Adaptation of the boardgame chess in C++ using object-orientated principles

Zhuang Gunter

9698981

School of Physics and Astronomy

The University of Manchester

PHYS30762

May 2019

Abstract

A simple two player turn based chess game was created for this project, which includes all the move sets for all chess pieces. The aim of this project was to implement the key features of the board game chess into a C++ programme, using concepts such as encapsulation, composition, inheritance and polymorphism. These concepts were demonstrated by class encapsulation using separate header files, piece class inheritance and the use of virtual functions.

1. Introduction

In classed-based programming, an object can refer to a function, a variable or a structure, usually containing data. Object-oriented programming (OOP) is a style of programming based around the object, in which the functions associated with the object can modify data within the class. Objects are created and made to interact with each other to form the final programme[1]. There are four main pillars in OOP:

- Encapsulation and Abstraction– creating classes containing data and operations in private/protected and public parts of the class respectively. Operations can be called by clients openly whilst the data stays inaccessible.
- Inheritance – by making the base class public in a derived class, the derived class inherits all the properties of the base class, enabling it to modify the base class data and operations.
- Polymorphism – the ability to use the same function name with different actions on different object types, usually achieved using function/operator overloading and function templates.

The aim of the project is to create a two player turn based chess programme, with functional move sets for all the pieces. The pieces are split into two colours, black and white, each player has control over 16 pieces, including a king, a queen, two rooks, two bishops, two knights and eight pawns. Player's turn ends when a valid move is entered, and the game terminates when one of the kings is captured.

2. Code Design & Implementations

2.1 Header and source files

Header files are used to make the code look concise and clear. There are four header files in the programme: chess, square, board and player.

Chess file includes the abstract base class named pieces, which is the parent class for all individual chess piece classes. It includes several virtual functions which are overridden by functions in the subclasses based on the allowed moves of the piece. Player file includes basic information about the two players.

Square file contains the class square, representing each square on the chess board. The square class contains a pointer to the piece residing on the square, and coordinates to indicate its position within the board. Board file contains the board class, which has a 2D array of 8 by 8 squares representing the chessboard. All gameplay related functions are included in the board file, as most of them requires the chessboard as an input parameter.

To access an operation defined in a separate file, the header needs to be included on top of the current header file. This can be problematic as it runs the risk of redefining functions if the same header is included multiple times. This problem was solved by using *#pragma once*, to make sure the source file is only included once during a single compilation. Another problem which occurred during compilation is circular dependency, where two or more header files had to include the other headers as some of the operations require a class type defined elsewhere. This was resolved by using forward declarations.

2.2 Class operations and game mechanics

The main inheritance tree implemented in the programme is between the piece parent class and all chess pieces which are sub classes. The sub classes inherit instance variables from the piece class, such as position of the square, colour and value of the piece. They also each have an

overridden function name move_check(), which checks the user input move against the allowed moves for the piece and returns a Boolean.

2.2.1 Position validation for different pieces

Several functions were created in the piece class to check for a clear path vertically, horizontally and diagonally. One of the functions is displayed below in figure 2. These functions are key and are used multiple times in the move check of each pieces. All three functions are similar in design and asks for the same three parameters, the starting position and end position, which are created using user input, and the current gameboard. The change in coordinate is recorded compared to the move conditions (e.g. change in x needs to be zero for a vertical move, change in x and y needs to be the same for a diagonal move). Function then iterates through all the squares in-between checking if they are occupied, returning true if all of them are empty.

```
81    // Function to check for a clear path between 2 squares in the same diagonal
82    bool Pieces::Diagonal_check(Square& start_square, Square& finish_square, Board& gameboard) {
83
84        bool check = true;
85        // Compute the difference in the coordinates
86        int dx = finish_square.get_x() - start_square.get_x();
87        int dy = finish_square.get_y() - start_square.get_y();
88        // Set up the direction of travel of the piece
89        int x_direction = 1; int y_direction = 1;
90        if (dx < 0) { x_direction = -1;}
91        if (dy < 0) { y_direction = -1;}
92        // Check to see if the change in x equals to the change in y - Cannot be a diagonal move if these are not the same
93        if (abs(dx) != abs(dy)) { check = false; }
94        if (abs(dx) == abs(dy)) {
95            for (int i = 1; i < abs(dx); i++) {
96                if (gameboard.get_square(i*x_direction + start_square.get_x(), i*y_direction + start_square.get_y())->is_occupied())
97                { check = false;}
98            }
99        }
100       return check;
101   };
```

Figure 2: Code for function diagonal_check(), which checks the occupancy for all squares between the starting and ending position. A Boolean variable is defined at the start to be true, and if any of the conditions in the if loops are met, it becomes false. Note that the direction of travel also needs to be taken into consideration, as it is used in the for loop to make sure the loop starts the iteration on the square adjacent to the starting position.

The move sets for the rook, bishop and the queen were easy to implement, as the rook can move in straight lines, the bishop can only move in diagonal lines, and the queen is a combination of both the rook and the bishop. The movement of the king and knight are also trivial, the king can only move one square in any directions, and the knight can move to any squares where the sum of the change in horizontal position and vertical position equals to three.

The move set of the pawn is more complicated. It can move one square in the forward direction, two squares in the forward direction on the first move, and capture pieces in the squares diagonally in front of them. The direction of travel depends on the colour of the pawns as they can only travel in the forward direction. Extra if loops were added to the function to take care of all the possible scenarios.

2.2.2 Check and end game situation

When a player's king is under attack from any of the opponent's piece, it is said to be in check. Figure 3 illustrates the code layout for function used to check if the current player's king is in check. The function first finds the position of the current player's king, then it checks all the opponent's pieces left on the board to see if they can advance to the position of the king. When the player is in check, the next move is only valid if king can no longer be captured. This can be achieved in several ways, by moving the king to a square where its safe, or by capturing the checking piece, or by placing an intermediate piece in the line of attack. This check needs to be done every time before the player makes a move.

In conventional chess, a player cannot make a move which will result in himself being in check. If the player is not in check but has no legal moves left to make, the game ends with a draw known as stalemate. However, the draw scenario was not implemented into the game, instead

the check function is called at the end of the input validation function. When the player is in check and makes a move, which does not free him from it, the function reminds the player of the consequences and ask if the move should be executed.

```
163     // Function to check to see if the player is in check
164     bool Board::check_in_check(Board& gameboard, string player_colour) {
165         // Declare variables used
166         bool check = false;
167         string opponent_colour;
168         Square* king_position = nullptr; Square* attacking_piece;
169         if (player_colour == "W") { opponent_colour = "B";}
170         else { opponent_colour = "W"; }
171         // Find the position of the player's king
172         for (int i = 0; i < 8; i++) {
173             for (int j = 0; j < 8; j++) {
174                 if (gameboard.chessboard[i][j]->get_Piece() != nullptr &&
175                     gameboard.chessboard[i][j]->get_Piece()->get_colour() == player_colour &&
176                     gameboard.chessboard[i][j]->get_Piece()->get_piece_value() == 1000) {
177                     king_position = gameboard.chessboard[i][j];}}}
178         // Find all enemy pieces on the board, and check to see if any of them can ove to the king's position
179         for (int i = 0; i < 8; i++) {
180             for (int j = 0; j < 8; j++) {
181                 if (gameboard.chessboard[i][j]->get_Piece() != nullptr &&
182                     gameboard.chessboard[i][j]->get_Piece()->get_colour() == opponent_colour) {
183                     attacking_piece = gameboard.get_square(i, j);
184                     if (gameboard.chessboard[i][j]->get_Piece()->move_check(*attacking_piece,
185                         *king_position, gameboard)) {
186                         check = true;}}}}
187         return check;
188     };
```

Figure 3: Code for function check_in_check(), which takes the current player's colour and the gameboard as arguments.

2.3 Converting user input

The chessboard is an 2D array consisting of 64 squares, each square is made up of 6 horizontal and 3 vertical dashes. Each square contains a pointer to a piece class, which has sub-classes for each type of pieces. The rows on the board are labelled with numbers from 1-8, and the columns are labelled with letters from a-h. The programme asks for a string input in the format of a letter followed by a number, checks to see if the inputs are within the range of the board by comparing the inputs against the pre-defined limits, then converts the input into integers which are used as the elements of the 2D array.

```
12      // Define the limits of the board
13      #define x_min 'a'
14      #define x_max 'h'
15      #define y_min '1'
16      #define y_max '8'
17
18      // Function to valdate the user input as the correct format
19      bool validate_input(string input) {
20          bool valid_input = false;
21          // Check to see if the input is a special command
22          if (input == "score" || input == "moves" || input == "help") { valid_input = true; }
23          //Check the input is the correct length
24          else if (input.size() != 2) {
25              cerr << "Input is not in the correct format!" << endl;
26              cerr << "Please make sure the input consists only of letters and numbers (e.g. A3, e4)!" << endl;
27              cin.clear();}
28          // Check input is in bound of the board
29          else if (tolower(input.at(0)) < x_min || tolower(input.at(0)) > x_max ||
30              tolower(input.at(1)) < y_min || tolower(input.at(1)) > y_max) {
31              cerr << "Input is out of bounds!" << endl;
32              cerr << "Please enter a letter between a - h, and a number between 1-8!" << endl;
33              cin.clear();}
34          else { valid_input = true; }
35          return valid_input;
36      }
```

Figure 4: Code for function validate_input(). Allows the programme to check for any illegal inputs. The conversion from char to int is not situated within the function but in the main programme, as the function returns a Boolean rather than integers.

The input validation function checks the user input against several criteria:

- Does the length of the string exceed 2?
- Is the selected square within the chessboard?
- Did the player enter a special command? (e.g. help to display instructions, or moves to display all the moves made in the game)

To see the detailed activities and outputs of the programme, a flow chart was made and shown in figure 4.

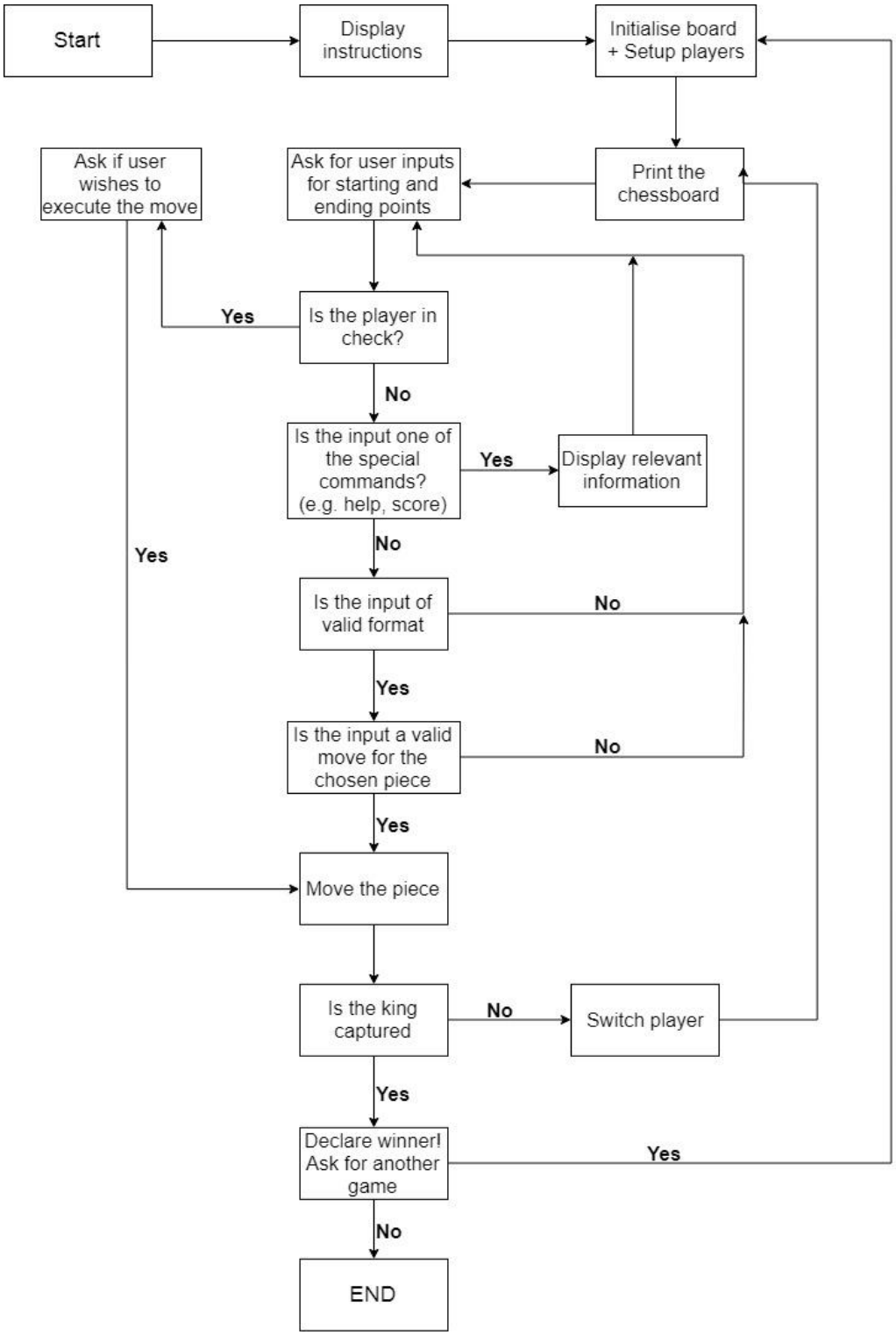


Figure 5: A flow chart showing the algorithm of the programme and how each decision impacts the paths it takes, made using draw.io.

2.4 Debugging and memory leak checks

Debugging were done using the built-in local window debugger in visual studio. Breakpoints were placed on the appropriate lines, by stepping in and out of various operations, all local variables can be displayed and compared to the expected values.

As the memory consumption of the programme depends on the length of the game, memory needs to be dynamically allocated, using the operators *new* and *delete*. Memory leak occurs when the programme does not free up the memory used to allocate variables, or when an object is defined in such a way that none of the functions can gain access to it. It is a leading cause in software aging[2]. Memory usage tool is integrated into the debugger in visual studio, which is shown in figure 5.

The first snapshot was taken just after compiling, the second was taken after the chessboard and all the chess pieces were created, the third was taken midway through a game. An increase in both number of objects and heap size can be seen between the first and second shot due to the addition of variables used in the chess pieces, and a decrease in memory can be seen between the second and third shot as the captured pieces are deleted from the board.

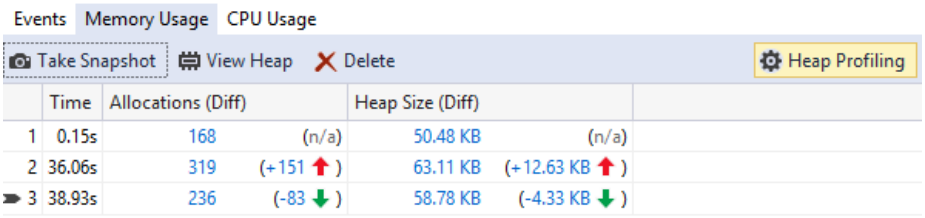
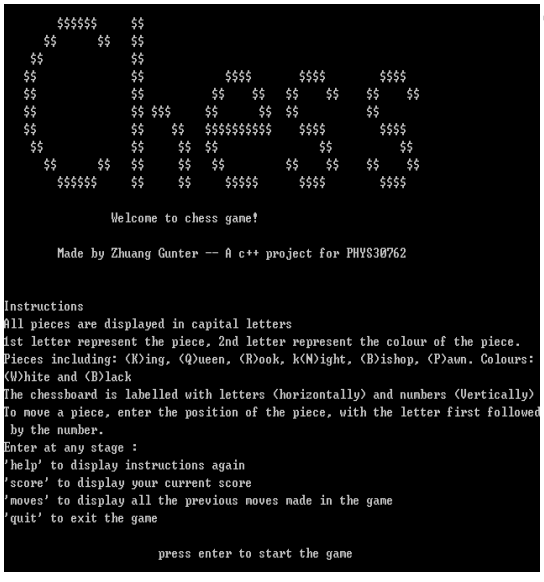


Figure 6: Memory usage tool in visual studio showing snapshots of the programme at different stages during the same run. The allocation difference column represents the number of objects and the heap size shows the number of bytes at this instance.

3. Results & Discussion

Figure 7(a) shows the initial greeting message followed by instructions for the programme. *Cout* was used to output all the texts to the command console. Figure 7(b) shows the chessboard at the start of the game.

7(a)



7(b)

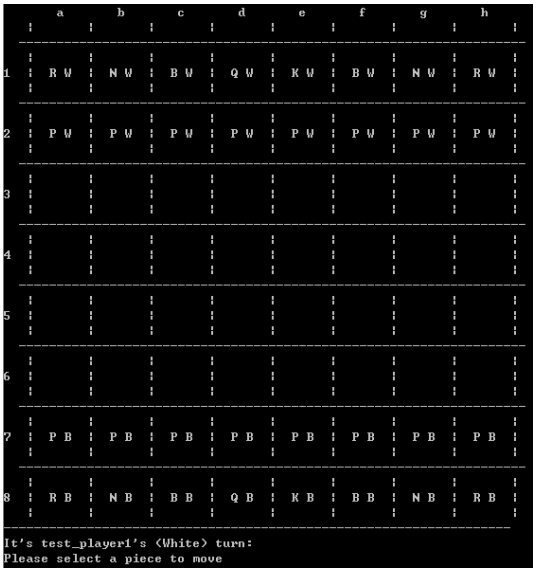
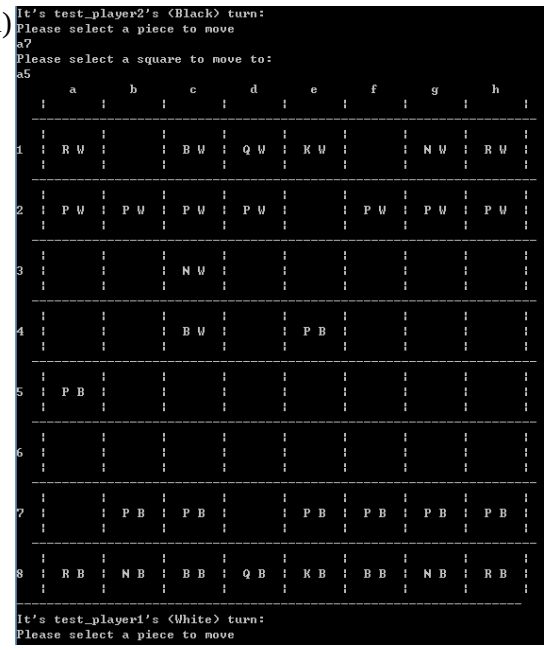


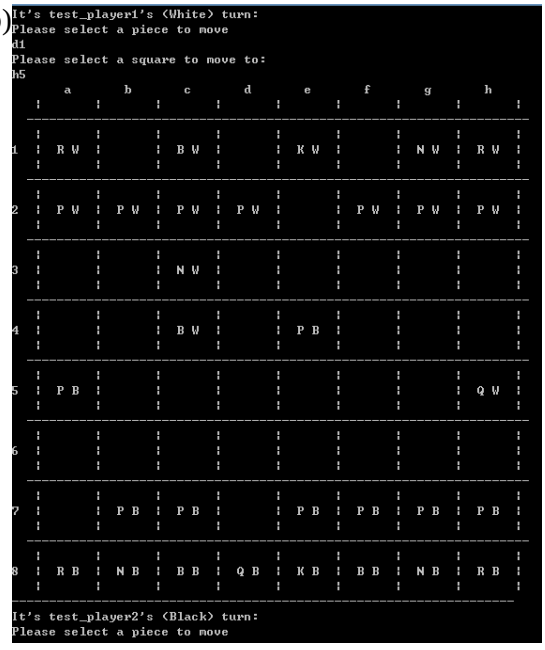
Figure 7: Screenshots of the command console. Figure 7(a) is the introduction interface which displays the chess logo and instructions on how to use the programme. Symbols for all chess pieces and all special commands are listed. Figure 7(b) shows the initial setup of the board. Black pieces are displayed at the bottom and white pieces are displayed at the top of the board unlike the conventional setup.

Figures 8(a) and 8(b) show the board after several moves and the corresponding inputs for those moves. Figure 8(c) shows the board after the black queen has advanced into square D2, placing the white player in check. *Moves* will display all previous valid moves made by both players, *Score* will display the sum of the values of the captured pieces and *quit* will exist the programme immediately. All functions are demonstrated in figure 8(d).

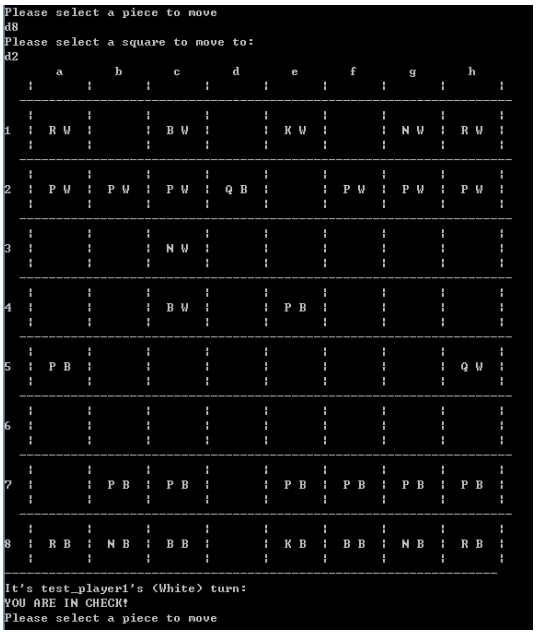
8(a)



8(b)



8(c)



8(d)

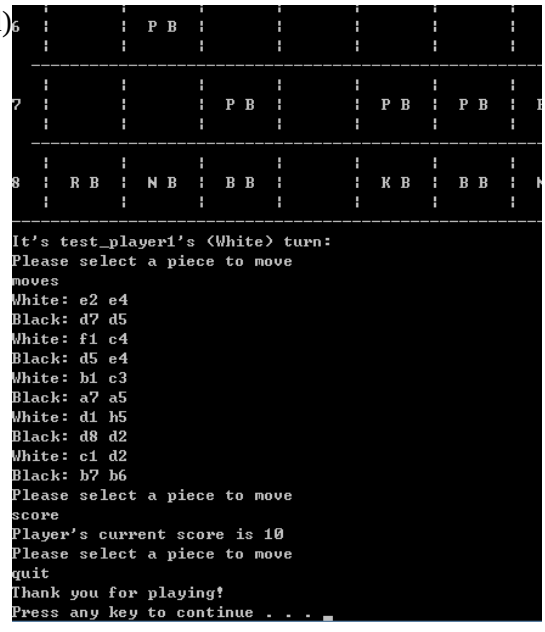


Figure 8: Screenshots during gameplay demonstrating all the operations.

The results for the programme are shown in the figures above. The programme takes in two digit inputs, check to see if the selected square is empty, if the piece belongs to the current player, if the selected piece is capable of moving to the destination, and if the destination square is umpty or occupied by a piece of the same colour. The programme then deletes the piece pointers in both squares and create a new pointer to the piece moved in the destination square. The check function works perfectly and it warns the player at the beginning of their turn if they are currently in check. Values are assigned to each piece and the player's score can be viewed at any given time. All valid moves made are stored in a vector of strings which can also be displayed.

There are several additional components which can be added to the programme. The most significant addition would an AI player. Starting by creating a function which returns a random move out of all the possible moves, and built upon the function by letting it choose the move which would capture a piece of the highest value, and further improving it by creating a tree diagram of all the possible moves for the next few turns and choosing the best move. Addition features the programme would benefit from is a takeback function, where the player can undo a bad move. The visualisation of the game interface can be improved by using the chess symbols defined in the Unicode character set, the chessboard can also be modified by using external libararies.

4. Conclusion

In conclusion, this programme is a simple replica of the board game chess, with all the basic movements implemented using OOP principles. The programme allows the user to play 2 player turned based chess, allows for movement, opening double jump for pawns, attacking vertically, horizontally and diagonally, obstruction check, and out of bound check. The programme utilises man C++ features, dynamic allocation of memory, iterators, static members, class hierarchy, virtual functions and separate header files. The main improvements for the programme would be an AI controlled opponent and more sophisticated graphics for the chess board using external library.

References

[1] Mitchell, J.C. (2003). *Concepts in Programming Languages*. Cambridge University Press.

[2] Shereshevsky, M., Crowell, J., Cukic, Bojan., Gandikota, V. and Liu Y. (2003). 'Software Aging and Multifractality of Memory Rescores'. *Proceedings of 2003 International Conference on Dependable Systems and Networks*. 1, pp.721-730.