C++ Object Oriented Programming: Chess

Owen Raymond

9324292

School of Physics and Astronomy

The University of Manchester

BSc in Physics

September 2015 - June 2018

Abstract

A simple chess engine has been developed using C++ object oriented programming techniques. The program allows for users to play a rudimentary two player game of chess. Methods used to achieve functionality are discussed and improvements to the game are recommended.

1. Introduction

Chess is a board game that is ubiquitous in popular culture which is believed to have originated in India in the 6th century [1]. It is a strategic board game where two players control their own set of sixteen pieces. The most important piece that each player owns is a king. Using logic the players take it in turns to move one piece as they attempt to outmanoeuvre each other and checkmate the opponent's king.

Both programming and chess are complex intellectual activities requiring creativity and logical reasoning. What to a human mind seems a simple step of logic demands explicit and extensive instruction to a computer which is lacking autonomy. Developing a two player game of chess presents a considerable challenge as well as providing a base for further development of an artificial intelligent (A.I.) player. The first computer programmed for playing chess in the 1950s was a machine named "Turbochamp" using an algorithm written by Alan Turing, who was a pioneer of artificial intelligence [2].

The purpose of this project was to create a functional rudimentary two player game of chess that makes use of the extensive features available in C++. This report will briefly outline the design and implementation of the objects used to form the game before discussing potential improvements.

2. Code Design and Implementation

The approach taken was to develop chess in four stages:

1. Instantiating a chess board and outputting it to console
2. Updating the board with legal piece movements
3. Developing a win condition
4. Forming the game in the main function

Fig.1 shows a schematic of the classes created in this project and their key member data and functions, which can be used for reference.

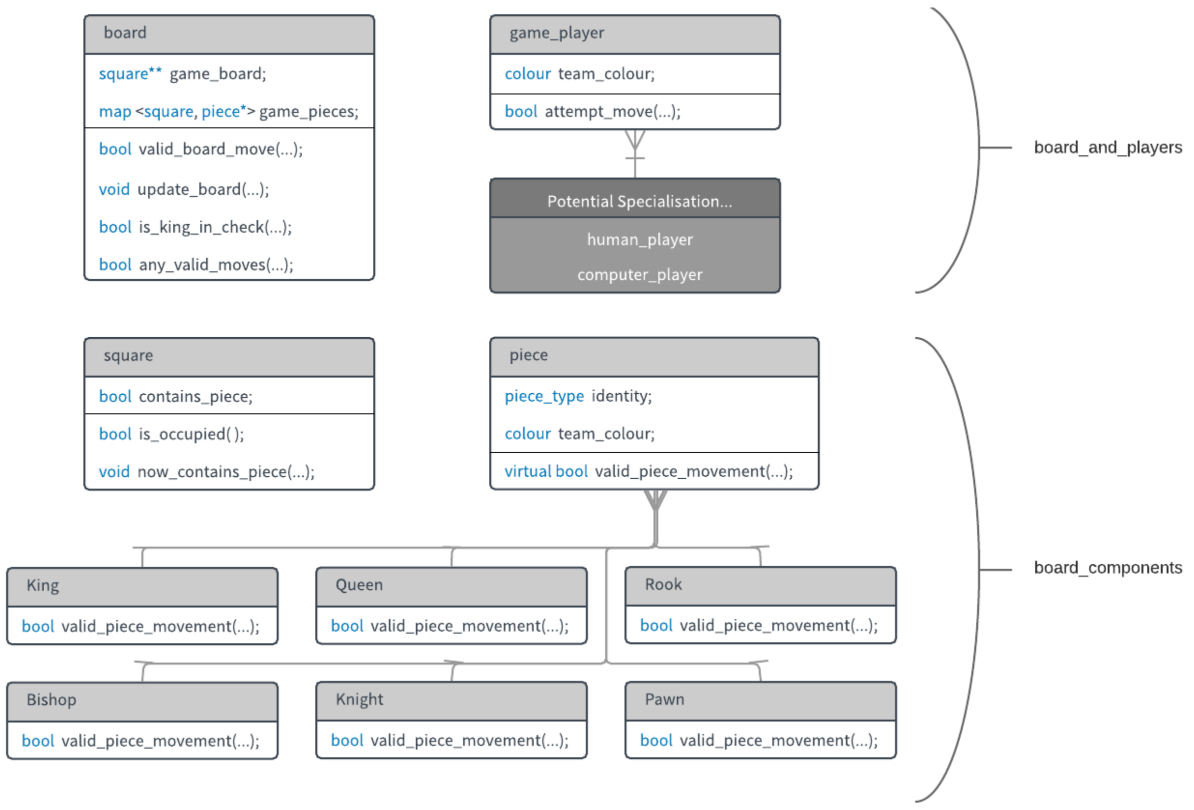

board
square** game_board;
map <square, piece*> game_pieces;
bool valid_board_move(...);
void update_board(...);
bool is_king_in_check(...);
bool any_valid_moves(...);
game_player
colour team_colour;
bool attempt_move(...);
Potential Specialisation...
human_player
computer_player
board_and_players
square
bool contains_piece;
bool is_occupied( );
void now_contains_piece(...);
piece
piece_type identity;
colour team_colour;
virtual bool valid_piece_movement(...);
board_components
King
bool valid_piece_movement(...);
Queen
bool valid_piece_movement(...);
Rook
bool valid_piece_movement(...);
Bishop
bool valid_piece_movement(...);
Knight
bool valid_piece_movement(...);
Pawn
bool valid_piece_movement(...);

Figure 1 A schematic detailing the key features and functions of the classes in the program. The King, Queen, Rook, Bishop, Knight and Pawn classes are all connected to the piece class to indicate they are derived from that base class

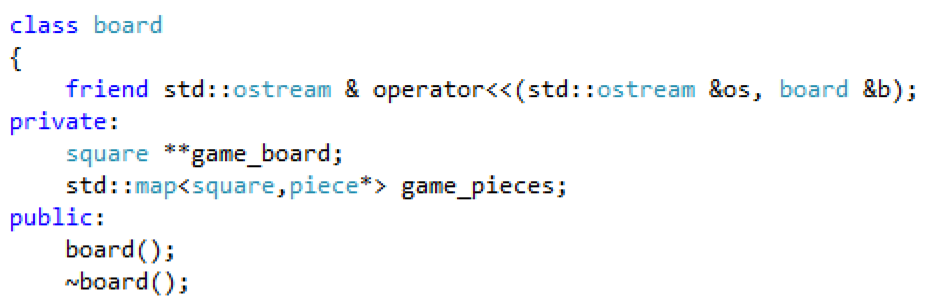
2.1 Instantiating a chess board

A chessboard consists of an 8x8 grid of squares containing 16 black pieces arranged on one side of the grid and 16 white pieces on the other. The fundamental components of a chess game are squares and pieces. The first files created for this project were the "board_components" header and cpp files, which contained classes for the square and piece objects.

The square class defines an object which contains the square positions on the board (row and column indices in the grid) and whether the square contains a piece.

A piece base class was defined from which all of the other pieces were derived. There are six piece varieties in chess which are pawn, rook, knight, bishop, queen and king. The only thing that differentiates one piece type from another is the way in which they move.

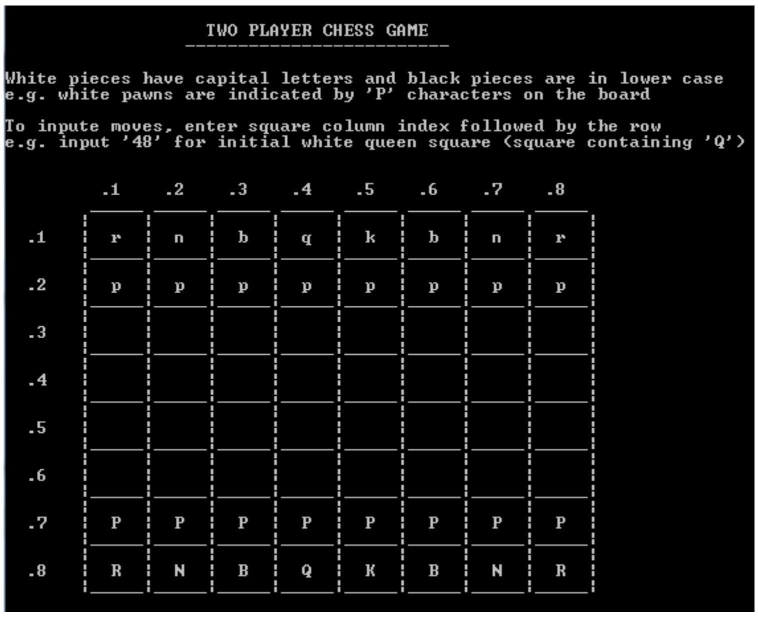
With the constituents of a chessboard defined, a board class was defined in a separate file which was named "board_and_players". The initial board class declarations are shown in Fig.2.



```cpp
class board
{
    friend std::ostream & operator<<(std::ostream &os, board &b);
private:
    square **game_board;
    std::map<square,piece*> game_pieces;
public:
    board();
    ~board();
```

Figure 2 Declarations that can be found in the “board” class

The private data in Fig.2 includes the “game_board” which becomes a 2D array of squares when a board object is constructed by board(). The other private data is the “game_pieces” map which is an associative container. In a game of chess all of the pieces belong to a square, but not all of the squares contain a piece at any one time. Therefore a pointer to each piece is stored in a map where a square (one from the game_board 2D array) acts as the key. The default board constructor was defined to initialise the 2D “game_board” array before creating the map of pieces using the “game_board” squares as the keys. The “<<” operator was overloaded to allow the chess board to be easily outputted to console. When the “<<” operator is called for a board object, the program loops through all of the squares in the 2D “game_board” array and checks if the square contains a piece, if it doesn’t a blank space is output and if it does a letter corresponding to the piece contained in the map with that square as the key is output. Depending on the piece being defined as white or black, the letter for the piece will be output in capital letters or lowercase respectively. The output of the board is shown in Fig.3.



```
                    TWO PLAYER CHESS GAME
                  -------------------------

White pieces have capital letters and black pieces are in lower case
e.g. white pawns are indicated by 'P' characters on the board

To inpute moves, enter square column index followed by the row
e.g. input '48' for initial white queen square (square containing 'Q')

            .1     .2     .3     .4     .5     .6     .7     .8
         _______________________________________________________
  .1     |  r   |  n   |  b   |  q   |  k   |  b   |  n   |  r   |
         |______|______|______|______|______|______|______|______|
  .2     |  p   |  p   |  p   |  p   |  p   |  p   |  p   |  p   |
         |______|______|______|______|______|______|______|______|
  .3     |      |      |      |      |      |      |      |      |
         |______|______|______|______|______|______|______|______|
  .4     |      |      |      |      |      |      |      |      |
         |______|______|______|______|______|______|______|______|
  .5     |      |      |      |      |      |      |      |      |
         |______|______|______|______|______|______|______|______|
  .6     |      |      |      |      |      |      |      |      |
         |______|______|______|______|______|______|______|______|
  .7     |  P   |  P   |  P   |  P   |  P   |  P   |  P   |  P   |
         |______|______|______|______|______|______|______|______|
  .8     |  R   |  N   |  B   |  Q   |  K   |  B   |  N   |  R   |
         |______|______|______|______|______|______|______|______|
```

Figure 3 A visual of an initial chessboard construction that is output to the console at the start of a game

2.2 Updating the board with piece movements

Physically moving a piece in a real game of chess is equivalent to updating the map structure in the board class. Moving a piece to an empty square involves deleting the old element in the map and updating the map to have a new element with the new square as a key. Taking a piece doesn't involve creating a new element, it just involves updating the element with the piece that has been taken with the new piece and deleting the element that contained the piece that was moved. Therefore a piece being taken corresponds to the "game_pieces" map shrinking in size by one element.

To make a move a player has to choose a move to input. The "game_player" class was defined in the "board_and_players" files to facilitate this. This class notably contains the players team colour as member data and an "attempt_move" function whose algorithm is illustrated in Fig.4.

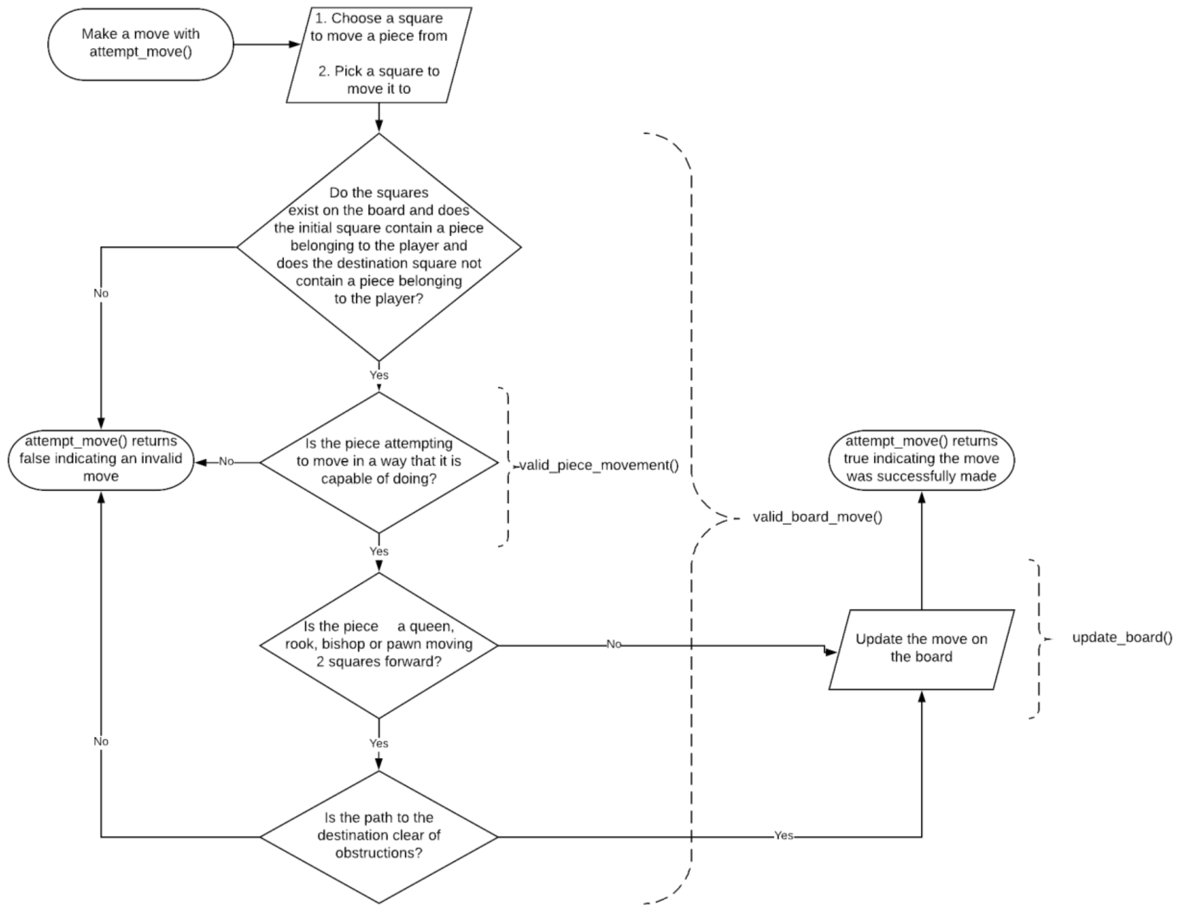

Make a move with attempt_move()
1. Choose a square to move a piece from
2. Pick a square to move it to
Do the squares exist on the board and does the initial square contain a piece belonging to the player and does the destination square not contain a piece belonging to the player?
No
Yes
Is the piece attempting to move in a way that it is capable of doing?
No
Yes
valid_piece_movement()
attempt_move() returns false indicating an invalid move
attempt_move() returns true indicating the move was successfully made
valid_board_move()
Is the piece a queen, rook, bishop or pawn moving 2 squares forward?
No
Yes
Update the move on the board
update_board()
No
Is the path to the destination clear of obstructions?
Yes

Figure 4 The attempt_move() function algorithm

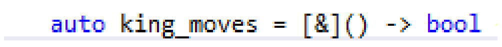
The Boolean game_player member function "attempt_move" prompts the player to input a move by specifying the index numbers of the desired squares. The squares chosen are used as arguments for the Boolean board member function "valid_board_move". This checks if the input index numbers correspond to squares on the board, as well as checking that the initial square contains a piece belonging to the player and that the final square doesn't contain a piece belonging to the player. If any checks fail "valid_board_move" returns false, prompting "attempt_move" to return false indicating an invalid move. "valid_board_move" then checks if the piece is trying to move in a valid way by calling the piece member function "valid_piece_movement" which is overridden for each different piece, making use of polymorphism. "valid_board_move" then specifically checks if the piece being moved is a queen, rook, bishop or a pawn moving two squares forward. An additional check is required for these pieces to make sure that the path they are taking is not obstructed by any other piece. If all of these conditions are satisfied "valid_board_move" returns true and "attempt_move" then calls the board class function "update_board" to update the boards game_pieces map before returning true itself to indicate the move has been updated on the board.

2.3 Developing a win condition

The win condition involved implementing "check" and "checkmate" conditions. When a player's king is under threat of being taken by an enemy piece, the king is in "check" and when a player cannot make a move to prevent their king being taken then their king is in "checkmate" and the enemy player has won. Three board member functions were added to allow for these conditions to be recognised by the program. Those functions were "find_king_square", "is_king_in_check" and "any_valid_moves".

The "find_king_square" function iterates over the map of the pieces and finds the king for the black or white colour that is entered as the argument and returns it. This square can then be input as an argument into the Boolean "is_king_in_check" function. This function iterates through the map of pieces and finds all of the enemy pieces on the board and asks the "valid_board_move" function if any of those pieces can validly move to take the king. If "valid_board_move" returns true for any of the pieces to move to take the king, then "is_king_in_check" returns true and then "any_valid_moves" function can be called to check if there are any moves that can relieve the threat.

The "any_valid_moves" function creates two vectors: One is filled with the squares that contain all of the king's team pieces and the other with "threat squares". The threat squares include any piece that is directly threatening to take the king and, if the piece threatening the king is a queen, bishop or rook, the squares along the path of the threat piece to the king. The function checks for two conditions: Can the king move to get out of check or can any of the king's team move to at least one of threat squares to intercept the threat. The use of lambda functions is demonstrated within this function, mainly to practice using them as the function could have been formed without them. They are a powerful tool introduced in C++ 11 which can be used to define functions within functions, which cannot be done otherwise. The declaration of a lambda function used in the "any_valid_moves" function is shown in Fig.5.



```
auto king_moves = [&]() -> bool
```

Figure 5 A declaration of lambda found in the "any_valid_moves" function

The “[&]” allows for the lambda function to see any of the variables currently in scope, the “( )” brackets are empty as there were no arguments needed for this function and the “-> bool” indicates the function will return a Boolean value.

2.4 Forming the game

The main program, illustrated in the flowchart in Fig.6 implemented the game of chess by instantiating two game_player objects and one board object. The game executes as a do while loop, which in essence is play the game while checkmate hasn't been reached. Within that do while are two do while loops, one for each player, in which each player is asked to enter a move, using “attempt_move”, until they pick a valid move. The player will be repeatedly asked to enter a move until “attempt_move” returns true, which also indicates that the board has been updated with the move, and breaks the loop. The game is a cycle of each player entering a valid move until one of them reaches the condition of putting the opposition king in check mate, winning the game. Before either player is prompted to attempt a move the function “is_king_in_check” is called and if the king is in check, the game calls the “any_valid_moves” function to check if king is in checkmate or not.

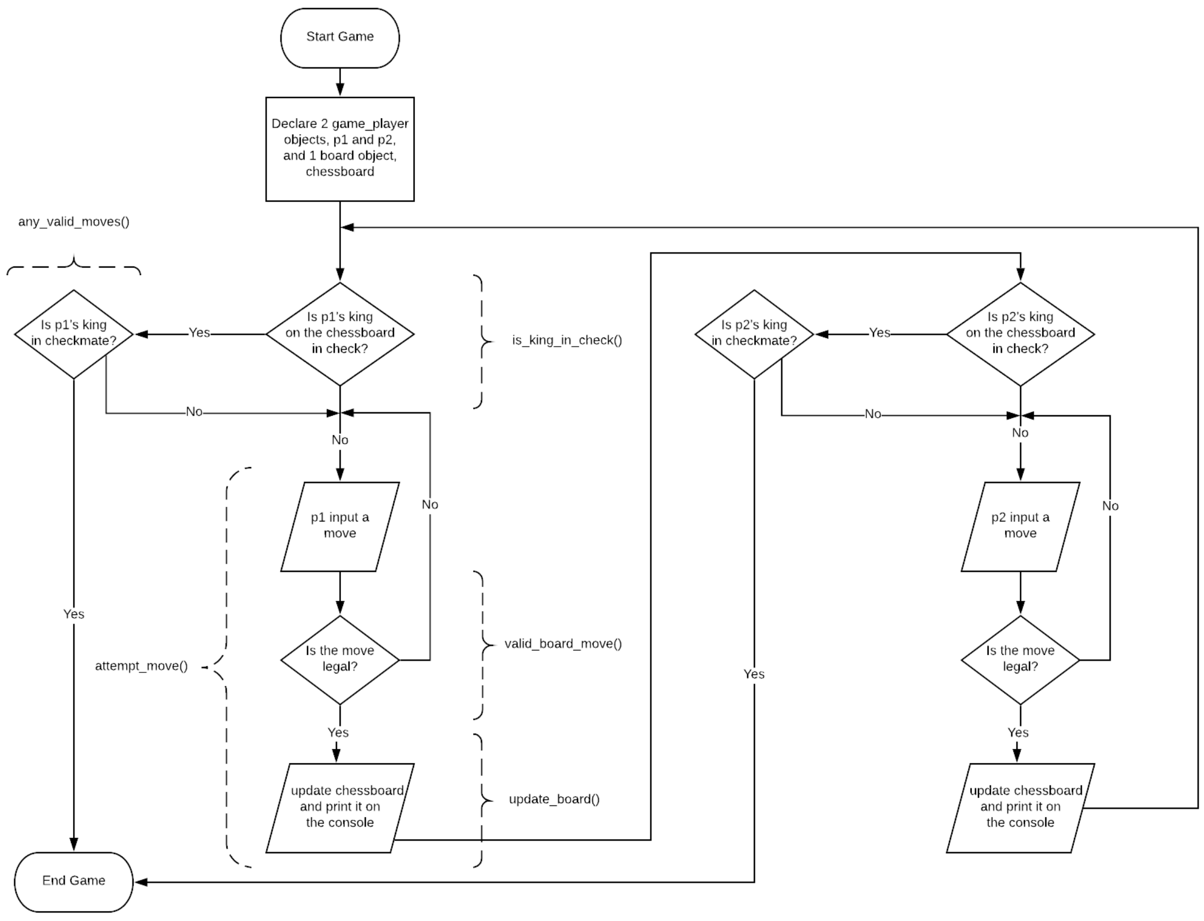
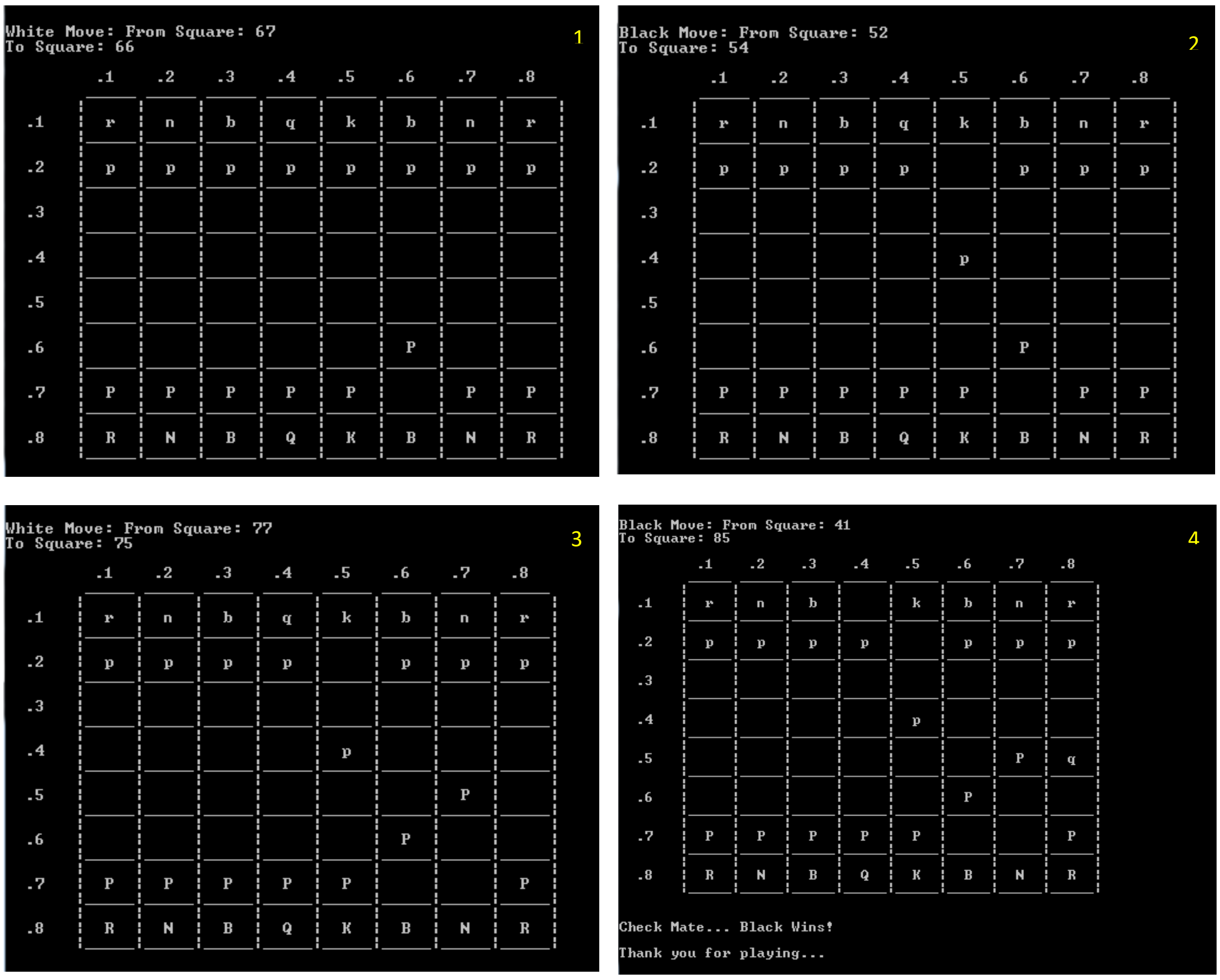

Start Game
Declare 2 game_player objects, p1 and p2, and 1 board object, chessboard
any_valid_moves()
Is p1's king in checkmate?
Yes
Is p1's king on the chessboard in check?
is_king_in_check()
No
No
p1 input a move
No
Yes
attempt_move()
Is the move legal?
valid_board_move()
Yes
update chessboard and print it on the console
update_board()
End Game
Is p2's king in checkmate?
Yes
Is p2's king on the chessboard in check?
No
No
p2 input a move
No
Yes
Is the move legal?
Yes
update chessboard and print it on the console

Figure 6 The chess game algorithm implemented in the “main” function of the program

3. Results

The game was tested thoroughly by moving each piece in a variety of ways and was found to be fully functional. The most important result to display is that checkmate is recognised by the program so that the game ends. This is shown in Fig.7.



1
```
White Move: From Square: 67
To Square: 66

        .1    .2    .3    .4    .5    .6    .7    .8
       _____ _____ _____ _____ _____ _____ _____ _____
 .1   |  r  |  n  |  b  |  q  |  k  |  b  |  n  |  r  |
 .2   |  p  |  p  |  p  |  p  |  p  |  p  |  p  |  p  |
 .3   |     |     |     |     |     |     |     |     |
 .4   |     |     |     |     |     |     |     |     |
 .5   |     |     |     |     |     |     |     |     |
 .6   |     |     |     |     |     |  P  |     |     |
 .7   |  P  |  P  |  P  |  P  |  P  |     |  P  |  P  |
 .8   |  R  |  N  |  B  |  Q  |  K  |  B  |  N  |  R  |
```

2
```
Black Move: From Square: 52
To Square: 54

        .1    .2    .3    .4    .5    .6    .7    .8
       _____ _____ _____ _____ _____ _____ _____ _____
 .1   |  r  |  n  |  b  |  q  |  k  |  b  |  n  |  r  |
 .2   |  p  |  p  |  p  |  p  |     |  p  |  p  |  p  |
 .3   |     |     |     |     |     |     |     |     |
 .4   |     |     |     |     |  p  |     |     |     |
 .5   |     |     |     |     |     |     |     |     |
 .6   |     |     |     |     |     |  P  |     |     |
 .7   |  P  |  P  |  P  |  P  |  P  |     |  P  |  P  |
 .8   |  R  |  N  |  B  |  Q  |  K  |  B  |  N  |  R  |
```

3
```
White Move: From Square: 77
To Square: 75

        .1    .2    .3    .4    .5    .6    .7    .8
       _____ _____ _____ _____ _____ _____ _____ _____
 .1   |  r  |  n  |  b  |  q  |  k  |  b  |  n  |  r  |
 .2   |  p  |  p  |  p  |  p  |     |  p  |  p  |  p  |
 .3   |     |     |     |     |     |     |     |     |
 .4   |     |     |     |     |  p  |     |     |     |
 .5   |     |     |     |     |     |     |  P  |     |
 .6   |     |     |     |     |     |  P  |     |     |
 .7   |  P  |  P  |  P  |  P  |  P  |     |     |  P  |
 .8   |  R  |  N  |  B  |  Q  |  K  |  B  |  N  |  R  |
```

4
```
Black Move: From Square: 41
To Square: 85

        .1    .2    .3    .4    .5    .6    .7    .8
       _____ _____ _____ _____ _____ _____ _____ _____
 .1   |  r  |  n  |  b  |     |  k  |  b  |  n  |  r  |
 .2   |  p  |  p  |  p  |  p  |     |  p  |  p  |  p  |
 .3   |     |     |     |     |     |     |     |     |
 .4   |     |     |     |     |  p  |     |     |     |
 .5   |     |     |     |     |     |     |  P  |  q  |
 .6   |     |     |     |     |     |  P  |     |     |
 .7   |  P  |  P  |  P  |  P  |  P  |     |     |  P  |
 .8   |  R  |  N  |  B  |  Q  |  K  |  B  |  N  |  R  |

Check Mate... Black Wins!
Thank you for playing...
```

Figure 7 A diagram showing four consecutive moves, indicated by the numbers in the top right corner of each board, that lead to the checkmate condition and successfully end the game

4. Discussion

The program functions well as a two player game but lacks the feature of having an A.I. to play against. An A.I. was under development with the basic logic to take a piece if it was possible, and to move a pawn if it could not take a piece. The program would have crashed if it could not do either of these things, but this was just an initial design. The A.I. class was designed with a "decide_move" member function to implement this. Within this function was a lambda function that returned the square containing the opponent piece of highest value that the piece could move to take. The lambda function was going to be used as the task in threads created for each piece in the A.I.'s team, to display the multithreading feature that was introduced in C++11. The aim was to allow all of the pieces in the A.I.'s team to search for the highest value opponent piece that they could take simultaneously. However, an issue with this implementation was quickly realised.

Threads require a name which means that the program needs to know how many threads will run before compilation. As the game progresses and pieces from the A.I.'s team are taken, fewer threads are required for the A.I. to decide a move. The only inelegant solution to this problem is to create sixteen conditions for the sixteen different possible amount of pieces the computer can control during different stages of the game. The development of the A.I. was abandoned at this stage due to this reason and a lack of time to work on it. Though it was an ambition to create an A.I., it proved to be a challenge that was not achieved in the time taken to create this project.

In regards to other features of C++ that were not included, there is potential for use of special pointers in the program. This was not achieved in the time taken to create the project as introducing them in place of the piece pointers in the "game_pieces" map compromised the code that was already working. There may also be potential for exceptions to be used, but no use was apparent during the development of the project.

In terms of functionality there are many subtle aspects to chess that were not displayed in this program, for example castling and en passant. These subtle moves would improve the game but were not necessary for the objective of this project, which was to display features of object oriented C++. A stalemate win condition should also have been included for completeness.

5. Conclusion

A functional rudimentary object-oriented version of chess was created in C++ which displays the abstraction, encapsulation, inheritance and polymorphism pillars of C++ successfully. Other powerful features of C++ were demonstrated such as the use of the associative map container, the introduction of lambda functions within functions and the separation of code into header files. The code was well organised lending itself to be easily developed by the original coder or another programmer in the future. Future developments could focus on developing an A.I. which could utilise the powerful multithreading feature for efficiency.

References

[1] H.J.R. Murray, A History of Chess: The Original 1913 Edition, 2015

[2] E. D. Reilly, Milestones in Computer Science and Information Technology, 2003

Word count: 2427 for whole document