

Object-Orientated Programming: Implementation of a chess game

Christopher Kitching

10134621

School of Physics and Astronomy

The University of Manchester

Third Year

May 2020

Abstract

In this paper we outline the implementation of a fully working chess game in C++ using principles from object-orientated programming (OOP), namely encapsulation, inheritance and polymorphism. The goal of the project was to write a code which would allow two people to play a full game of chess, where all advanced rules are in play, and include some additional features to improve the quality of life. As an extension I developed a fairly sophisticated AI to allow a single person to play against the computer.

1. Introduction

Chess is a popular two-player strategy board game played out on an 8x8 grid. Each player is assigned a colour, white or black, and begins the game with 16 pieces, with the ultimate aim being to checkmate the opponents king piece, i.e. placing it in inescapable threat of capture [1]. The rules of chess are somewhat complicated; with six unique chess pieces, each with its own rule set on how it is allowed to move around the board and capture other pieces. There are also several advanced moves such as castling, en passant and pawn promotion which require unique conditions to be satisfied.

The game of chess lends itself well to an OOP implementation and we make heavy use of its three main ideas throughout the program. Encapsulation allows us to wrap data, and the functions that manipulate said data, in a single unit. This has several benefits such as reducing complexity, protecting our data and making the program more flexible to changes. The application of inheritance allows us to create derived classes, from an existing base class, which inherit all properties of that base class, as well as having additional features of its own. This is useful for implementing the pieces. Finally, we make use of both compile time polymorphism, to overload operators, and run time polymorphism, allowing us to create virtual functions in the base class that can be overridden in the derived classes.

In addition to these main ideas we also make use of other advanced features such as header files, smart pointers, exceptions and recursion to improve the program.

2. Code design and implementation

```
// Begin main program
int main() {

    int number_of_games{ 0 };
    int player_number;
    int difficulty;

    // loop for multiple games
    bool another_game;
    do {
        game chess; // instantiate a chess game

        // only print intro on the first game
        if (number_of_games == 0) {
            chess.introduction();
            std::cout << std::string(1, '\n');
        }

        player_number = number_of_players(); // get the number of players

        // if single player game, get the difficulty level
        if (player_number == 1) difficulty = select_difficulty();
        else difficulty = 1;

        another_game = chess.start(player_number, difficulty); // start the game
        number_of_games++;
    } while (another_game);

    std::cout << std::string(1, '\n');
    std::cout << "Thanks for playing!" << std::endl;

    return 0;
} // End of main program
```

Figure 1: Code showing the main game loop in the main.cpp file.

The program is divided through several .cpp files, each with its own header file, in order to make it more readable. The header files contain the definitions of the relevant classes, whose member functions are then implemented in the corresponding .cpp file. Each header file contains a header guard to prevent multiple definitions. The main file is fairly short, as is typical with game logic code. It simply contains a loop for multiple games, where a game object is instantiated and the start function is called on it, along with two simple functions which get the number of players and, for a single player game, the difficulty. The main program is shown in Figure 1.

2.1. The piece classes

In the header file of the base_piece class we first create an enum class to define the two player colours, as well as a structure of size_t variables to represent board positions. This setup is shown in Figure 2. Here, we also utilise run time polymorphism by declaring three functions

```
// enumeration for piece colour
enum class colour { white, black };

// structure for position of piece
struct position {
    size_t x;
    size_t y;
};
```

Figure 2: Code showing the implementation of an enum class for player colour and a structure for board position.

which overload operators. First the comparison operator, to check if two positions are equal, and then the insertion operator, to output the player colour and piece positions to the console. The declaration of these functions in the base_piece.h file are shown in Figure 3.

```
std::ostream& operator<<(std::ostream&, const colour&); // overload insertion operator for colour
std::ostream& operator<<(std::ostream&, const position&); // overload insertion operator for board positions
bool operator==(const position&, const position&); // function to overload comparison operator for positions
```

Figure 3: Code showing three function declarations in the base_piece.cpp file to overload the comparison and insertion operators.

To implement the pieces, we use a class hierarchy which is illustrated in Figure 4. The

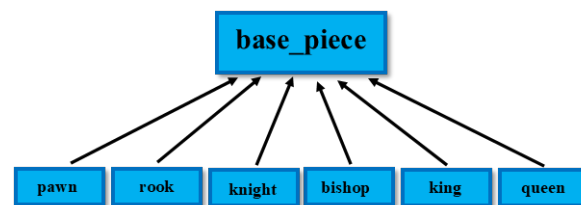


Figure 4: Class hierarchy used to implement the chess pieces. Created using Microsoft Publisher tools.

base_piece class is an abstract base class which makes use of the enum class and struct by having three protected variables, a colour, a position and a type, to uniquely specify a piece on the board. The class has a pure virtual function, shown in Figure 5, which will be overridden in the derived classes to check a move is valid.

There are six derived classes, one for each of the piece types. An example for the bishop class is shown in in Figure 6. Each of the derived classes overrides the valid_move function

```
virtual bool valid_move(const position&) = 0; // pure virtual function to test if move of specific piece is valid
```

Figure 5: Code showing the implementation of a pure virtual function in the abstract base_piece class.

which tests if the requested move matches the allowed move pattern for that piece. For example, the bishop can move an arbitrary number of spaces along any diagonal. Classes for pieces with special moves, i.e. the pawn, king and rook, have additional private Booleans, to test if they are allowed to perform these moves, along with access function to get and set this variable.

```
// derived class for bishop piece
class bishop : public base_piece
{
public:
    // constructors
    bishop(colour, position);
    virtual ~bishop() {};

    // public functions
    bool valid_move(const position&); // check move is valid for the bishop
};
```

Figure 6: Code showing the implementation of the bishop class, derived from the base_piece class.

2.2. The board class

To implement the board class, a class called square, as shown in Figure 7, was first created. This

```
class square
{
private:
    std::shared_ptr<base_piece> chess_piece; // each square contains a pointer to a chess piece
public:
    // constructors
    square() : chess_piece{ nullptr } {};
    ~square() {};

    //public functions
    std::shared_ptr<base_piece> get_piece();
    void set_piece(std::shared_ptr<base_piece> piece);
    void clear();
};
```

Figure 7: Code showing the implementation of the square class.

class contains one private variable which is a smart pointer, more specifically a shared pointer, to the base_piece object. The advantage of using smart pointers is we avoid memory leaks due to not deallocating memory. Smart pointers are always used in the program as oppose to raw pointers. The access functions of this class allow us to get or set pieces and are heavily used throughout the program.

The board class then, contains a private variable which is an 8x8 array of square objects. This class contains several public functions to control how pieces move around the board and to draw the board to the console.

First, in the constructor we dynamically allocate each piece to its initial position, shown schematically in Figure 8, using the language construct 'new'. An example of this code for the white bishops is shown in Figure 9. We can then make use of the move_piece function to re-assign piece positions, as shown in Figure 10.

Before calling this move function however, we go through several more complicated functions, implemented in this class, to test a moves legality. Firstly, we test separately for a special move

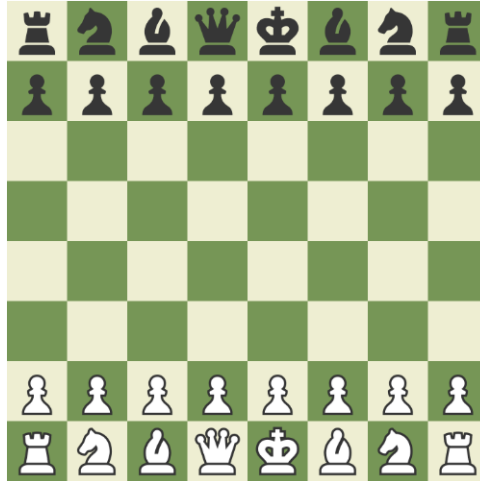


Figure 8: Image showing the initial setup of a chessboard [2].

```
// white bishops
position white_bishop_pos_1{ 2,0 };
game_board[white_bishop_pos_1.x][white_bishop_pos_1.y].set_piece(std::shared_ptr<base_piece>(new bishop(colour::white, white_bishop_pos_1)));
position white_bishop_pos_2{ 5,0 };
game_board[white_bishop_pos_2.x][white_bishop_pos_2.y].set_piece(std::shared_ptr<base_piece>(new bishop(colour::white, white_bishop_pos_2)));
```

Figure 9: Code showing the dynamic allocation of the white bishops to the board.

such as an en passant or castle as they have unique conditions and we need to record specific information about these moves so they can be undone if required. If not, we check the conditions of a regular move. If the piece is not a knight, it cannot jump over other pieces, so we test for pieces on the move path via the `obstructing_piece` function which loops over the intermediate positions and looks for a piece. Additionally, there is a function to test if the player is in check. You can never move into check and, if already in check, you must move out of it. The check condition can be tested by looping over all the opponents piece and testing if any of them can capture the king.

2.3. The game class

The game class contains all information about the current game and controls the flow of the game. This class is majority self-contained and so most of the variables and functions are private. It includes private variables to store the current players turn and several Booleans to keep track of special moves or win conditions. There are also several private functions which handle all game actions, validate moves and inputs, and cast between variables. We always pass arguments to member functions by reference to improve the speed of the program and these member functions are, if possible, made constant in order to avoid unwanted changes to objects.

The game class makes use of several C++ containers in order to store information about the game. A simple structure called 'round' contains the white and black move for each round. Each move is a string containing the initial and final position and is formatted as, for example, 'A2-C3'. A double-ended queue stores each of these rounds and is useful as it allows elements to be inserted or deleted from the front or back of the queue. The captured pieces are stored in vectors. Figure 11 shows the implementation of these containers in the game class.

```

// move the piece from initial to final position
void board::move_piece(const position& initial_pos, const position& final_pos)
{
    std::shared_ptr<base_piece> piece_to_move( game_board[initial_pos.x][initial_pos.y].get_piece() ); // get the piece from the initial position
    piece_to_move->set_position(final_pos); // change position of the piece
    game_board[final_pos.x][final_pos.y].set_piece(piece_to_move); // update final position on board to contain the piece
    game_board[initial_pos.x][initial_pos.y].set_piece(nullptr); // delete piece from its initial position on board
}

```

Figure 10: Code showing the implementation of the move_piece function in the board.cpp file.

```

// save the moves of each round
struct round
{
    std::string white_move;
    std::string black_move;
};

// double ended queue to store the rounds
std::deque<round> rounds;

// save the captured pieces in vectors
std::vector<std::shared_ptr<base_piece>> white_captured;
std::vector<std::shared_ptr<base_piece>> black_captured;

```

Figure 11: Code showing the implementation of containers to store game information in the game class.

The save and load game member functions are a particularly useful application of this data. Not only are these functions a gameplay improvement, they are also a powerful debugging tool. Testing advanced features such as castling required tedious setting up of the board, these functions bypass that process. The save function writes all information about the game to a .dat file, chosen by the user, which can then be read in at any time through the load function. The load game function is one instance where we make use of a try/catch block. We first read in the white player move, and then attempt to read in the black move. If an out of range exception is thrown, we catch it and set a Boolean to flag that there is no black move. We then use this fact to read in the moves correctly. The implementation of this code is shown in Figure 12. Note, a

```

// define the separators
std::size_t move_seperator{ line.find(" ") };
std::size_t player_seperator{ line.find("|") };

// read in white move
loaded_move[0] = line.substr(0, move_seperator);
loaded_move[1] = line.substr(move_seperator + 1, player_seperator - 3);

// try to read in black move
bool no_black_move{ false };
try {
    loaded_move[2] = line.substr(player_seperator + 1, move_seperator);
    loaded_move[3] = line.substr(move_seperator + 7);
}
// if no black move
catch (const std::out_of_range&) {
    no_black_move = true;
}

```

Figure 12: Code showing the implementation of try/catch to correctly read in moves in the load function, found in the game.cpp file.

game can only be loaded from the same game mode that it was saved in, i.e. you cannot load a two-player game in the single player mode. The program will catch this error, along with all other user input errors.

We also include an undo function, which can be repeatedly called, and is usable in both one and two player games. It reads in the last move from our rounds container and then resets the positions of all pieces involved. The difficulty comes in having to reset complex moves such as en passants. To do this, we utilise several vector Booleans which take snapshots of the game

state before and after moves are made. The state can then be fully restored after undoing a move.

Finally, the game class also implements a function to display all the players available moves. To do this, we run through all positions on the board to find a piece belonging to the player. Then we run through all positions again and test if a move to that position is legal, if it is, we store it in a vector. To print this information, as well as other relevant information such as captured pieces, we make use of vector iterators, an example is shown in Figure 13. We also exploit this function

```
// white captures
std::cout << "WHITE captured: ";
for (auto captured_vector_it = white_captured.begin(); captured_vector_it < white_captured.end(); captured_vector_it++) {
    std::cout << (*captured_vector_it)->get_type() << " ";
}
```

Figure 13: Code showing an example of a vector iterator to print relevant game information, namely the white captured pieces.

to develop our AI which is discussed in Section 4.

3. Results

When the game begins the player is greeted with an introduction screen, as shown in Figure 14, which gives a brief outline of how the game will work.

```
Welcome to OOP C++ Project - Chess!

Instructions:
Pieces are designated by W or B for colour, followed by a letter to denote the piece type:
    pawn = 'P', rook = 'R', knight = 'N', bishop = 'B', queen = 'Q' and king = 'K'
White always goes first
All advanced chess rules are in play, i.e double pawn advance, en passant, castling, pawn promotion.
Upon pressing enter you will be asked if you wish to play a one or two player game.
If a single player game is chosen you will be asked to select the difficulty: 'E' = easy, 'M' = medium, 'H' = hard
At the start of each turn you will be prompted for an 'action', type:
    'N' = start a new game,
    'L' = load a previously saved game,
    'S' = save the current game,
    'M' = make the next move (enter two co-ordinates; a letter (column) followed by a number (row) e.g "A4"),
    'U' = undo the last move,
    'Q' = quit the game,
    'A' = moves available,
    'H' = display these instructions again
The undo ability can be used in both game modes as many times as you like
You can only load a game in the same mode it was saved in, i.e cannot load a single player game in two player mode.
```

Figure 14: Game introduction screen outputted to the console on start up.

On pressing enter, the player is prompted for whether they wish to play a one or two player game. If they choose single player, they are also asked for a difficulty. Upon selection, the game board is printed and the white player, white always goes first, is prompted for a non-case sensitive action. This is shown in Figure 15. The actions are explained in the introduction. 'N' starts a fresh game, while 'S' and 'L' will prompt you for a file name to save or load a game, respectively. 'H' will re-print the instructions for the game and 'Q' quits the game entirely, breaking the main game loop. Entering 'M' prompts the player to enter initial and final positions, these inputs are non-case sensitive. An example of this is shown in Figure 16. If the input is invalid in any way, an error message is outputted, and the player is continuously asked for a move until a valid one is entered. If playing a single player game, the computer will then make its move before prompting the player for another action. In a two-player game the turn will be alternated and black can enter their action.

```

Press enter to start the game

Number of players (1/2):1
Player vs Computer

Difficulty level [(E)asy, (M)edium, (H)ard]:e
Easy mode selected!

---      ****      ****      ****      ****
-8- BR *BN* BB *BQ* BK *BB* BN *BR*
---      ****      ****      ****      ****
---      ****      ****      ****      ****
-7- *BP* BP *BP* BP *BP* BP *BP* BP
---      ****      ****      ****      ****
---      ****      ****      ****      ****
-6-      ****      ****      ****      ****
---      ****      ****      ****      ****
---      ****      ****      ****      ****
-5-      ****      ****      ****      ****
---      ****      ****      ****      ****
---      ****      ****      ****      ****
-4-      ****      ****      ****      ****
---      ****      ****      ****      ****
---      ****      ****      ****      ****
-3-      ****      ****      ****      ****
---      ****      ****      ****      ****
---      ****      ****      ****      ****
-2- WP *WP* WP *WP* WP *WP* WP *WP*
---      ****      ****      ****      ****
---      ****      ****      ****      ****
-1- *WR* WN *WB* WQ *WK* WB *WN* WR
---      ****      ****      ****      ****
-----
-a--b--c--d--e--f--g--h--
-----

Whites' turn

(N)ew Game, (L)oad Game, (S)ave Game, (M)ove, (U)ndo, (Q)uit, (A)vailable moves, (H)elp
Enter an action:

```

Figure 15: The initial game board outputted to the console.

```

Enter co-ords of piece to move (letter, number):a2
Enter co-ords of position to move to (letter, number):A4

```

Figure 16: An example move performed by the white player from A2 to A4. The input is non-case sensitive.

After every turn, a summary of the current game so far is outputted to the console beneath the game board. This includes the most recent five moves made by each player and all the captured pieces. An example of this is shown in Figure 17.

The last two actions are by far the most complex from a coding standpoint but offer a significant quality of life improvement. Entering 'U' will undo the last move entirely and remove all its details from the summary. Entering 'A' will list all available moves that the player can make. An example is shown in Figure 18.


```

--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-7- *BP* BP *BP*      *BP*      *BP* BR
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-6-      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-5-      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-4- WP      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-3-      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-2-      *WP*      *WP* WP *WP* WP      *
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-1- *WR* WN *WB* WQ *WK* WB      *WR
--      *BN*      *BQ* BK *BB* BN      *
-----a---b---c---d---e---f---g---h---
Last moves (white | black):
7....g5-h7 | h8-h7
6....f3-g5 | h7-h6
5....h3-g4 | d4-c3
4....h2-h3 | d5-d4
3....c2-c3 | c8-g4
-----
WHITE captured: B
BLACK captured: P N
-----

```

Figure 17: An example of the game summary, updated at the end of each turn.

```

There are 31 legal moves available for white:
[a1-a2] [a1-a3] [a1-b1] [a4-a5] [b2-b3]
[b2-b4] [c3-a2] [c3-b1] [c3-b5] [c3-d1]
[c3-d5] [c3-e2] [c3-e4] [d2-d3] [d2-d4]
[e1-d1] [e1-e2] [e3-e4] [f1-a6] [f1-b5]
[f1-c4] [f1-d3] [f1-e2] [f2-f3] [f2-f4]
[g1-e2] [g1-f3] [g1-h3] [g2-g3] [h2-h3]
[h2-h4]
-----
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-7- *BP* BP *BP* BP *BP* BP *BP*
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-6-      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-5-      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-4- WP      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-3-      *WN*      *WP*      *WN*      *WP*
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-2-      *WP* WP *WP*      *WP* WP *WP*
--      *BN*      *BQ* BK *BB* BN      *
--      *BN*      *BQ* BK *BB* BN      *
-1- *WR*      *WB*      *WK* WB *WN* WR
--      *BN*      *BQ* BK *BB* BN      *
-----a---b---c---d---e---f---g---h---
-----

```

Figure 18: An example of the available_moves action. All available moves that can be made by the player are output to the console.

4. Developing an AI

To develop an AI, we build off of the `available_moves` function. At the simplest level we could find all available moves for the computer, then choose a move at random, seeded with the time. However, the move would be far from the best one. To create a competent AI the computer needs more knowledge about chess.

Firstly, we assign a weight to each piece based on its value, as shown in Figure 19. The game













	10		-10
	30		-30
	30		-30
	50		-50
	90		-90
	900		-900

Figure 19: Weights assigned to each piece based on its value. Created using Microsoft Publisher tools. The values chosen are based off of data from [3].

class implements a function called `find_best_move` which uses the `available_moves` function to list all the possible moves for the computer. It then loops through moves and attempts to maximise, if white, or minimise, if black, the sum of all weights. The move that returns the best value is chosen.

This can be improved further via a minmax algorithm. We use a search tree to explore all possible moves to a depth determined by the difficulty level chosen at the start. To do this, we recursively call the `find_best_move` function until we reach the ending leaves of the tree, where the board state is ultimately evaluated. In other words, we simulate a certain amount of moves into the future, to find the move that will lead to best end result.

This method is computationally expensive so to counter this we make use of alpha-beta pruning [4]. Here we evaluate the board state after each move and if, at any point, the state is worse than the previous state, we disregard that branch of the tree. Even with this change the process is still expensive, for this reason we only recursively call the function three times at max, corresponding to the 'hard' difficulty.

These improvements make it so the AI will prioritize capturing an opponent's piece, as well as sacrificing its own weaker pieces if it means saving a more valuable one. This is somewhat naive as if there are no pieces to capture the move will be arbitrary. To change this, we implement a position evaluate function, which considers the value of having a piece in a certain position.

For example, pieces near the centre are often more valuable as they have more move options. We create several 2D arrays to hold these weights, known as piece-square tables, an example for the white bishops is shown in Figure 20.

```
// white bishop position evaluation array
double white_bishop_pos_eval[8][8]{
    {-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0},
    {-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0},
    {-1.0, 0.0, 0.5, 1.0, 1.0, 0.5, 0.0, -1.0},
    {-1.0, 0.5, 0.5, 1.0, 1.0, 0.5, 0.5, -1.0},
    {-1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, -1.0},
    {-1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0},
    {-1.0, 0.5, 0.0, 0.0, 0.0, 0.0, 0.5, -1.0},
    {-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0}
};
```

Figure 20: Code showing the implementation of a 2D array to hold the weights of positions for the white bishops. This table is a re-scaled version of the table found at [3].

5. Discussion and conclusions

In this project we successfully demonstrated the implementation of a fully working chess game in C++ using Object-Orientated features, namely encapsulation, inheritance and polymorphism. The game had all advanced moves in play and included several additional quality of live features, such as save/load game, undoing moves and listing the possible available moves. We also developed a competent AI for use in a single player game.

A natural extension would be to implement a GUI to improve the look of the game. The AI could be improved in several ways. For example, letting the piece-square tables evolve with the game state.

6. Length and date

The number of words in this document is 2492

This document was submitted on 06/05/20 at 13:00

References

- [1] H. J. R. Murray, *A history of chess*. Clarendon Press, 1913.
- [2] Y. Zhong, “Object-oriented implementation of chess game in c++,” in *Journal of Physics: Conference Series*, vol. 1195, p. 012013, IOP Publishing, 2019.
- [3] G. Isenberg, “Evaluation - chessprogramming wiki.” <https://www.chessprogramming.org/Evaluation>, May 2018. [Online; accessed 05/05/20].
- [4] D. E. Knuth and R. W. Moore, “An analysis of alpha-beta pruning,” *Artificial intelligence*, vol. 6, no. 4, pp. 293–326, 1975.