Creating a Console Chess Game in C++

Luis Miguel San Martín Fernández (9975891)

School of Physics and Astronomy

The University of Manchester

Object Oriented Programming in C++

May 2020

Abstract

The objective of this project was to program a playable 1v1 Chess Game in the Console. A wide range of C++ features were used, from smart pointers to lambda functions. Some additional features not studied in the course were also implemented such as enumerated classes and function try blocks for constructors. Given there are 6 different types of piece in Chess, each with a unique set of allowed moves, use of inheritance and polymorphism proved very useful. An abstract class interface for all the pieces was used, and the different pieces' methods were accessed using base class smart pointers to dynamic memory allocated pieces. A board class contained a vector of the unique pointers to all the pieces. All the methods used in the main function for the game to develop belong to the board class. The result is a chess game with visually appealing output which correctly implements the allowed moves of all the chess pieces, as well as piece captures. The next step would be to implement 'check', since currently the game ends when a king is captured.

1 Introduction

The choice of the board game *chess* was motivated by the high complexity of the game, which allowed me to learn and display the use of Object Oriented Programming (OOP) and advanced C++ features. The chess board is a 8×8 grid of alternating white and black squares. Columns are indicated with alphabetical letters from 'a' to 'h' while rows are indicated with numbers from 1 to 8. Tiles whose column and row add up to even and odd numbers are black and white respectively. There are two teams in opposite sides of the board: team white and team black. The pieces of the white team are placed in the first and second rows, while the black pieces are placed on the seventh and eighth. A total of 16 pieces constitutes each team: 8 pawns, 2 rooks, 2 knights, 2 bishops, a queen and a king. Teams alternate turns and they can move one piece per turn. It is standard for the white team to start. Moving a piece onto a tile occupied by a piece of the opposite team captures it, disabling its use for the rest of the game. Games take around 40 total moves on average [1], ending with one of the team's king being *checkmated*, with no possibility of escaping capture. The aim is to capture the other team's king whilst making a wise choice of moves to ensure your pieces are not captured by the enemy, which would reduce your chances of winning. Some pieces are more powerful than others due to the range of moves they are allowed to perform. These will be explained later in this document.

The code correctly implements the possible moves for all the different pieces, along with piece captures. It allows the user to get the allowed moves of all the pieces of a colour, as well as the allowed moves of any selected piece. A big effort was put into making the code efficient and the program secure from corruption or unhandled exceptions. A visually appealing console interface was developed, where not only the game-board is shown but also a history of moves and a cemetery containing the captured pieces.

2 Code design and implementation

This section is divided in 3 subsections to explain the 3 categories in which all the programme files are subdivided, as summarised in Table 1.

Helper files	**Chess classes**	**Implementation**
`coordinate_transforms`	`board`	`main.cpp`
`enum_attributes.h`	`piece`	
`allowed_moves_getter.h`	`pawn`	
`console_visualisation`	`rook`	
	`knight`	
	`bishop`	
	`queen`	
	`king`	

Table 1: *All the files which constitute the program, divided in three categories. Filenames with no extension specification include both '.h' and '.cpp'.*

Disclaimer: the code shown in the snippets that will follow in this report may have no comments. This was purposely done so that more code could fit in the screenshots. The actual code does contain appropriate comments.

2.1 Helper files

Helper files include those functions, classes or constant variables which are useful for the rest of the program but are independent and do not belong in any of the pieces or board classes.

2.1.1 Files `coordinate_transforms`

The contents of the header file `coordinate_transforms.h` (excluding the header guard and the includes) are shown in Figure 1. All the code is under the `namespace coordinates`. It declares constants and functions which facilitate working with locations in the board. A template function with two parameters for reversal of any `std::map`'s keys and values is defined. It uses a ranged `for` loop to iterate over the `std::pair`s making up the dictionary. A private `namespace` wraps a function to create a dictionary from `char` to `int` which loops over the characters of an alphabetical string to convert the columns to numbers. The anonymous namespace was used because this function was created to be used solely in this file for the instantiation of a constant alphabetical dictionary. A numerical dictionary is also created by inverting the alphabetical one. Four functions are declared whose definition is in the `coordinate_transforms.cpp` file. There is a function to check whether a location is within board range $0 < \text{location} \leq 64$; another one to convert from board coordinates, which are a pair of `char` and `int`, to a 1D location from 1 to 64; another one to do the opposite conversion; and finally one to get the column number given a 1D location.

The implementation (in .cpp file) of the functions which convert from board coordinates to 1D location and vice-versa are shown in Figure 2. The first one uses the alphabetical dictionary to get the column number from the `char` of the `std::pair`. To check a letter from `'a'` to `'h'` was inputted, the built-in member function `.at()` was used, which can throw the standard exception `std::out_of_range`. If this happens, the error is caught here, but a new error of the same type is thrown with a custom message on the `.what()` member function of the exception. The re-throw was done so that the stack can be unwinded and the error handled in a more appropriate place. A similar error is thrown if the row is out of bounds.

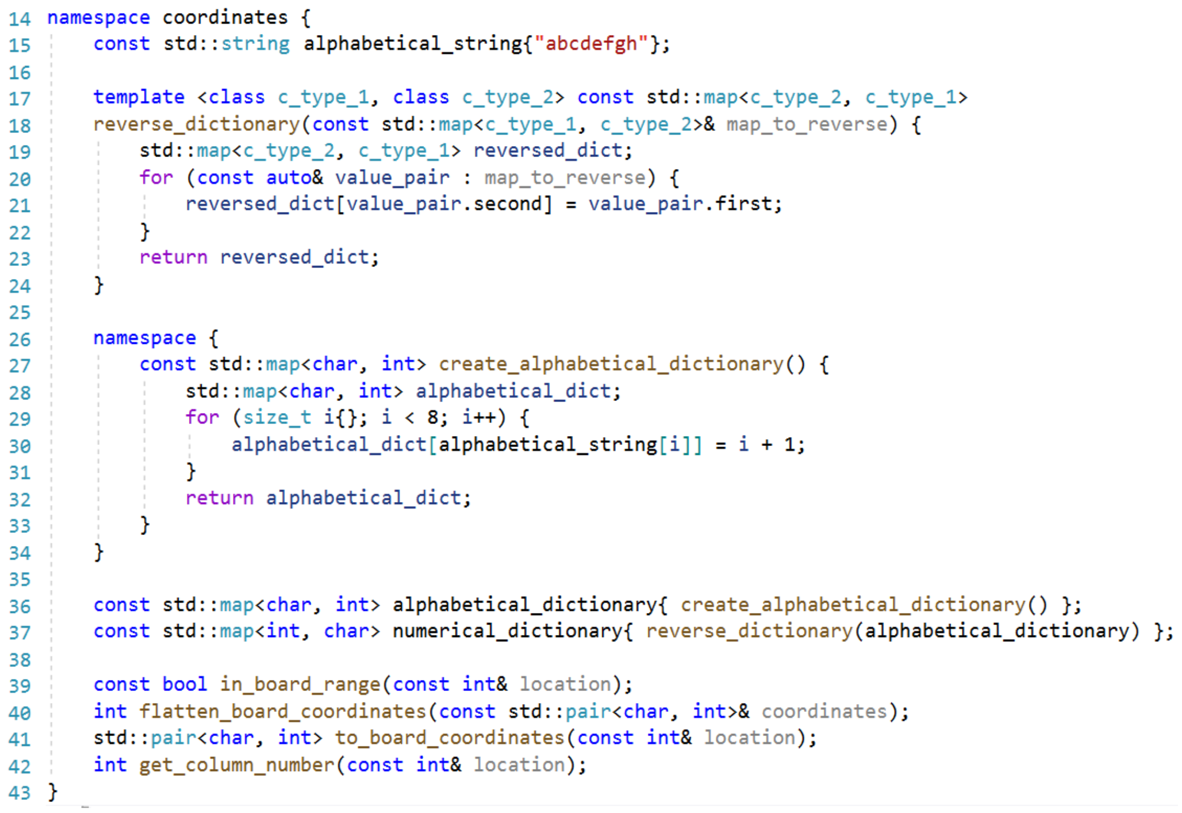
The function to convert from location to board coordinates makes use of integer division, modulo `%` operator and ternary operators to calculate the row and column. It then uses the numerical dictionary to convert the column number to `char`.

2.1.2 File `enum_attributes.h`

This header file defines three different enumerated classes for their use in different options of piece colour, piece symbol and board occupation. Using these classes brings the benefit of making the rest of the code more understandable. Objects of these classes can be static-casted to `int`eger values to facilitate their comparison. This header file also declares two constant dictionaries, one to obtain the opposite colour and another one to convert the piece colour to a `std::string`.

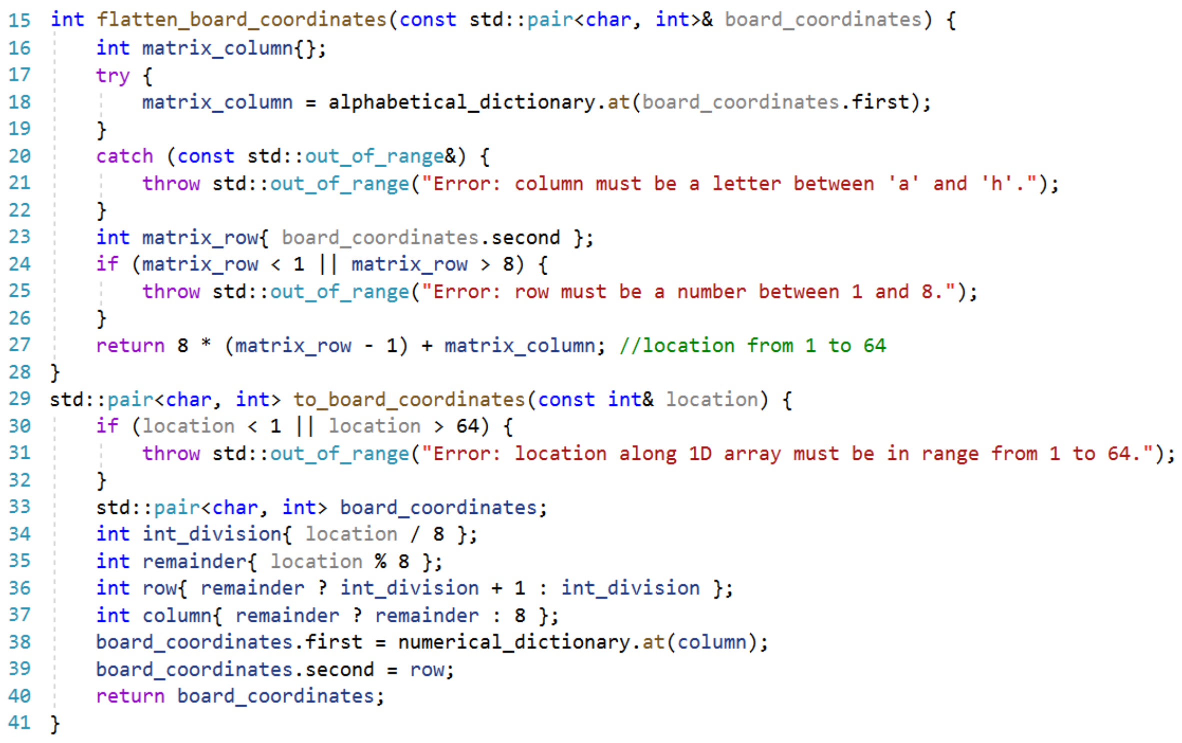
2.1.3 File `allowed_moves_getter.h`

This header declares and implements a template class which is used to get the allowed moves of the rook, bishop and queen pieces. The class takes takes as parameter an



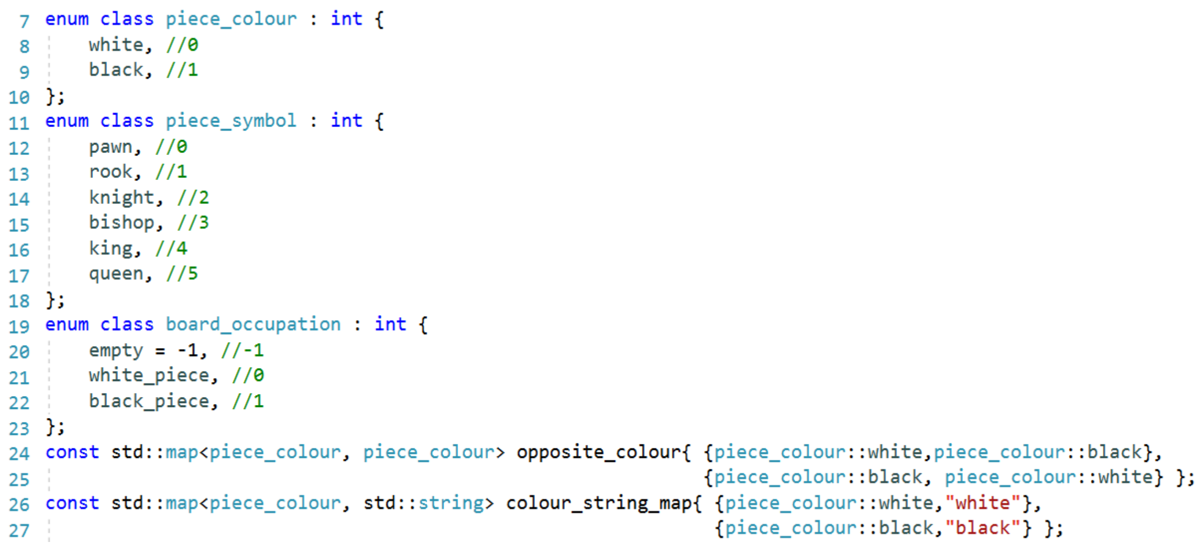
```cpp
namespace coordinates {
    const std::string alphabetical_string{"abcdefgh"};

    template <class c_type_1, class c_type_2> const std::map<c_type_2, c_type_1>
    reverse_dictionary(const std::map<c_type_1, c_type_2>& map_to_reverse) {
        std::map<c_type_2, c_type_1> reversed_dict;
        for (const auto& value_pair : map_to_reverse) {
            reversed_dict[value_pair.second] = value_pair.first;
        }
        return reversed_dict;
    }

    namespace {
        const std::map<char, int> create_alphabetical_dictionary() {
            std::map<char, int> alphabetical_dict;
            for (size_t i{}; i < 8; i++) {
                alphabetical_dict[alphabetical_string[i]] = i + 1;
            }
            return alphabetical_dict;
        }
    }

    const std::map<char, int> alphabetical_dictionary{ create_alphabetical_dictionary() };
    const std::map<int, char> numerical_dictionary{ reverse_dictionary(alphabetical_dictionary) };

    const bool in_board_range(const int& location);
    int flatten_board_coordinates(const std::pair<char, int>& coordinates);
    std::pair<char, int> to_board_coordinates(const int& location);
    int get_column_number(const int& location);
}
```

Figure 1: *Contents of the* `coordinate_transforms.h` *header file. The template function to reverse a dictionary is split in 2 lines to fit the screenshot to the margins of this document.*



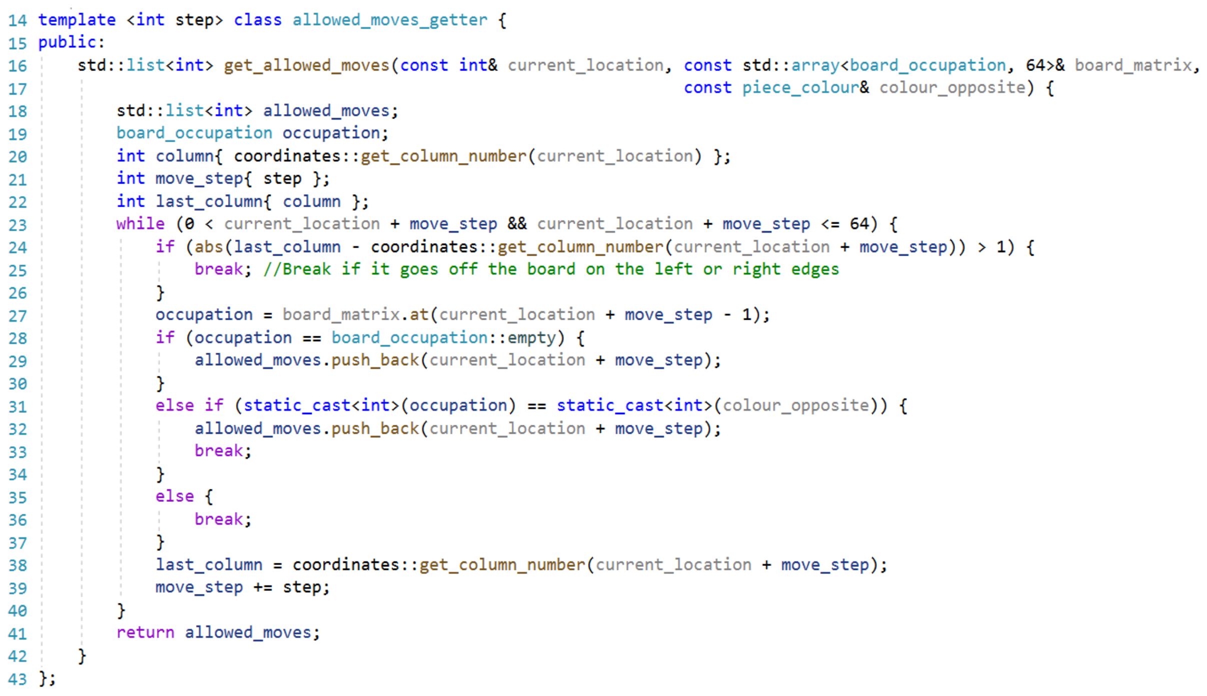
```cpp
int flatten_board_coordinates(const std::pair<char, int>& board_coordinates) {
    int matrix_column{};
    try {
        matrix_column = alphabetical_dictionary.at(board_coordinates.first);
    }
    catch (const std::out_of_range&) {
        throw std::out_of_range("Error: column must be a letter between 'a' and 'h'.");
    }
    int matrix_row{ board_coordinates.second };
    if (matrix_row < 1 || matrix_row > 8) {
        throw std::out_of_range("Error: row must be a number between 1 and 8.");
    }
    return 8 * (matrix_row - 1) + matrix_column; //location from 1 to 64
}
std::pair<char, int> to_board_coordinates(const int& location) {
    if (location < 1 || location > 64) {
        throw std::out_of_range("Error: location along 1D array must be in range from 1 to 64.");
    }
    std::pair<char, int> board_coordinates;
    int int_division{ location / 8 };
    int remainder{ location % 8 };
    int row{ remainder ? int_division + 1 : int_division };
    int column{ remainder ? remainder : 8 };
    board_coordinates.first = numerical_dictionary.at(column);
    board_coordinates.second = row;
    return board_coordinates;
}
```

Figure 2: *Implementation in* `coordinate_transforms.cpp` *of the functions to convert from board coordinates to 1D location and vice-versa.*



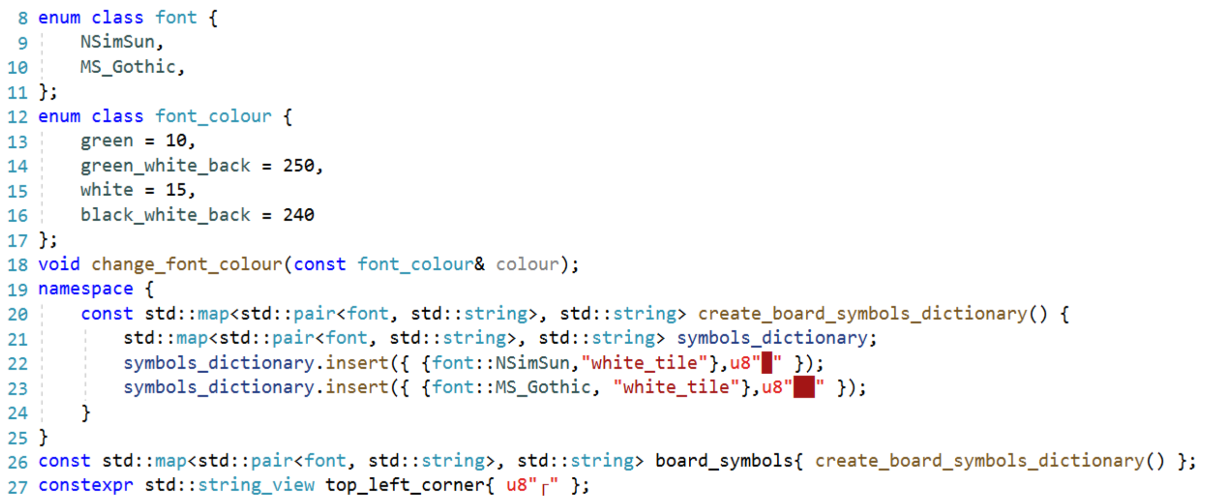
```cpp
enum class piece_colour : int {
    white, //0
    black, //1
};
enum class piece_symbol : int {
    pawn, //0
    rook, //1
    knight, //2
    bishop, //3
    king, //4
    queen, //5
};
enum class board_occupation : int {
    empty = -1, //-1
    white_piece, //0
    black_piece, //1
};
const std::map<piece_colour, piece_colour> opposite_colour{ {piece_colour::white,piece_colour::black},
                                                            {piece_colour::black, piece_colour::white} };
const std::map<piece_colour, std::string> colour_string_map{ {piece_colour::white,"white"},
                                                             {piece_colour::black,"black"} };
```

Figure 3: *Contents of the* `enum_attributes.h` *header file.*



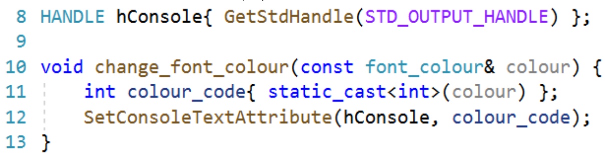
```cpp
template <int step> class allowed_moves_getter {
public:
    std::list<int> get_allowed_moves(const int& current_location, const std::array<board_occupation, 64>& board_matrix,
                                                                   const piece_colour& colour_opposite) {
        std::list<int> allowed_moves;
        board_occupation occupation;
        int column{ coordinates::get_column_number(current_location) };
        int move_step{ step };
        int last_column{ column };
        while (0 < current_location + move_step && current_location + move_step <= 64) {
            if (abs(last_column - coordinates::get_column_number(current_location + move_step)) > 1) {
                break; //Break if it goes off the board on the left or right edges
            }
            occupation = board_matrix.at(current_location + move_step - 1);
            if (occupation == board_occupation::empty) {
                allowed_moves.push_back(current_location + move_step);
            }
            else if (static_cast<int>(occupation) == static_cast<int>(colour_opposite)) {
                allowed_moves.push_back(current_location + move_step);
                break;
            }
            else {
                break;
            }
            last_column = coordinates::get_column_number(current_location + move_step);
            move_step += step;
        }
        return allowed_moves;
    }
};
```

Figure 4: *Contents of the* `allowed_moves_getter.h` *header file.*



```
enum class font {
    NSimSun,
    MS_Gothic,
};
enum class font_colour {
    green = 10,
    green_white_back = 250,
    white = 15,
    black_white_back = 240
};
void change_font_colour(const font_colour& colour);
namespace {
    const std::map<std::pair<font, std::string>, std::string> create_board_symbols_dictionary() {
        std::map<std::pair<font, std::string>, std::string> symbols_dictionary;
        symbols_dictionary.insert({ {font::NSimSun,"white_tile"},u8"█" });
        symbols_dictionary.insert({ {font::MS_Gothic, "white_tile"},u8"██" });
    }
}
const std::map<std::pair<font, std::string>, std::string> board_symbols{ create_board_symbols_dictionary() };
constexpr std::string_view top_left_corner{ u8"┌" };
```

(a) Header



```
HANDLE hConsole{ GetStdHandle(STD_OUTPUT_HANDLE) };

void change_font_colour(const font_colour& colour) {
    int colour_code{ static_cast<int>(colour) };
    SetConsoleTextAttribute(hConsole, colour_code);
}
```

(b) Cpp

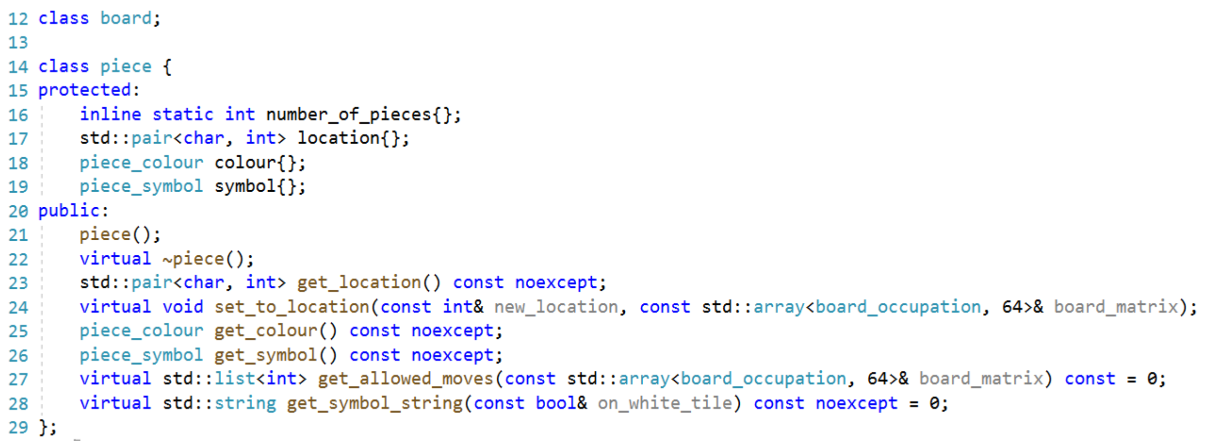
Figure 5: *Header and implementation of* `console_visualisation`.

`int`eger, which is the step indicating the direction of motion of the piece. The class contains a single member function which takes as argument the current location and opposite colour to the piece whose allowed moves to get, as well as the board matrix.

The member function contains a `while` loop which iterates over the allowed moves until a location which is out of bounds or occupied by a piece is found. If a location occupied by the opposite colour is found, the move is recorded to allow for capture. If occupied by a piece of the same colour, it `break`s right away.

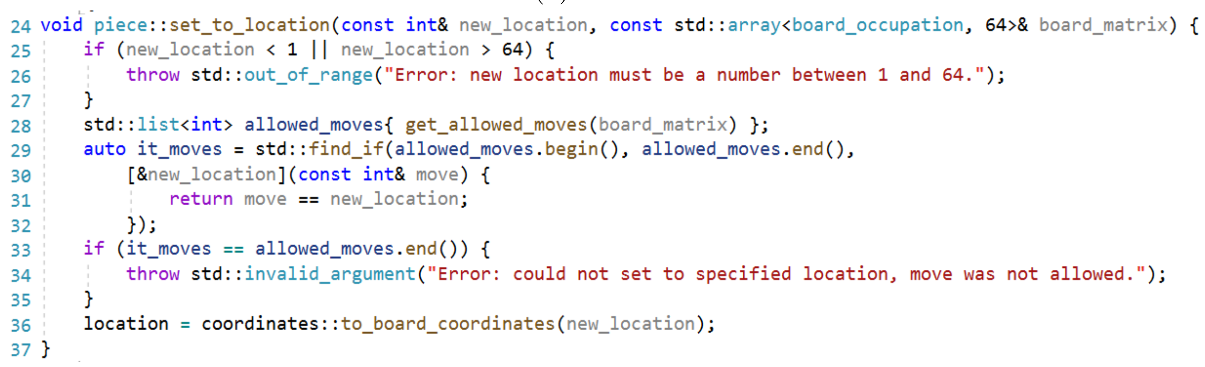
2.1.4 Files `console_visualisation`

The header file is shown in Figure 5a). It defines two enumerated classes which are used for the options of the type of font and its colour. A function to change colour is declared, whose implementation is shown in Figure 5b). It uses the output handle from the `Windows.h` header file. The header file has a private namespace with a definition of a function to create a `std::map` of Unicode symbols which takes as argument a pair of `font` type and `std::string` and returns a UTF-8 `std::string`. Only two of the dictionary entries are shown in the snippet for brevity. This dictionary is necessary because the two fonts which support Unicode symbols, which are **NSimSun** and **MS_Gothic**, output some Unicode symbols differently. This function is used to instantiate a dictionary which is called board symbols since all the symbols are to be used in the output from the `board` class, as will be discussed later. Symbols which are outputted the same using either font are also defined here as a `constexpr std::string_view` variables. Only one of them is shown at the bottom of Figure 5a).



```cpp
class board;

class piece {
protected:
    inline static int number_of_pieces{};
    std::pair<char, int> location{};
    piece_colour colour{};
    piece_symbol symbol{};
public:
    piece();
    virtual ~piece();
    std::pair<char, int> get_location() const noexcept;
    virtual void set_to_location(const int& new_location, const std::array<board_occupation, 64>& board_matrix);
    piece_colour get_colour() const noexcept;
    piece_symbol get_symbol() const noexcept;
    virtual std::list<int> get_allowed_moves(const std::array<board_occupation, 64>& board_matrix) const = 0;
    virtual std::string get_symbol_string(const bool& on_white_tile) const noexcept = 0;
};
```

(a) Header



```cpp
void piece::set_to_location(const int& new_location, const std::array<board_occupation, 64>& board_matrix) {
    if (new_location < 1 || new_location > 64) {
        throw std::out_of_range("Error: new location must be a number between 1 and 64.");
    }
    std::list<int> allowed_moves{ get_allowed_moves(board_matrix) };
    auto it_moves = std::find_if(allowed_moves.begin(), allowed_moves.end(),
        [&new_location](const int& move) {
            return move == new_location;
        });
    if (it_moves == allowed_moves.end()) {
        throw std::invalid_argument("Error: could not set to specified location, move was not allowed.");
    }
    location = coordinates::to_board_coordinates(new_location);
}
```

(b) Cpp

Figure 6: *Header and member function implementation of the piece class.*

2.2 Chess classes

2.2.1 Piece

The class piece is an abstract interface for all the pieces. Its header is shown in Figure 6a). It contains as member data a `static` to count the number of pieces in game, a `std::pair` of board coordinates for the piece's location, and colour and symbol objects from the `enum_attributes.h` header. Member functions to get the location, the colour, the symbol and the symbol as a `std::string` are defined. The latter is a pure virtual function and takes as argument a `bool`ean variable which indicates whether the piece is located on a white or black tile of the board. A virtual function to set to a new location is defined which takes as argument the new location and a `std::array` of board occupations, which represents the board. The board is passed by a `const`ant reference to avoid unnecessary copies. A pure virtual function to get the allowed moves is defined which also takes the board matrix as argument. It returns a `std::list` of allowed locations the piece can move to.

Concerning the implementation of this class, it may be mentioned that the default constructor and virtual destructor increase and decrease the `static` counter of the number of pieces in game. The implementation of the member function to set to a new location is shown in Figure 6b). It calls the member function to get the allowed moves, and uses `std::find_if` from the algorithm header to iterate over it. A lambda function which captures the new piece location by reference is used to check whether it is an allowed move. If not, an `std::invalid_argument` error is thrown. If the move is allowed, the board coordinates member data is set to the `std::pair` of `char` and `int` obtained by applying the function to convert to board coordinates defined in the `coordinates` file.

2.2.2 Specialised pieces

All the piece specialisations in Table 1 set the base class to `public` in their class declaration. They overload the pure virtual member functions of the piece class. All the pieces have a parametrised contructor which takes as argument the column and the piece's colour.

The pawn can move by 1 tile either forward (by 2 tiles if in the starting position) or, if there is a piece to capture, diagonally. These positions are directly checked against the board matrix to get the allowed moves. The king can move by 1 tile in any direction around it (horizontal, vertical and diagonally). The knight can move in L-shapes to locations differing from the original by 1 row and 2 columns, or vice-versa. These last two pieces have their allowed moves checked using nested `for` loops to run over all the permutations of horizontal and vertical changes allowed. This is illustrated for the king in Figure 7a).

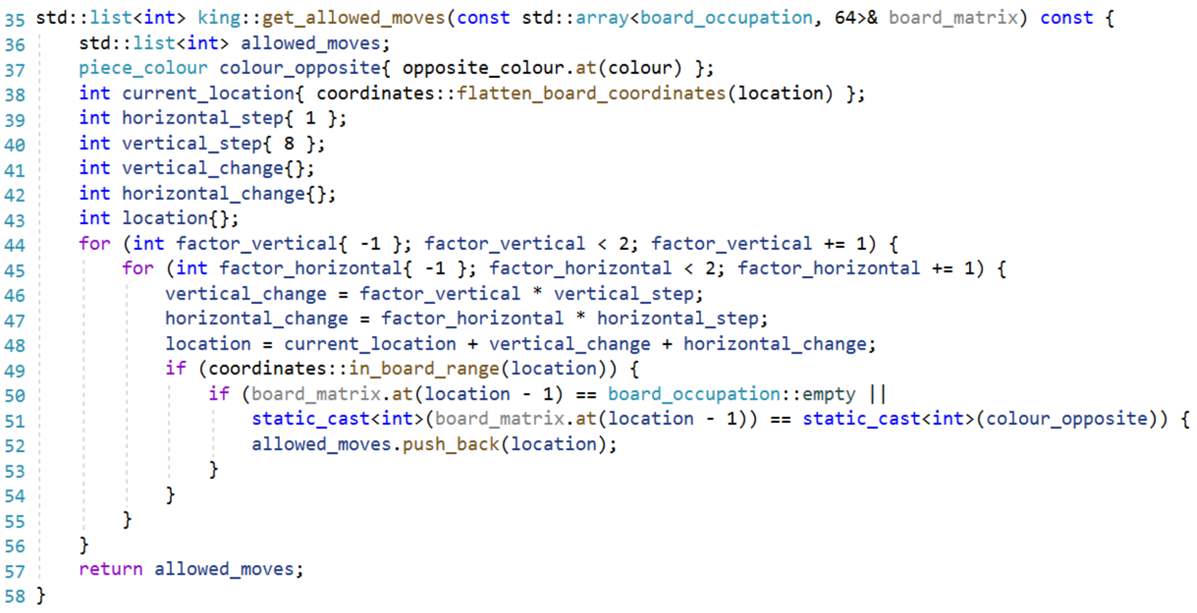
The rook, bishop and queen calculate their allowed moves using objects of the class in the `allowed_moves_getter.h` file instead. The example for rook is shown in Figure 7b). The `std::list` built-in member function `splice` was used to 'steal' all the allowed moves calculated with the getters. The rook moves in straight lines along rows and columns, the bishop moves along the four diagonal directions and the queen combines the moves of these two. These pieces can keep moving until they encounter a piece or the edge of the board.

2.2.3 Board

The board contains all the pieces and enables most of the functionality of the game. Its header file is shown in Figure 8. The pieces in-game and captured are separately contained in vectors of unique base class pointers. Since many times one only needs to know what the occupation of the board is, without needing access to the pieces themselves, the board matrix was coded separately as a `std::array` of 64 board occupation objects (from the `enum_attributes.h` helper file). This makes the code more efficient and safe. If an individual piece needs to be accessed, a dictionary can be used which takes in the board coordinates of the desired piece and returns its index in the piece vector. There is also a member data which is a vector of `std::string`s to record the history of moves.

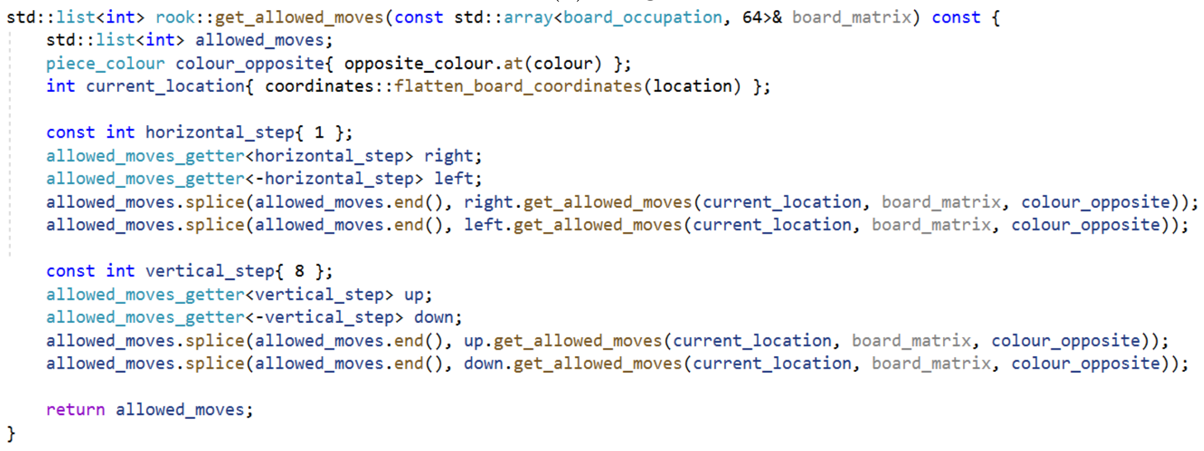
There are member functions to get and set the occupation of the board matrix. Other methods obtain the allowed moves of one or all the pieces, set the location of a given piece to a new one, or capture a piece sending it to the cemetery. Finally, a member function to show the board has two overloads, allowing to highlight a selected piece in a certain location.

The implementation of the default board constructor is shown in Figure 9b). It uses an initialiser list to pre-set the size of the in-game pieces vector (memory reservation). Everything is wrapped in a "function `try` block" to catch bad memory allocations and other exceptions so that a message can be outputted and then the error re-thrown to unwind the stack. A template class, shown in Figure 9a), wrapped in an anonymous namespace, is used to fill the vector of pieces column by column. The pieces are constructed using dynamic memory allocation with `std::make_unique`. The board matrix and piece index dictionary are filled in a `for` loop over the 64 1D locations.



```cpp
std::list<int> king::get_allowed_moves(const std::array<board_occupation, 64>& board_matrix) const {
    std::list<int> allowed_moves;
    piece_colour colour_opposite{ opposite_colour.at(colour) };
    int current_location{ coordinates::flatten_board_coordinates(location) };
    int horizontal_step{ 1 };
    int vertical_step{ 8 };
    int vertical_change{};
    int horizontal_change{};
    int location{};
    for (int factor_vertical{ -1 }; factor_vertical < 2; factor_vertical += 1) {
        for (int factor_horizontal{ -1 }; factor_horizontal < 2; factor_horizontal += 1) {
            vertical_change = factor_vertical * vertical_step;
            horizontal_change = factor_horizontal * horizontal_step;
            location = current_location + vertical_change + horizontal_change;
            if (coordinates::in_board_range(location)) {
                if (board_matrix.at(location - 1) == board_occupation::empty ||
                    static_cast<int>(board_matrix.at(location - 1)) == static_cast<int>(colour_opposite)) {
                    allowed_moves.push_back(location);
                }
            }
        }
    }
    return allowed_moves;
}
```

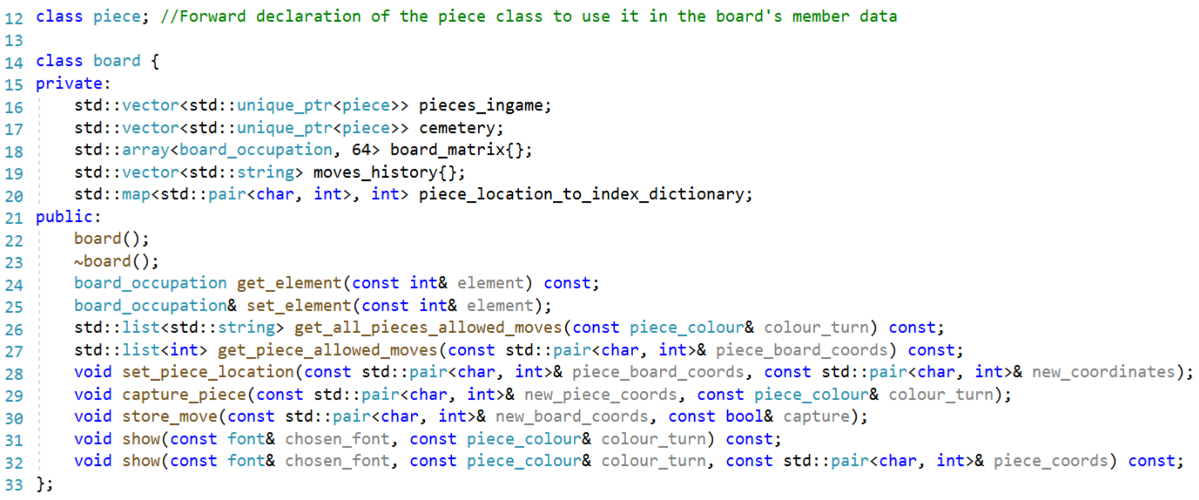
(a) King



```cpp
std::list<int> rook::get_allowed_moves(const std::array<board_occupation, 64>& board_matrix) const {
    std::list<int> allowed_moves;
    piece_colour colour_opposite{ opposite_colour.at(colour) };
    int current_location{ coordinates::flatten_board_coordinates(location) };

    const int horizontal_step{ 1 };
    allowed_moves_getter<horizontal_step> right;
    allowed_moves_getter<-horizontal_step> left;
    allowed_moves.splice(allowed_moves.end(), right.get_allowed_moves(current_location, board_matrix, colour_opposite));
    allowed_moves.splice(allowed_moves.end(), left.get_allowed_moves(current_location, board_matrix, colour_opposite));

    const int vertical_step{ 8 };
    allowed_moves_getter<vertical_step> up;
    allowed_moves_getter<-vertical_step> down;
    allowed_moves.splice(allowed_moves.end(), up.get_allowed_moves(current_location, board_matrix, colour_opposite));
    allowed_moves.splice(allowed_moves.end(), down.get_allowed_moves(current_location, board_matrix, colour_opposite));

    return allowed_moves;
}
```

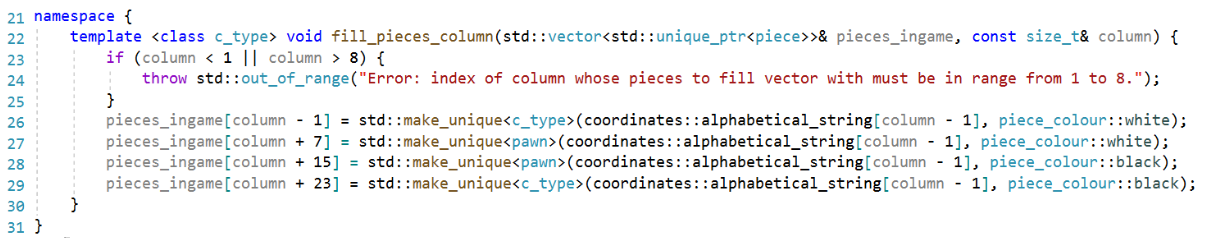
(b) Rook

Figure 7: *Member functions to get the allowed moves of the king and the rook.*



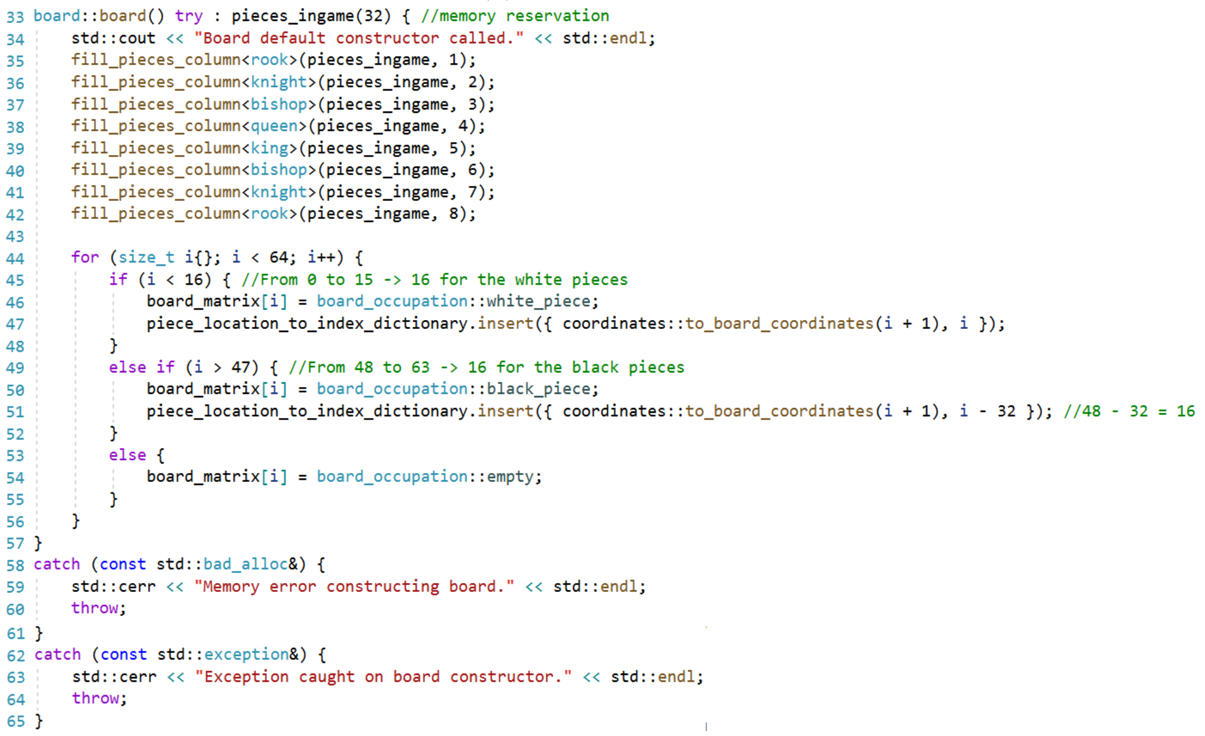
```cpp
class piece; //Forward declaration of the piece class to use it in the board's member data

class board {
private:
    std::vector<std::unique_ptr<piece>> pieces_ingame;
    std::vector<std::unique_ptr<piece>> cemetery;
    std::array<board_occupation, 64> board_matrix{};
    std::vector<std::string> moves_history{};
    std::map<std::pair<char, int>, int> piece_location_to_index_dictionary;
public:
    board();
    ~board();
    board_occupation get_element(const int& element) const;
    board_occupation& set_element(const int& element);
    std::list<std::string> get_all_pieces_allowed_moves(const piece_colour& colour_turn) const;
    std::list<int> get_piece_allowed_moves(const std::pair<char, int>& piece_board_coords) const;
    void set_piece_location(const std::pair<char, int>& piece_board_coords, const std::pair<char, int>& new_coordinates);
    void capture_piece(const std::pair<char, int>& new_piece_coords, const piece_colour& colour_turn);
    void store_move(const std::pair<char, int>& new_board_coords, const bool& capture);
    void show(const font& chosen_font, const piece_colour& colour_turn) const;
    void show(const font& chosen_font, const piece_colour& colour_turn, const std::pair<char, int>& piece_coords) const;
};
```

Figure 8: *Contents of the* `board.h` *header file.*



```cpp
namespace {
    template <class c_type> void fill_pieces_column(std::vector<std::unique_ptr<piece>>& pieces_ingame, const size_t& column) {
        if (column < 1 || column > 8) {
            throw std::out_of_range("Error: index of column whose pieces to fill vector with must be in range from 1 to 8.");
        }
        pieces_ingame[column - 1] = std::make_unique<c_type>(coordinates::alphabetical_string[column - 1], piece_colour::white);
        pieces_ingame[column + 7] = std::make_unique<pawn>(coordinates::alphabetical_string[column - 1], piece_colour::white);
        pieces_ingame[column + 15] = std::make_unique<pawn>(coordinates::alphabetical_string[column - 1], piece_colour::black);
        pieces_ingame[column + 23] = std::make_unique<c_type>(coordinates::alphabetical_string[column - 1], piece_colour::black);
    }
}
```

(a) Helper function



```cpp
board::board() try : pieces_ingame(32) { //memory reservation
    std::cout << "Board default constructor called." << std::endl;
    fill_pieces_column<rook>(pieces_ingame, 1);
    fill_pieces_column<knight>(pieces_ingame, 2);
    fill_pieces_column<bishop>(pieces_ingame, 3);
    fill_pieces_column<queen>(pieces_ingame, 4);
    fill_pieces_column<king>(pieces_ingame, 5);
    fill_pieces_column<bishop>(pieces_ingame, 6);
    fill_pieces_column<knight>(pieces_ingame, 7);
    fill_pieces_column<rook>(pieces_ingame, 8);

    for (size_t i{}; i < 64; i++) {
        if (i < 16) { //From 0 to 15 -> 16 for the white pieces
            board_matrix[i] = board_occupation::white_piece;
            piece_location_to_index_dictionary.insert({ coordinates::to_board_coordinates(i + 1), i });
        }
        else if (i > 47) { //From 48 to 63 -> 16 for the black pieces
            board_matrix[i] = board_occupation::black_piece;
            piece_location_to_index_dictionary.insert({ coordinates::to_board_coordinates(i + 1), i - 32 }); //48 - 32 = 16
        }
        else {
            board_matrix[i] = board_occupation::empty;
        }
    }
}
catch (const std::bad_alloc&) {
    std::cerr << "Memory error constructing board." << std::endl;
    throw;
}
catch (const std::exception&) {
    std::cerr << "Exception caught on board constructor." << std::endl;
    throw;
}
```

(b) Default constructor

Figure 9: *Board construction.*

```cpp
void board::capture_piece(const std::pair<char, int>& new_piece_coords, const piece_colour& colour_turn) {
    int captured_piece_index;
    try {
        captured_piece_index = piece_location_to_index_dictionary.at(new_piece_coords);
    }
    catch (const std::out_of_range&) {
        throw std::out_of_range("Error: piece to capture was not found at the new board coordinates.");
    }
    std::unique_ptr<piece> captured_piece{ nullptr };
    try {
        captured_piece = std::move(pieces_ingame.at(captured_piece_index)); //steal data from piece vector
    }
    catch (const std::out_of_range&) {
        throw std::runtime_error("Internal error: no piece to capture was found at the index returned by the \
                                    piece_location_to_index_dictionary.");
    }
    if (captured_piece->get_colour() == colour_turn) { //piece to capture is of the same colour as the colour turn!
        throw captured_piece->get_colour();
    }
    if (captured_piece->get_symbol() == piece_symbol::king) { //king was captured! Throw symbol indicating game over
        throw captured_piece->get_symbol();
    }
    cemetery.push_back(std::move(captured_piece)); //give ownership of captured piece to cemetery
}
```

(a) Capture

```cpp
void board::store_move(const std::pair<char, int>& new_board_coords, const bool& capture) {
    int piece_index{};
    try {
        piece_index = piece_location_to_index_dictionary.at(new_board_coords);
    }
    catch (const std::out_of_range&) {
        throw std::out_of_range("Error: no moved piece to store move was found at the specified board coordinates.");
    }
    std::string piece_symbol_string;
    try {
        piece_symbol_string = pieces_ingame.at(piece_index)->get_symbol_string(false); //check piece at index exists
    }
    catch (const std::out_of_range&) {
        throw std::runtime_error("Internal error: no moved piece to store move was found at the index \
                                    returned by the piece_location_to_index_dictionary.");
    }
    std::ostringstream move_stringstream; //Use ostringstream to concatenate move onto a string looking like "♙ a2"
    move_stringstream << piece_symbol_string << " " << new_board_coords.first << new_board_coords.second;
    moves_history.push_back(move_stringstream.str());
    move_stringstream.str(""); //clear ostringstream
}
```

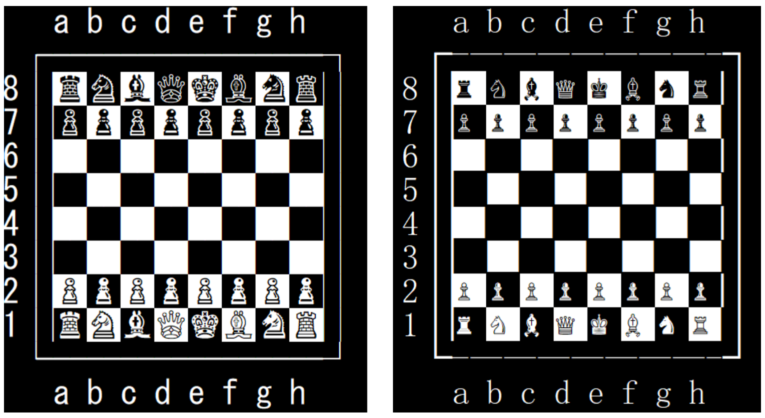
(b) Store move

Figure 10: *The implementation of two member functions of the board class.*

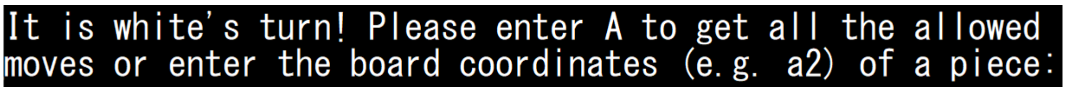
The implementation of the member functions to capture a piece and store a move are shown in Figure 10. The first one checks there is a piece to be captured and then uses move semantics to add it to the cemetery using `std::move`. If the captured piece is the king, its `piece_symbol` is thrown to indicate game over to main. The second one uses a `std::ostringstream` to record a move as a `std::string` including the piece's symbol and new board coordinates.

2.2.4 Main

The main function includes all the header files described in this report. Since board coordinates are used multiple times, at the top there is written `using board_coordinates_t` as a substitute for `std::pair` of `char` and `int`. All the errors that are thrown from any called functions are appropriately handled by either ending the program with `return` 1 or asking the user for new input. The main function is structured in 3 `while` loops, which run over the turns, the piece choice and the action choice respectively. Input handling is done by using `std::getline` and checking the resulting string has the expected contents.

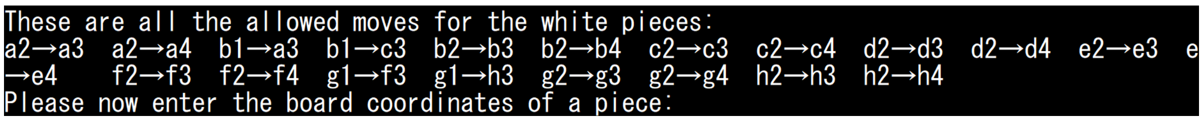


(a) Board



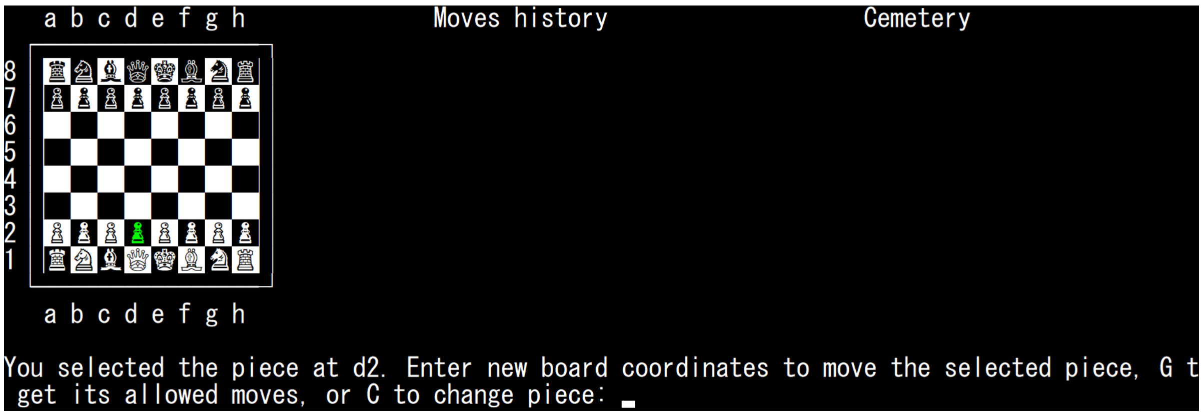
```
It is white's turn! Please enter A to get all the allowed
moves or enter the board coordinates (e.g. a2) of a piece:
```

(b) Initial options



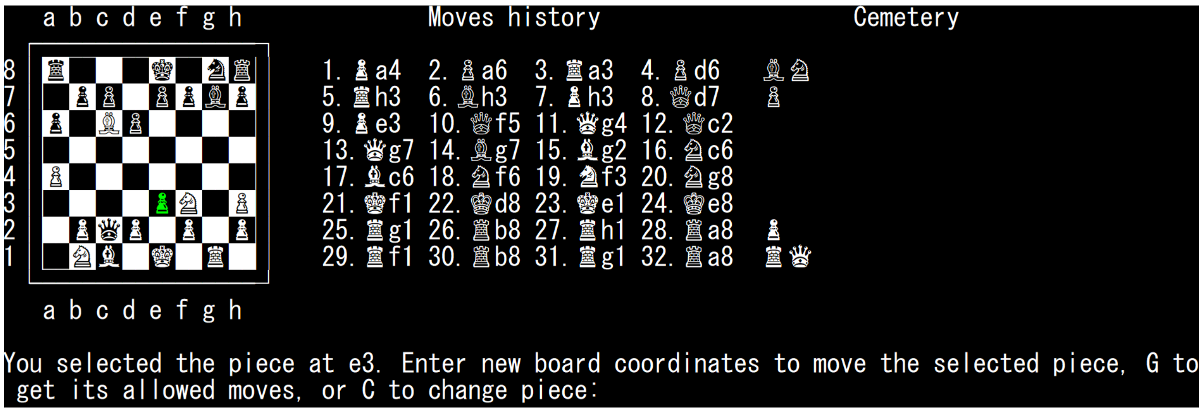
```
These are all the allowed moves for the white pieces:
a2→a3  a2→a4  b1→a3  b1→c3  b2→b3  b2→b4  c2→c3  c2→c4  d2→d3  d2→d4  e2→e3  e
→e4    f2→f3  f2→f4  g1→f3  g1→h3  g2→g3  g2→g4  h2→h3  h2→h4
Please now enter the board coordinates of a piece:
```

(c) All moves



```
 a b c d e f g h                Moves history                  Cemetery

You selected the piece at d2. Enter new board coordinates to move the selected piece, G t
 get its allowed moves, or C to change piece:
```

(d) Options after piece selection



```
 a b c d e f g h                Moves history                  Cemetery

                        1. ♟a4  2. ♙a6  3. ♜a3  4. ♙d6     ♗♘
                        5. ♜h3  6. ♗h3  7. ♟h3  8. ♕d7     ♙
                        9. ♟e3  10. ♕f5 11. ♛g4 12. ♕c2
                        13. ♛g7 14. ♗g7 15. ♝g2 16. ♘c6
                        17. ♝c6 18. ♘f6 19. ♞f3 20. ♘g8
                        21. ♚f1 22. ♔d8 23. ♚e1 24. ♔e8
                        25. ♜g1 26. ♖b8 27. ♜h1 28. ♖a8    ♟
                        29. ♜f1 30. ♖b8 31. ♜g1 32. ♖a8    ♜♛

You selected the piece at e3. Enter new board coordinates to move the selected piece, G to
 get its allowed moves, or C to change piece:
```

(e) Ingame

Figure 11: *Look of the board (a) for the MS Gothic (left) and NSimSun (right) fonts. The options at the beginning of a turn are shown in b) and d). An example of the result from choosing to get all the allowed moves is shown in c). Finally, an in-game view is shown in e).*

At the start of each turn, the allowed moves for all the pieces are obtained. If no moves are available, *stalemate* is declared and the game ends in a draw. Every time a new turn is reached or a valid piece/action choice is made, the console is cleared for better visualisation.

3 Results

The user is initially asked to change the console font to either **MS Gothic** or **NSimSun** since these are the fonts that support Unicode symbols. For the former and the latter, the board looks like the left and right respectively in Figure 11a). The user is then asked to select an option from the shown in Figure 11b). If `'A'` is chosen, all the allowed moves are shown as illustrated in Figure 11c). Once a piece is selected, the piece is highlighted in green in the board and new options are shown. An example is in Figure 11d).

After the game develops, the code would end up looking like in Figure 11e). The game will eventually end when one of the kings is captured.

4 Conclusion

This code successfully implements a visually appealing interface for playing chess in the console. It implements the allowed moves of the different pieces successfully, as well as piece captures. The code could be improved by the implementation of 'check', which is the situation in which the king is a position that would allow the opposite team to capture it in their next turn, and hence reduces the possible moves to only those that would cancel this capture. If none were available, the game would end. In my code, the game ends when a king is captured.

References

[1] Chess Statistics - www.chessgames.com/chessstats.html