raw pointer to the board that the square is part of.

Member Functions:

- `move(bool colour, moveCode target)` - this function checks if the move from the square to a square defined by `target.finalSquare` is legal, it then completes the move if it is.

- `printLine(int lineNumber)` - this function prints one of the 7 lines that make up the visual representation of the square.

2.2 Algorithms

Many algorithms were implemented to ensure game play followed the rules of chess. For brevity only the `checkmate()` function is described.

2.2.1 `checkmate(bool colour)`

This is a member function of the `board` class, it is used to check if the game has concluded, it returns a boolean value after examining if one of the players is in checkmate. The scheme for doing this is as follows:

- Use the `check()` member for board. If this returns false the function returns false, else it continues.

- The function then moves the king to every square it can legally move to and examines whether the king is still in check after this move. If `check()` returns false for any of these configurations the function returns false, else it continues. The king is moved back to its original square.
- The function then uses a lambda function to scan the board to find pieces belonging to the player. Once it has found a piece another lambda function is used to scan and find squares the piece can legally move to. Once it finds a square it moves the piece there and examines if the king is still in check. If the piece successfully blocks the check the function returns false. Once the function has checked if all the player's pieces could block the check, and found none, it returns true.

2.3 Advanced Features

2.3.1 Smart Pointers

Smart pointers were used when initialising all objects apart from the game object. Shared pointers were used for the board and square objects, as these had to have their pointers held in more than one location, see Figure 1. Raw pointers were used to access higher level objects, Eg. square → board. Ideally weak pointers would have been used, however `weak_from_this` functionality has not been implemented in C++ 11. The player and piece objects are only held in one location so unique pointers were used for them. The piece unique pointers had to be moved when the piece was being moved, this was done using the move function of unique pointers

Listing 1: Moving Unique pointers

```
void setPiece(std::unique_ptr<piece> & newPiece){occupation = std::move(
    ↪ newPiece);}
```

2.3.2 Exceptions

Exceptions were used to great effect when validating user inputs. They were used to check if an input was correctly formatted, if the input move was legal and for pawn promotion. User inputs and moves are wrapped in the the try statement:

Listing 2: Try statement

```
try{
   if(turn){
      std::cout << white->getName();
      move = white->getMove();
   }
   else{
      std::cout << black->getName();
      move = black->getMove();
   }

   this->move(turn, move);
   break;
}
```

`getMove()` will throw error flags if the user input is wrongly formatted, whilst move() will throw error flags if the moves are illegal these are caught to give tailored error messages, for example when trying to move into check:

Listing 3: Example catch statement

```
if(errorFlag == 10) {std::cout << "Invalid move: You are in check you cannot
    ↪ make this move\n";}
```

2.3.3 Lambda Functions

Lambda functions were used to iterate through the board and perform a number of different functions. For example, the board member function, `print()` used a lambda function to print each row of the board.

Listing 4: Example lambda function

```
int rowNumb(0);
    for_each(squarePoint->rbegin(), squarePoint->rend(), [&rowNumb, this] (const std::vector<square> & row){
        rowNumb++;
        printRow(row, rowNumb);
    });
```

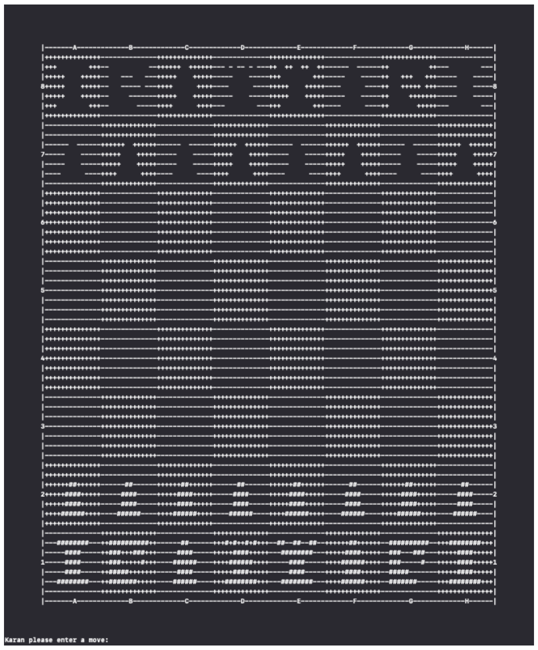
in this function the lambda captures `rowNumb` by reference, allowing it to be altered within the lambda. This was necessary as each row had to have the correct row number printed alongside it. `this` was captured so that `printRow()` could access its data.

2.3.4 Header and CPP files

The class definitions were split between four header files: game, player and piece each had separate files, whilst square and board were defined in the same file. This was required as the member functions in each class required member functions from the other class. The functions were implemented in three CPP files, one for game, one for player and one for board, square and piece.

3 Results

As far as it has been tested the program runs smoothly with all rules and functions implemented correctly apart from one bug in the checkmate function - it works in some situations - however it does not return true when it should be doing so in other situations, this is not a problem with the class structure, simply an issue with the implementation of that specific function. Piece movement was tested extensively and showed no bugs. The code was also checked for memory leaks, nothing was found. The board was printing in a visually appealing way as shown in Figure 4.



```
Karan please enter a move:
```

Figure 4: The visual representation of the board

4 Conclusion

Debugging functions could have been written to test and debug the more advanced functions like `checkmate`. A lot of time when into configuring the board in such a way to test this function. Friend functions could also have been used for moving pieces around; if these were declared within the board and square classes they would have access to data throughout the board without the need for the squares to contain pointers to the board. Advanced features in C++ 11, led to efficient code being written, especially using lambda functions in the `checkmate` and `print()` functions, and using exceptions to validate user inputs. The code could be developed to save and load games. The player class could be written so that users could play against the computer. Using object-orientated programming meant the code could be written far more easily than by using functional programming.

Word Count: 2500