Designing and programming an object-orientated chess program in C++

Rhydian Windsor

Student No: 9437658

School of Physics and Astronomy

The University of Manchester

PHYS30762&63762 Project Report

May 2017

Abstract

An object-orientated chess board was designed and written in C++. The board asks for user input for moves for both white and black sides and then checks if the move is legal. The board also inspects the current state of the board after each move to find check or checkmate conditions, and adjusts which moves are legal accordingly. If a legal move is entered, the new state of the board is printed out in a console window. The program also gives the option of the user to have their game recorded as a .PGN (Portable Game Notation) plain text file.

1 Introduction

Chess is one of the most ancient and widely-played games known to mankind. The rules of modern chess are believed to have been formulated in southern Europe around 1200 AD, although variants can be traced back hundreds of years earlier[1].The advent of computers, however, has changed the face of the game. Since IBM's *Deep Blue* beat then-World Champion Garry Kasparov in 1996, chess computers have become increasingly powerful and now are hundreds of Elo rating points higher-rated than the best human chess-players[2]. Furthermore, chess computers allow deeper analysis of the game than before and are attributed as a major reason for the increasing Elo of top players.

In this project, an object-orientated chess board was designed in C++. This calculates which moves are legal for each piece on the board and allows a user to conduct a game, playing as both sides. As a game, chess lends itself to polymorphic, object-orientated design due to its variety of piece types, each with unique moving and taking patterns as well as asymmetries between black and white pieces. One challenge of writing this program is that it must constantly observe the state of the board in order to determine whether the king of the colour about to make a move is in check. This would restrict which moves are legal. The board must also determine when checkmate occurs - that is, no legal move exists which will move the king out of check.

The program also has functionality which allows the chess game being played to be recorded as a Portable Game Notation (.PGN extension) file. This plain text file is the standard way of electronically storing chess games.

2 Code Design and Implementation

2.1 Class Hierarchy

This program consists of an abstract `Piece` base class. From this, classes for each individual pieces are derived which in turn each have two classes derived from them; one for white and one for black instances of the piece. This class hierarchy illustrated in Figure 1. `Piece` contains pure virtual methods such as `virtual bool Piece::move` and `virtual bool Piece::take`, responsible for each piece moving and taking which vary from piece to piece. It also contains non-virtual functions useful to all derived classes such as `void change_board_position`, which allows a piece to be moved to any square on the board, regardless of whether the move is legal for that piece or not (this is used in castling).

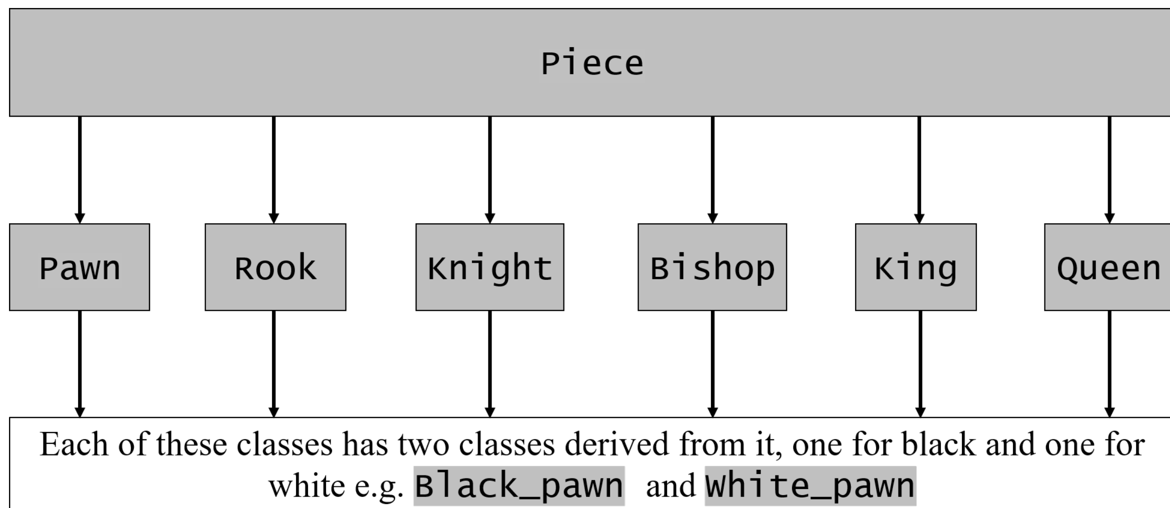

Figure 1: A figure showing the class hierarchy of the program. `Piece` is an abstract base class from which six classes for pieces are derived. Two further classes are derived from each of these, corresponding to black and white instances of the pieces.

2.2 The `vector<Piece*> pieces_vector` container

All the pieces on the board are stored in a `std::vector` container of pointers to the `Piece` class, called `pieces_vector`. The vector container is fairly memory intensive However, as the program only contains 32 pieces, this causes no noticeable problems with the program running slowly. The mechanics of promotion in chess meant that a dynamic memory container had to be used.

In this program pieces frequently must be accessed by their rank and file number. Here rank refers to the y-coordinate of the piece and file to the x-coordinate (in contradiction to standard chess nomenclature). Whilst this would have been easy in a `std::map` container, this is more difficult in vector where the element's indices do not tell us its position on the board. To solve this problem the `find_piece` function was defined, shown in Figure 2.

```
Piece* find_piece(int file_no, int rank_no, std::vector<Piece*> vector_of_pieces) {
    for (unsigned i = 0; i < vector_of_pieces.size(); i++) { //loop over vector of piece pointers
        //if file number and rank number match and piece hasn't been captured
        if (file_no == vector_of_pieces[i]->file && rank_no == vector_of_pieces[i]->rank){
            if (vector_of_pieces[i]->has_been_captured == false) {
                return vector_of_pieces[i];
            }
        }
    }
    return nullptr; //if no piece found, return null pointer
}
```

Figure 2: Function to return a pointer to a piece in `vector<Piece*> vector_of_pieces` if it has the same rank and file number as the function parameters.

2.3 `Piece` abstract base class

This class consists of several `protected` variables; `int rank`, `int file`, `char notation_letter`, `int number_of_moves_made` and `bool has_been_captured`. `int rank` and `int file` are integers which allows the `Piece` object to store it's own location. The piece class also contains `static char board[9][9]` member. This static data array is updated as pieces move so that it contains the standard notation letter of each piece in its board position. White pieces are indicated by capital letters and black pieces by lower case letters. For example if there is a black knight on the square h5, then `board[5][8]` will contain the character `n`.

The advantages of using a `board` static data array are many-fold. Firstly it makes the function to print out the current state of the board, `void print_board`, much simpler. It also means that complex operations which require knowledge of locations of other pieces on the board, such as determining whether a piece can be taken, does not necessitate looping through the `pieces_vector` multiple times, which could slow the program down. Finally having an array with all the pieces in makes conceptualizing the board whilst programming much easier. The reason that the board is 9 by 9 instead of 8 by 8 like a standard chess board is that this means `board[i][j]` contains the board on the ith file and j^{th} rank. Also the letters and numbers used to mark the board are stored in `board[0][y]` and `board[x][0]`, the first rank and column of the board respectively. This is useful in `print_board`.

2.4 File Structure

The program is split into seven files. This consists of "mainProgram.cpp", the source file for the main program and three headers files, each with their own implementation files."Piece_Functions_Header.h" contains the declaration for piece specific functions and "piece_functions.cpp" the respective function definitions. Similarly, "General_functions_header.h" contains the declarations the functions not associated with `Piece` class and "general_functions.cpp" contains the definitions of these. A third header file was included; "PGN_output_header.h". This declares functions responsible for producing the .PGN output file of the game. These functions are defined in "PGN_output_functions.cpp".

2.5 User Interface

This program is console-based, which presents several problems when displaying the board. One major problem was differentiating between white and black pieces. Using lower case letters for black pieces' notation letters and upper case letters for white pieces' location was suitable for the code to determine whether a piece is black or white. However, due to similarities between lower and upper case p displayed in the console window, it became difficult to distinguish each colors pawns when the board was displayed.

For this reason, the `<windows.h>` header was included. This contains several Windows-specific functions which allow the console window to be modified. The most important use of this in the project is allowing white and black pieces notation letters to be printed in different color text, allowing them to be distinguished. This function responsible for this is shown in Figure 3.

```
void print_board() {
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE); //acts as a 'pointer' to console window
    SetColorAndBackground(3, 0); //sets font color to blue and background to red
    for (int i = 8; i >= 0; i--) { //loops over all columns
        print_horizontal_line(); //prints horizontal line of dashes
        for (int j = 0; j < 9; j++) { //loops over all rows
            if (j == 1) { cout << "  "; }
            cout << " | ";
            if (is_black(square_occupancy(j, i)) && i >= 1) { SetColorAndBackground(10, 0); } //if black, font is green
            else if (is_white(square_occupancy(j, i)) && i >= 1) { SetColorAndBackground(15, 0); }//white font
            cout << square_occupancy(j, i); //print notation letter
            SetConsoleTextAttribute(hConsole, 3); //set console color back to blue
        }
        cout << " | " << endl;
    }
    cout << endl;
    SetColorAndBackground(0, 14); //set console color back to default yellow background
}
```

Figure 3: Function used to print out the current state of the board. `hConsole` is a handle, which allows reference to the console window.

The `print_board` function in Figure 3 makes use of the `SetColorAndBackground`, shown in Figure 4.

```
void SetColorAndBackground(int ForgC, int BackC)
{
    WORD wColor = ((BackC & 0x0F) << 4) + (ForgC & 0x0F);
    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), wColor);
}
```

Figure 4: This function changes the text and background color of the console window. `WORD` is a windows-specific unsigned data type, which acts as reference to the colors.

2.6 Portable Game Notation Output

Portable game notation files (.PGN extension) are the standard file format for recording chess games. The files are plain text and are designed to be easily comprehended by humans as well as being simple to parse and generate for computer programs. This program has functionality which allows chess games played to be recorded as a .PGN file. A .PGN file consists of two parts; “tag pairs” and “movetext”. Tag pairs (enclosed by square brackets) contain information about the chess match and movetext contains the series of moves made in the match, written in standard chess notation.

In this program “PGN_output_header.h” contains all the functions responsible for recording the game. The functions are contained in a separate namespace , `filewrite`, which allows them to be easily distinguished from the rest of the program. As .PGN files require a tag pair stating the date on which the game took place, this header makes use of the `time.h` header. The function that writes the tag pairs to the file is shown in figure 5.

```
namespace filewrite {
    std::ofstream create_PGN_file(std::string output_file, std::string player1, std::string player2) {
        time_t rawtime; //get raw time from time.h
        struct tm timeinfo;
        //put time informatino into struct tm timeinfo
        time(&rawtime);
        localtime_s(&timeinfo, &rawtime);
        std::ofstream file;
        file.open(output_file + ".PGN");
        //print out basic tag pairs for a PGN file
        file << "[Event \"Rhydian's Chess Game Match\"] \n";
        file << "[Site \"Manchester, LANCS GBR\"] \n";
        file << "[Date \""<<(timeinfo.tm_year + 1900)<<"."<< (timeinfo.tm_mon + 1)<<"."<< timeinfo.tm_mday<<"\"] \n";
        file << "[White \""<<player1<<"\"] \n";
        file << "[Black \""<<player2<<"\"] \n";
        file << " \n \n";
        return file;
    }
```

Figure 5: The function responsible for opening and writing writing tag pairs to the output .PGN file.

3 Results

This program is an object-orientated console-based chess board which allows chess games to be conducted, checking the validity of moves. Users input moves by entering the square of the piece they want to move and are prompted to enter the square they want to move it to. The program checks the validity of the move and ,if valid, the move is executed. If not valid, the move must be re-entered. The program must also determine whether the king of the color making the move is in check, which limits moves available to that color. Checkmate must also be determined as a game-ending condition. As an extension the board also allows games to be recorded as .PGN files under a file name of the user's choice.

The flowchart in Figure 6 shows the basic structure and functionality of the program.

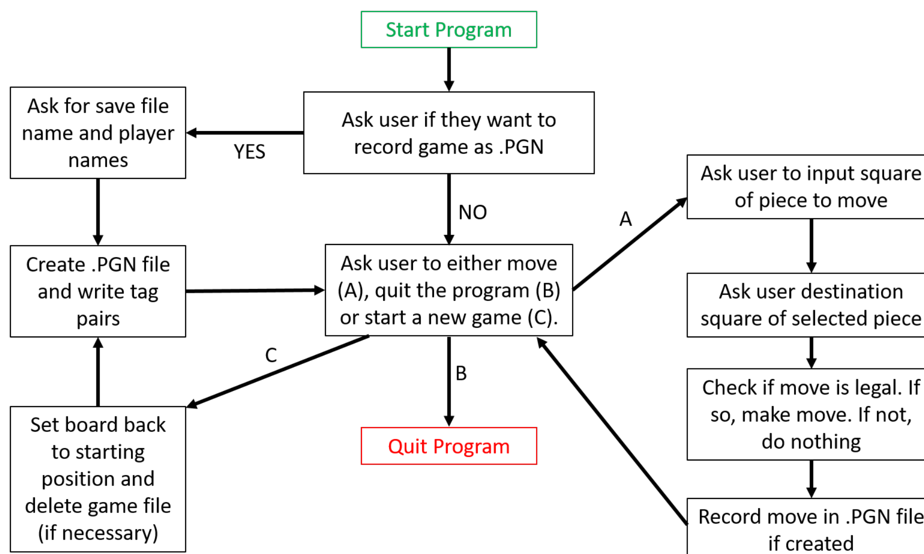

Figure 6: A flowchart showing the basic menu structure of the program. If the program detects that one side is in checkmate, it prints out a message informing the user, and then quits.

Figure 7 shows the program during operation. This shows the console window when the user is prompted to make a move. Incorrect inputs (i.e. not valid square locations) are detected and the user is asked to re-enter.

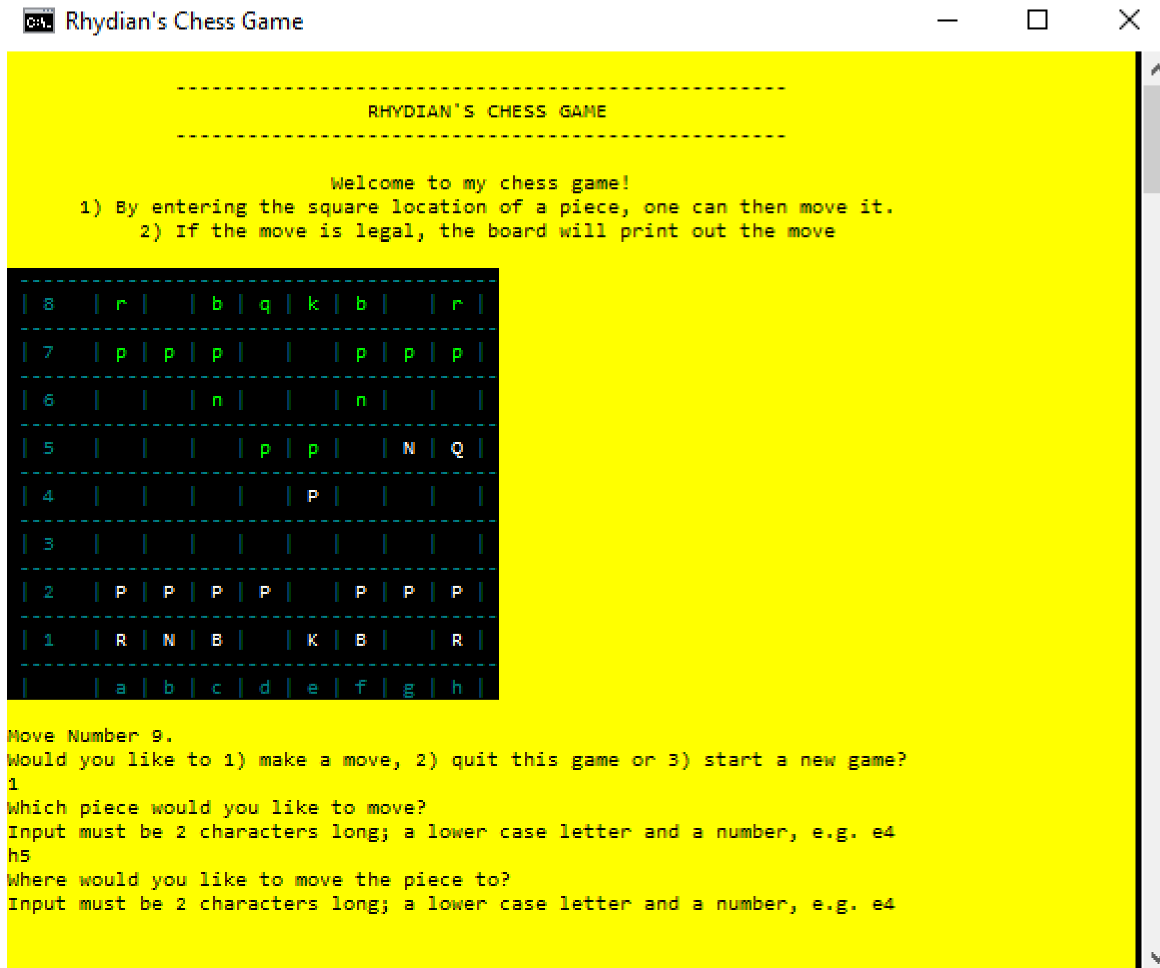

Figure 7: A screenshot of the console window during the operation. This is an example of a user (white) making a move.

An example of a .PGN file from recording a game using the board is stored as “ExampleGame.PGN” in the project file. A screenshot is also shown in Figure 8.

```
ExampleGame.PGN - Notepad
File  Edit  Format  View  Help
[Event "Rhydian's Chess Game Match"]
[Site "Manchester, LANCS GBR"]
[Date "2017.5.13"]
[White "Rhydian"]
[Black "McCoy"]


1. e4 e5 2. Nf3 Nc6 3. c3 Nf6 4. d4
```

Figure 8: A screenshot of a PGN file open in notepad. The square brackets at the top of the file are the tagpairs, the rest of the file is the movetext, which gives tells the moves played in the game.

4 Discussion and Conclusions

This program has all the functionality proposed in the project proposal, plus the extension of .PGN file generation. However, there are several things that could be improved.

Firstly, the `Piece` class contains many variables, the constant calculation of which takes up a significant amount of memory. Whilst this causes no noticeable performance problems for the program in its current state, when an AI is added this could slow the program. This could be improved by using the `std::map` template instead of std::vector container, which would make the `rank` and `file` variables redundant. One advantage of this container is that each index can only contain one element (value). This would allow the program to access pieces by an index which would relate to the position of the piece on the board and hence less memory-intensive looping through the `pieces_vector` would be necessary. Furthermore the program could be improved by using move semantics, which would lend itself to the idea of moving pieces in chess.

An obvious extension of this project is to include an AI which will allow the user to play against the computer. Whilst the developer of this program would like to introduce this at some stage, time constraints meant it was not possible to complete before project submission. An idea to make the computer make a random legal move after player input was considered but rejected. Most chess computer algorithms are based on evaluating the position of the board and assigning some score to the current board position based how good a position is deemed to be for the computers side. Examples of conditions which may be used to evaluate how good a position is include the number of pieces each side has, if there is any unguarded material, the position of the king (center of the board is deemed weaker).

However, one complaint often levelled at chess computers is how different they feel to playing another human. One way around this is to use machine learning to teach the computer how to play chess; by analysing games and observing outcomes of great players the computer could learn to become a great chess player. Obviously this is well beyond an undergraduate coding project, however it is an exciting idea and I believe is a superior model for designing chess computers than the current technique.

References

[1] Ostler, J 2009, *History of Chess*, viewed 13th May 2017. Available at https://www.chess.com/blog/jim_ostler/history-of-chess12

[2] Larsen, E. J. 2017, *A Brief History of Computer Chess* viewed 13th May 2017. Available at http://www.thebestschools.org/magazine/brief-history-of-computer-chess/

Typeset using LaTeX.

Word count: 2196