Object-Oriented Programming (OOP) in C++: Chess Project

Harry Halfhide

9916374

School of Physics and Astronomy

The University of Manchester

May 2020

Abstract

This report describes the use of various principles of OOP to create a 2-player game of chess. The board and pieces are modelled using classes, inheritance, and polymorphism. The rules of chess are implemented using class member functions and data. The programme, classes, and functions are organised into separate header and source files. Advanced features of C++ including shared pointers, templates, and static data are used.

1 Introduction

Object-oriented programming (OOP) is a style of programming based on the design and implementation of objects, which are entities used to represent or model a component of a system [1]. Objects are instances of classes, which are groups of objects with common characteristics [2]. The aim of this project is to use the principles of OOP to create a game of chess. Chess is a good candidate for the use of OOP as the game is built out of a board and multiple types of pieces. The board and pieces can be modelled using principles of OOP such as classes, objects, inheritance, and polymorphism.

2 Code Design and Implementation

2.1 Classes and Inheritance

The description of pieces in the code utilised inheritance, where a derived class is defined to inherit attributes and behaviour from 1 or more base classes [3]. Base classes often use virtual functions, which ensure that the correct function is called for an object. Base classes which contain only virtual functions are abstract base classes. An abstract base class, "`piece`", is defined in "Piece.h", shown in figure (1). "`piece`" contains "`public`" functions. "`public`" informs the compiler that everything declared below can be accessed from outside the class. Every class is surrounded by header guard to prevent multiple definition.

```
#ifndef PIECES
#define PIECES
class piece
{
public:
	virtual ~piece() {};
	virtual int get_player() = 0;
	virtual std::string check_type() = 0;
	virtual std::vector<int> allowed_moves(board game_board, int pos) = 0;
};
#endif
```

Figure 1 – The abstract base class "`piece`" in "Piece.h" surrounded by header guard.

For each piece type, a new class is derived from "`piece`" in its own header file. An example is shown in figure (2). Derived classes from "`piece`" such as "`pawn`" in "Pawn.h" inherit publicly from piece, meaning that all things inherited from "`piece`" maintain their privacy level in the derived class. Derived piece classes contain "`private`" data, meaning that it can only be accessed from inside the class. The integer "`player`" describes which player the object belongs to. In this example, static data integers "`p1_pawns`" and "`p2_pawns`" declared inside the class, describe the number of pawns belonging to each player. Static data can be accessed and modified by all objects belonging to a class.

```
class pawn : public piece
{
private:
	int player{};
	// Keeps track of the  total amount of pawns for each player
	static int p1_pawns;
	static int p2_pawns;
public:
	pawn(int p);
	~pawn();
	int get_player();
	std::string check_type();
	std::vector<int> allowed_moves(board game_board,int pos);
};
```

Figure 2 – The class "`pawn`" in "Pawn.h", which is derived from "`piece`".

Functions declared in derived piece classes in header files are defined outside of the class in a corresponding .cpp source file. Static data is initialised outside in the source file, as seen in figure (3). Each derived piece class contains a parameterised constructor with 1 parameter "`p`", which sets the object's "`player`" integer. When an instance of a derived piece class, such as "`pawn`" is created, 1 is added to "`p1_pawns`" or "`p2_pawns`" based on the value of "`player`". Similarly, 1 is removed from the corresponding static data integer when the destructor is called.

```
#include"Pawn.h"
int pawn::p1_pawns{};
int pawn::p2_pawns{};
pawn::pawn(int p)
{
    player = p;
    if (p == 1) {
        p1_pawns++;
        std::cout << "Player 1's pawn has been created. Player 1 pawns: " << p1_pawns << std::endl;
    }
    else {
        p2_pawns++;
        std::cout << "Player 2's pawn has been created. Player 2 pawns: " << p2_pawns << std::endl;
    }
}
```

Figure 3 – The initialisation of static data members and the description of a parameterised constructor in "Pawn.cpp".

The other type of object used is "`board`" defined in "Board.h", shown in figure (4). "`board`" contains a vector of pointers to "`piece`" objects called "`squares`" which represent the 64 squares on the board.

```
class board
{
public:
  // Polymorphic vector of pieces
  std::vector<std::shared_ptr<piece>> squares;
  board();
  ~board() {}
  void move_piece(int o, int n);
  void print_board(std::vector<int> allowed, std::vector<std::string> p1_moves, std::vector<std::string> p2_moves, int player);
  bool check_check(int player);
};
```

Figure 4 – The "`board`" class in "Board.h".

Base class pointers can point to objects from derived classes. Therefore, the pointers within the "`squares`" vector can point to objects from multiple classes derived from "`piece`", making "`squares`" a polymorphic vector.

	A	B	C	D	E	F	G	H
7	56	57	58	59	60	61	62	63
6	48	49	50	51	52	53	54	55
5	40	41	42	43	44	45	46	47
4	32	33	34	35	36	37	38	39
3	24	25	26	27	28	29	30	31
2	16	17	18	19	20	21	22	23
1	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7

The pointers within "`squares`" are a type of smart pointer called a shared pointer, which automatically destroy an object when the last remaining shared pointer owning the object is destroyed or assigned to another object [4]. There is therefore no need to manually delete an object to free memory. The "`board`" default constructor assigns 64 "`nullptrs`" to the "`squares`" vector, "`nullptrs`" represent empty squares. The relationship between vector index and board coordinates is shown in figure (5).

Figure 5 – The squares on the board that each pointer in "`squares`" is assigned to.

2.2 Main Game code

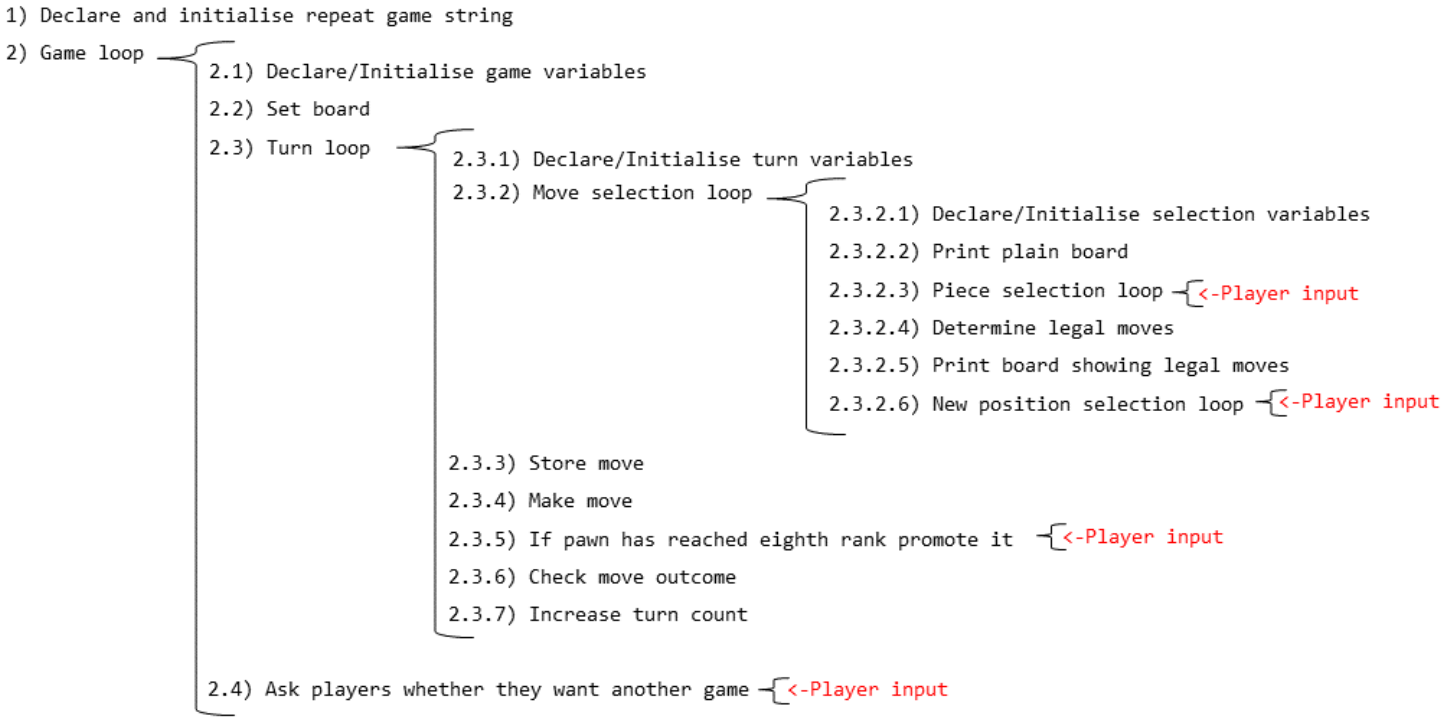

Figure 6 – The structure and individual steps of the main game code.

The structure of the game code, in "Game.cpp" is shown in figure (6), it is comprised of 3 main while loops. Firstly, the repeat game string is initialised to "Y", the game loop will repeat while repeat is "Y".

2) Game Loop

Variables which last an entire game are declared/initialised. The new "`board`" is declared. The pointers in "`squares`" are assigned to their correct pieces with the correct member data by the "`set_board_classic`" function.

Other game variables are: 2 string vectors to store players moves; "`p1/p2_moves`", a turn count integer; "`count`", and a boolean; "`legal_moves`", which is only true if the next player has legal moves. Next is the turn loop, which repeats until "`legal_moves`" is false. Players are asked if they would like another game, they can then change the repeat game variable to either "Y" or "N". Players are asked again until a valid input is given.

2.3) Turn Loop

The first step is to declare and initialise variables which last an entire turn. These include strings "`original_pos`" and "`new_pos`" which hold the coordinates of the starting and destination squares. "`new_pos`" is initialised to "X". Integers "`player`" and "`opponent`" are initialised to `1` or `2` and are determined by "`count`". Next, the move selection loop gets a legal move from a player, this is stored in "`p1_moves`" or "`p2_moves`". The move is made, if the move results in a pawn reaching eighth rank, the player is asked for a promotion choice until a valid one is provided. The promotion is made. Next, the outcome of the move is determined. This is dependent on 2 booleans: "`check`", which is only true if the opponent is in check, and "`legal_moves`" which is only true if the next player has legal moves available. Placing yourself into check is an illegal move. The outcome is decided as shown in table (1). The turn count is increased by `1`.

Table 1 – The outcome of a turn.

		check_check	
		true	false
legal_moves	true	Opponent has been checked `std::cout << "Player " << opponent << " is in check!";`	As normal
	false	Opponent has been checkmated `std::cout << "Checkmate! Player " << player << " wins!";`	Stalemate `std::cout << "Stalemate! Game over!";`

2.3.2) Move Selection Loop

Selection variables are declared, these are 2 integer vectors: "`allowed`" and "`allowed_checked`". "`allowed`" containing the indexes of legal squares for a piece to move to (does not consider whether the player is playing themselves into check). "`allowed_checked`" is a similar vector with self-checking moves removed. A representation of the board is outputted, this also contains a table displaying each player's last 24 moves. The piece selection loop asks the player to select a piece until a valid piece is chosen. "`allowed_checked`" is determined, the board is printed again with asterisks placed in allowed squares. Next is the new position selection loop, which asks the player to enter a destination square until an allowed one is entered. If "`X`" is entered, the programme will return to the top of the move selection loop, the player will be able to select another piece.

2.3 Piece Member Functions

All types of pieces have three member functions inherited from "`piece`". Access functions "`get_player`" and "`check_type`" return the player integer and the type string (e.g. "`pawn`"). "`allowed_moves`" fills a vector of integers with the indexes of "`squares`" pointers that are legal for that piece to move to (does not consider self-checking).

	Up	Down	Null
Right	+9	-7	+1
Left	+7	-9	-1
Null	+8	-8	0

Table 2 – The change in "`squares`" index for each direction.

Figure (5) shows that moves in any direction can be described by a change of index in "`squares`", these are shown in table (2). Rows in figure (5) are defined by dividing the index by 8 and rounding down to the next whole number. Columns are defined by the remainder after dividing the index by 8. This allows piece location and movement to be completely described by the associated "`squares`" indexes.

The rook, bishop, and queen all use functions declared in "Moves.h" and defined in "Moves.cpp" in their "`allowed_moves`" functions. Functions in "Moves.h" iterate over "`squares`" using the values in table (2). Depending on the next square scenario, a different action is taken, shown in table (3). The queen uses these functions in every direction, the bishop only uses diagonal functions, and the rook only uses horizontal and vertical functions.

Table 3 – The action taken by "Moves.h" functions for each scenario.

Next Square Scenario	Action
Free	Add square to allowed, continue
Opponent piece	Add square to allowed, stop
Player piece	Stop
Across board edge	Stop

"allowed_moves" in "pawn" allows for a change of "squares" index of +/-1 or +/-2 into free squares on their first turn for player 1/2. Player 1/2's pawn can change "squares" index by +/- 7 or +/-9 if the destination square points to an opponent's piece.

"allowed_moves" for "king" and "knight" create a preliminary list of allowed "squares" indexes as if there were no other pieces. This is only dependant on the current position, for example if a piece is on the lower board edge, then it cannot move downwards. This preliminary list is then iterated over, if the "squares" pointer is "nullptr", or points to an opponent's piece, it is added to "allowed".

2.4 Board Member Functions

The void function "move_piece" assigns the "squares[destination square]" pointer to the piece being moved and then assigns "squares[original square]" to "nullptr". Any taken pieces are automatically destroyed due to the use of shared pointers.

The boolean function "check_check" iterates over "squares" and finds the position of the player's king and opponent's pieces. Allowed moves are found for all the opponent's pieces. If the king's position is an allowed move for any of the opponent's pieces, then the function returns true, otherwise if returns false.

The void function "print_board" contains an array of char arrays called "display". Characters "_", " " and "|" are used to draw the board and a table which displays previous moves. The last 24 entries of "p1_moves" and "p1_moves" are iterated over and added to the table. The "squares" vector is iterated over, if the square contains a piece this is represented in "display". Pieces are represented by the first letter of their name and the player they belong to; knights are represented by "N". Pieces and letters/numbers marking the coordinates are filled in a different order depending on whether player 1 or 2 is specified. This is to reflect the perspective of the player in a real-life game. "display" is then printed using 2 for loops.

2.5 Game Functions

Functions that are used in "game.cpp" are declared in "game_functions.h" and are defined in "game_functions.cpp". This contains functions to: set the board, re-format inputs, check valid inputs, and to convert between "squares" index and coordinate string, among others. It also contains the template function "test_move_for_check", shown in figure (7). This takes a move where a piece is taken, performs it, and adds it to "allowed_checked" if the move does not self-check the player. It also reverses the move afterwards. This requires a template as the type of piece that needs to be replaced to reverse the move will not be the same every time.

```
template <typename T>
void test_move_for_check(board& game_board, int orig_pos, int new_pos, int player,
std::vector<int>& allowed_checked)
{
    int opponent;
    if (player == 1) opponent = 2;
    else opponent = 1;
    // Put stream into fail state so that test moves performed by the programme dont cause output
    std::cout.setstate(std::ios_base::failbit);
    // Perform move
    game_board.move_piece(orig_pos, new_pos);
    // If player does not play themselves into check, add to allowed checked
    if (game_board.check_check(player) == false) allowed_checked.push_back(new_pos);
    // Move piece back
    game_board.move_piece(new_pos, orig_pos);
    // Replace taken piece
    game_board.squares[new_pos] = std::make_shared<T>(opponent);
    // Clear failbit
    std::cout.clear();
}
```

Figure 7 – The template function "test_move_for_check".

This is used in the function "`test_all_for_check`", shown in figure (8), which performs the same function for a vector of moves, considering taking and non-taking moves.

```
void test_all_for_check(std::vector<int> allowed, board& game_board, int player, int pos, std::vector<int>&
allowed_checked)
{
    // Perform each allowed move, if the move does put the player in check then add to allowed_checked
    for (auto allowed_it = allowed.begin(); allowed_it < allowed.end(); allowed_it++) {
        // For moves in which no opponent pieces are taken
        if (game_board.squares[*allowed_it] == nullptr) {
            game_board.move_piece(pos, *allowed_it);
            if (game_board.check_check(player) == false) allowed_checked.push_back(*allowed_it);
            game_board.move_piece(*allowed_it, pos);
        }
        // For moves in which opponent pieces are taken
        else if (game_board.squares[*allowed_it]->check_type() == "pawn") {
            test_move_for_check<pawn>(game_board, pos, *allowed_it, player, allowed_checked);
        }
        else if (game_board.squares[*allowed_it]->check_type() == "rook") {
            test_move_for_check<rook>(game_board, pos, *allowed_it, player, allowed_checked);
```

Figure 8 – The use of template function "`test_move_for_check`", in "`test_all_for_check`".

3 Results

```
Player 2's pawn has been created. Player 2 pawns: 7
Player 2's pawn has been created. Player 2 pawns: 8

                                CHESS

     A    B    C    D    E    F    G    H         P1    P2
   ____ ____ ____ ____ ____ ____ ____ ____      ____  ____
  |    |    |    |    |    |    |    |    |    |    |    |
7 | R2 | N2 | B2 | Q2 | K2 | B2 | N2 | R2 |
  |____|____|____|____|____|____|____|____|
  |    |    |    |    |    |    |    |    |
6 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 |
  |____|____|____|____|____|____|____|____|
  |    |    |    |    |    |    |    |    |
5 |    |    |    |    |    |    |    |    |
  |____|____|____|____|____|____|____|____|
  |    |    |    |    |    |    |    |    |
4 |    |    |    |    |    |    |    |    |
  |____|____|____|____|____|____|____|____|
  |    |    |    |    |    |    |    |    |
3 |    |    |    |    |    |    |    |    |
  |____|____|____|____|____|____|____|____|
  |    |    |    |    |    |    |    |    |
2 |    |    |    |    |    |    |    |    |
  |____|____|____|____|____|____|____|____|
  |    |    |    |    |    |    |    |    |
1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 |
  |____|____|____|____|____|____|____|____|
  |    |    |    |    |    |    |    |    |
0 | R1 | N1 | B1 | Q1 | K1 | B1 | N1 | R1 |
  |____|____|____|____|____|____|____|____|

Player 1 please enter the position of the piece you wish to move:
```

Figure 9 – The creation of pieces (upper), the representation of the board, and request for player 1 to select a piece(lower).

When the programme is started, all steps up to 2.3.2.3 are completed without user input, static data for all pieces are increased by their constructors. The user is then asked to select a piece, this is shown in figure (9). The player can then enter the coordinate in upper or lower case with any number of spaces, the board is printed showing available moves, shown in figure (10).

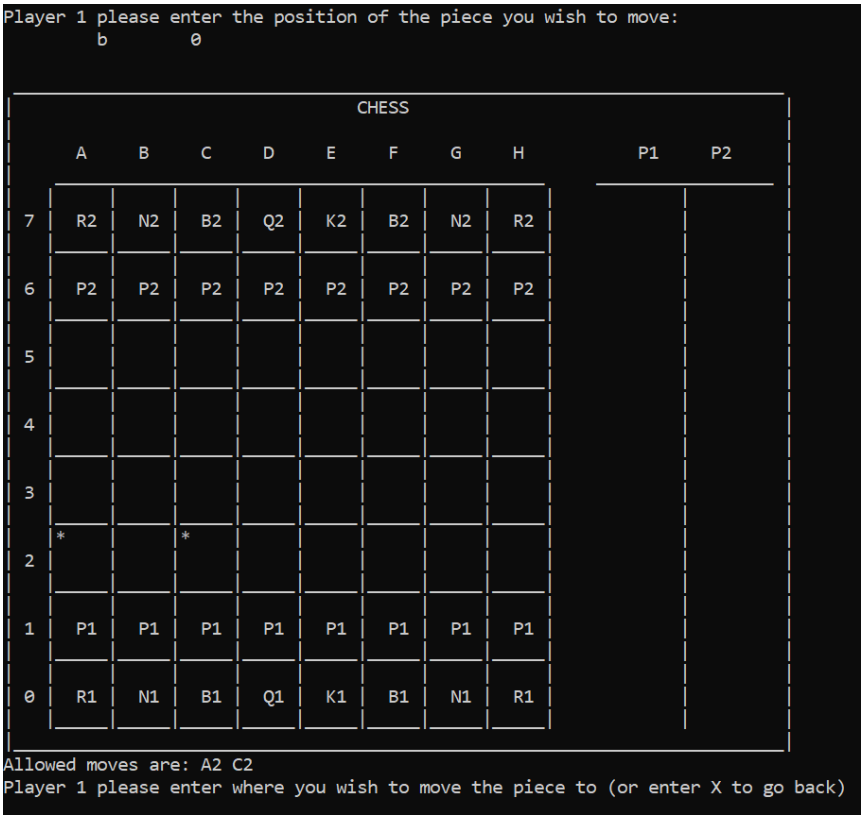

Figure 10 – Player 1's input (upper) and the board showing available moves

If the player enters "X" in upper or lower case with any number of spaces, the output will be the same as in figure (10). If the player enters an allowed move, then the move is made. The board then is shown from player 2's perspective with the move added to the move table, shown in figure (11). Player 2 is then asked for a move.

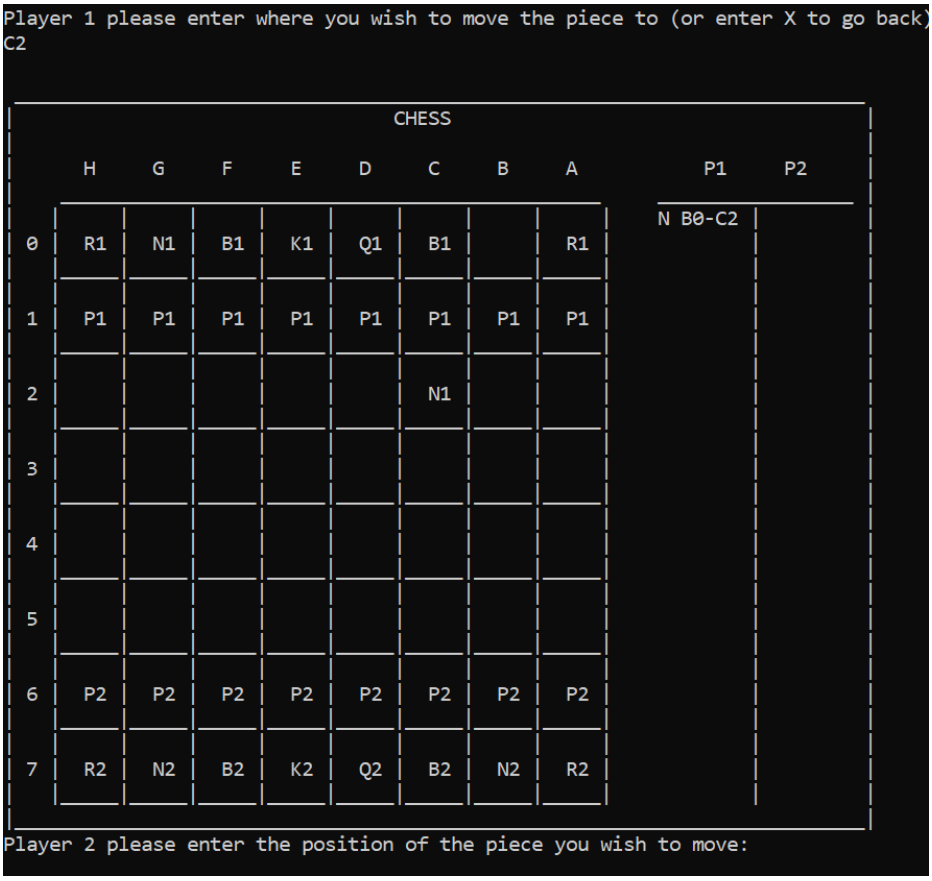

Figure 11 – The board from player 2's perspective, the previous move has been added to the table

If a piece is taken, it's shared pointer is reassigned to the piece taking it by "`move_piece`", the taken piece is then automatically destroyed, it's static data is updated by the destructor, this is shown in figure (12).

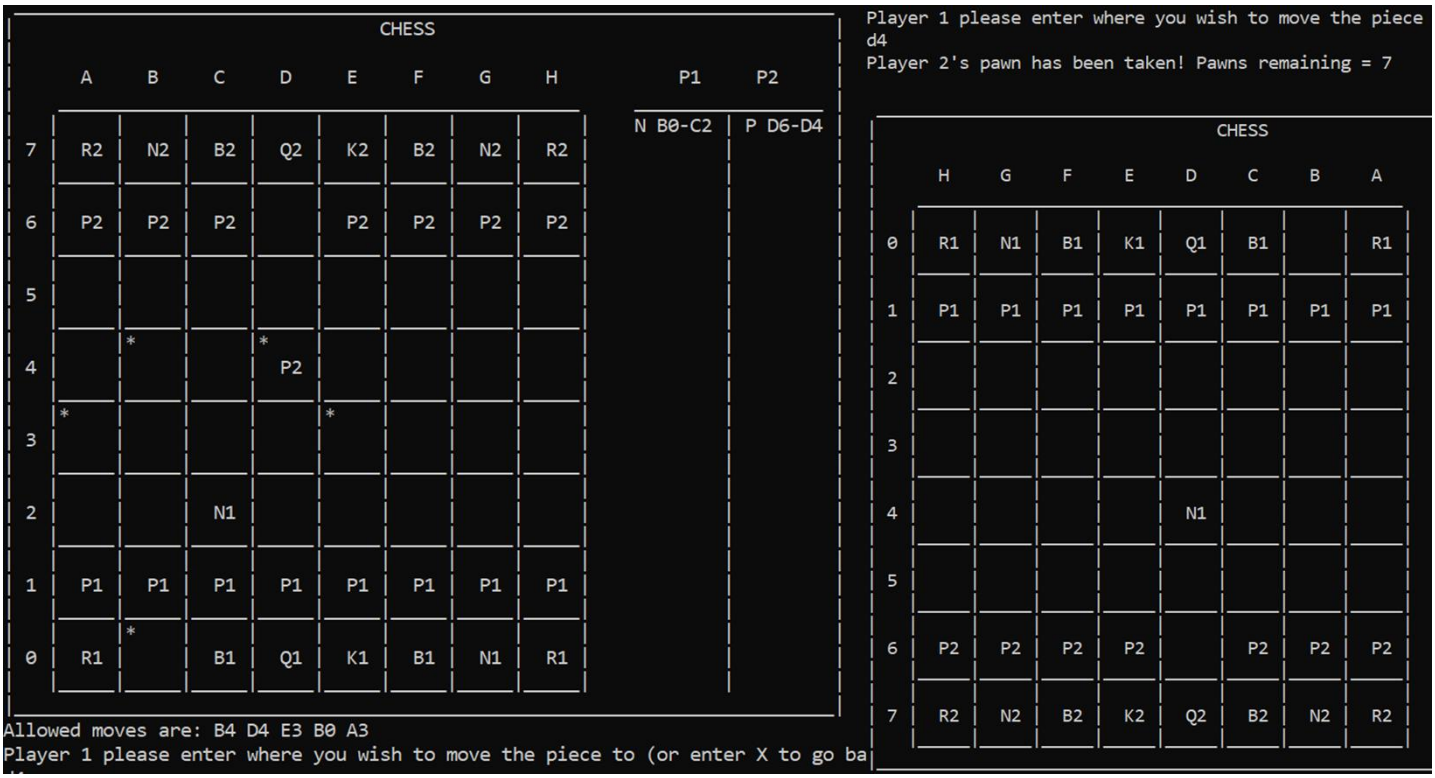

Figure 12 – A piece being taken, static data is updated and outputted by the destructor (upper right).

Promotions also update static data; this is shown in figure (13).

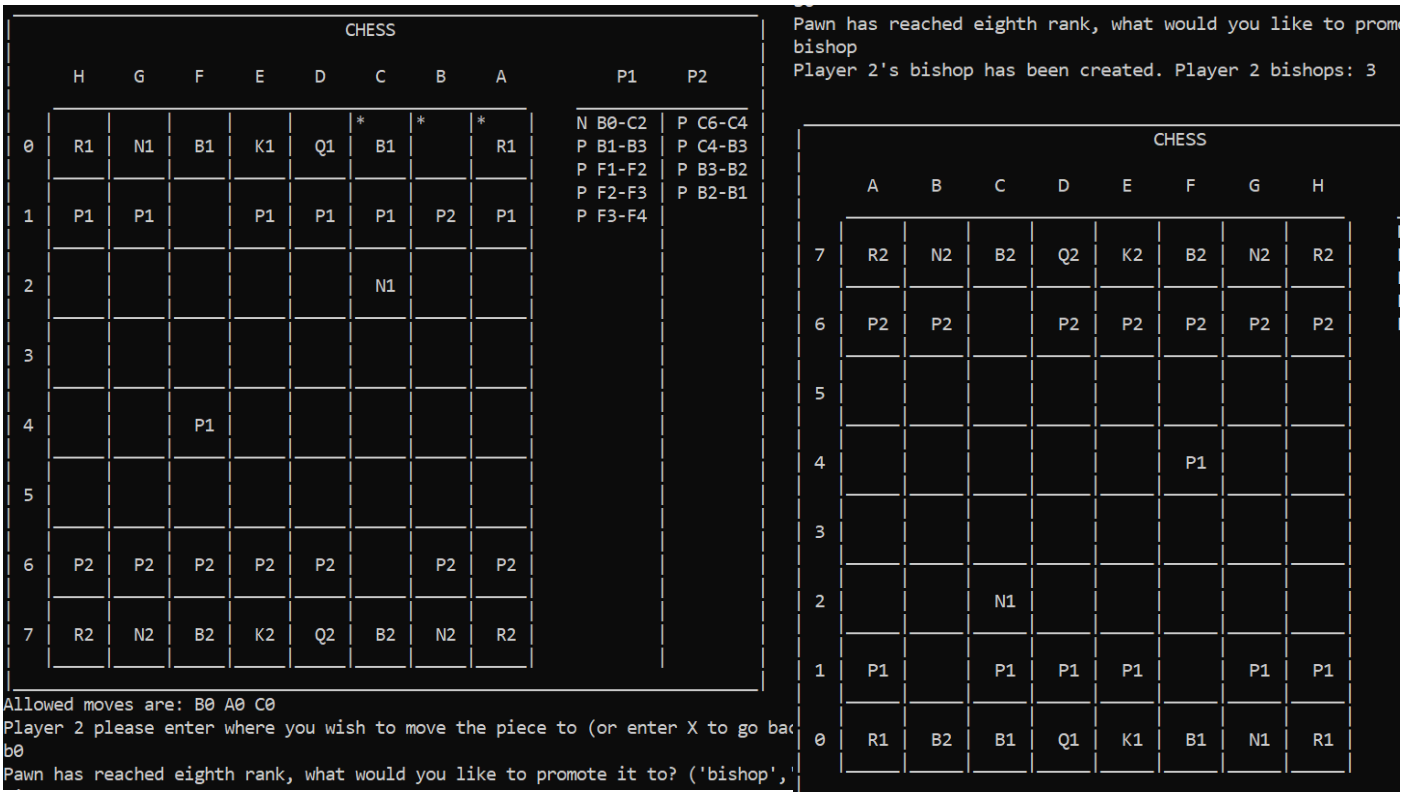

Figure 13 – A pawn being promoted, static data is updated and outputted by the constructor (upper right).

If the opponent is in check but still has legal moves, the programme will output that they are in check. They must then move themselves out of check. This is shown in figure (14)

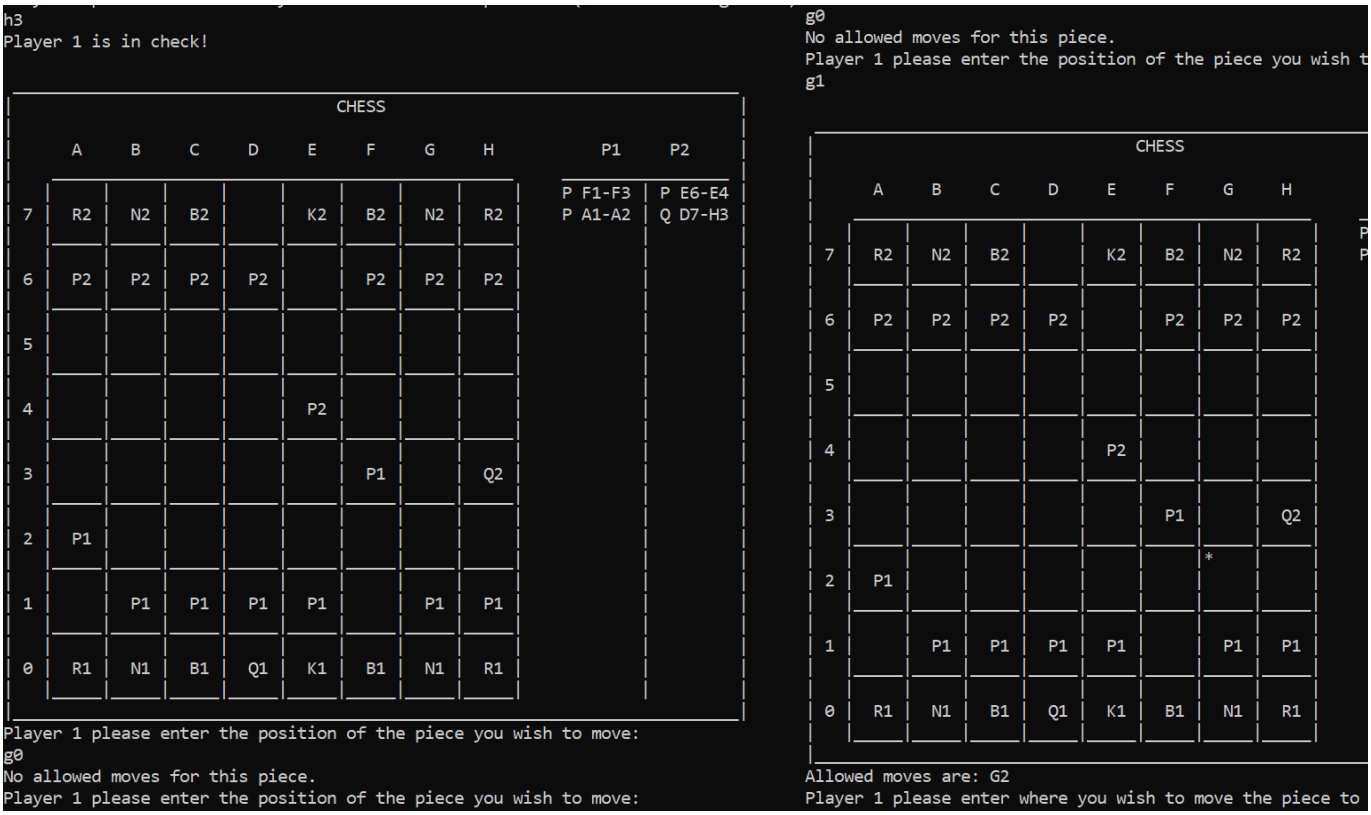

Figure 14 –A player in check. The player must move themselves out of check.

The game ends when the next player has no legal moves. If the player is in check, then they are checkmated, and the other player wins. If they are not in check, then the game is a stalemate. In both cases the player will be asked if they want to play another game, this is shown in figure (15).

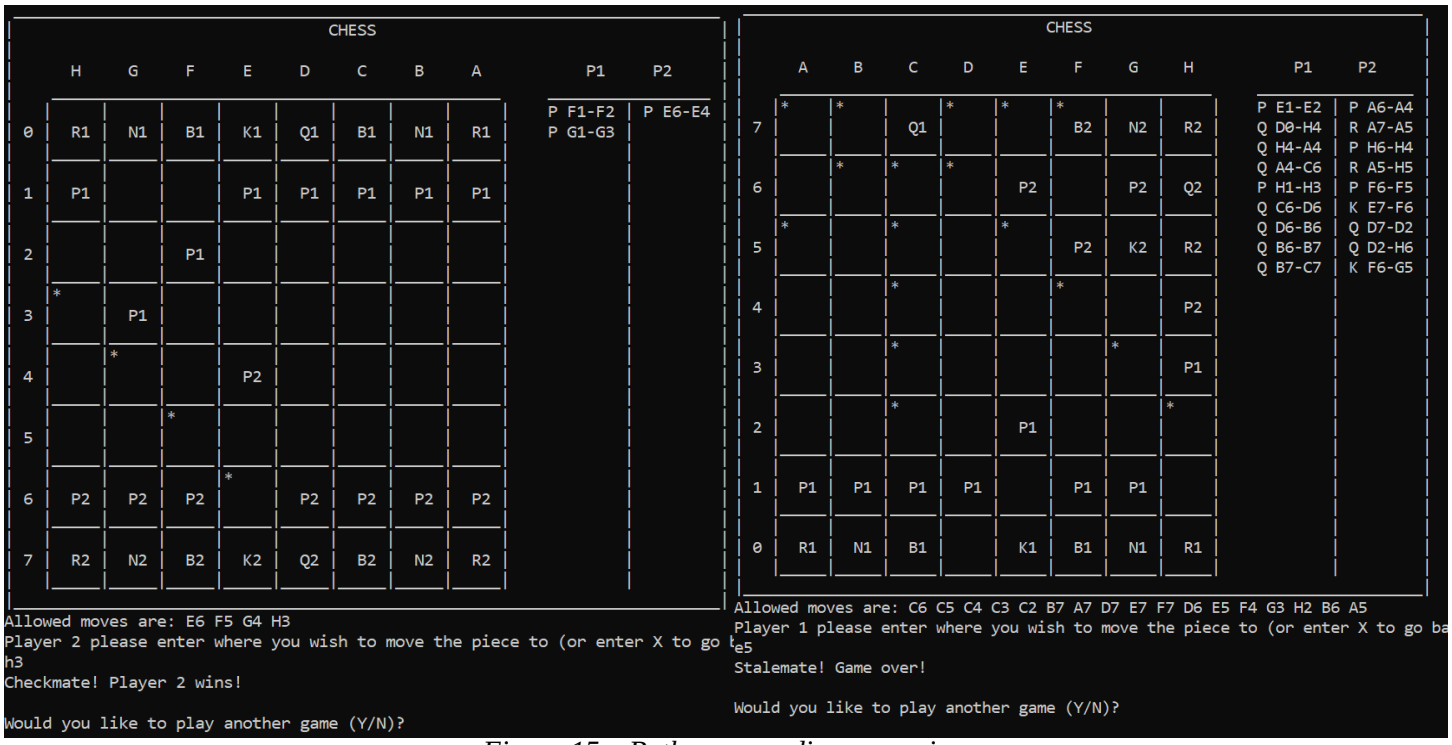

Figure 15 – Both game ending scenarios.

4 Discussion

The programme successfully recreates a game of chess for 2 players using principles of OOP including polymorphism, inheritance, objects, derived classes, abstract base classes, member data, and member functions. The programme can; return allowed moves for any piece (considering self-checking), output a representation of the board from 2 perspectives with previous moves and allowed moves shown, and determine the outcome of a move. The game uses advanced features of C++ including smart pointers, templates, and static data.

The programme could be developed further by using classes to describe players. Player objects could have member data detailing name, rank, and records of previous games including full move lists and game stats, which would be updated each game. A static data leader board could also be included.

Word Count: 2484

References

[1] M. Husband, D. Nguyen, S. Wong. Principles of Object-Oriented Programming. *University Press of Florida*, Sep. 2009.

[2] R. Pecinovsky. Learn Object Oriented Thinking and Programming. *Tomas Bruckner,* Nov. 2013.

[3] R. Reenskaug, P. Wold, O. A. Lehne. Working with objects The OOram Software Engineering Model. *Manning Publications Co,* 1996.

[4] https://en.cppreference.com/w/cpp/memory/shared_ptr. Accessed 06/05/2020

Appendix 1: Impact of COVID19 on the work completed and report

Lack of access to computers and the internet:
My accommodation only provides a small table and folding chair, I also only have a small laptop. This caused me neck and back pain after working for long periods.

Limitation in the outcome:
The pain limited the amount of time I could work for in one stint and made it harder to concentrate, meaning that I had less time to complete the report and was more error prone.