



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics

Basics of Programming 2
(BMEV8IAA03, CS16D)

CPPCHESS DEVELOPER DOCUMENTATION

FINAL PROJECT

Instructor
Dr. Vaitkus Márton

Author
Diego Davidovich Gomes
NEPTUN Code: GHXLNI

May 13, 2025

Contents

1	Introduction	1
2	Solutions	1
2.1	General Design	1
2.2	External Libraries	1
3	Classes	2
3.1	Overview	2
3.2	Game	3
3.3	Board	3
3.4	Piece and the Derived Classes	4
3.5	Graphics	5
3.6	Texture	5
3.7	ChessGUI	6
4	Data Structures	6
4.1	Flattened Arrays	6
4.2	Attacking Boards	7
4.3	Use of Vectors	7
5	Key Algorithms	7
5.1	FEN Parser	7
5.2	Move Generation	8
5.3	Move Validation	10
5.4	Checkmate/Draw Detection	10
6	Testing and Verification	11
6.1	Defensive Programming	11
6.2	Manual Testing	11
7	Improvements and Extensions	12
7.1	Limitations	12
7.2	Future Work	13
8	Difficulties Encountered	13
9	Conclusion	14
10	Glossary	15
11	References	16

1 Introduction

Chess is a complex strategy board game for two players, played on a square board containing 64 squares and each player controls a different set of pieces [1]. It has a very well-defined set of rules and is regarded as one of the most complex games existent. Due to its complexity and different elements, it is a perfect choice to be implemented in software, especially using **Object-Oriented Programming**. As a computer engineering student and chess enthusiast, this project has the goal of developing a two-player virtual chess board using the C++ programming language, applying correctly the different principles of object-oriented programming to develop modular and maintainable code.

This document provides an overview of the CPPChess game, its implementation details, and the underlying logic to help developers understand the code. The document explains the solutions, classes, used data structures, algorithms and object-oriented concepts implemented.

2 Solutions

This section presents the high-level description of the program, focusing in the overall design and implementation approach.

2.1 General Design

Due to the dynamic nature of the game and a necessity for a visualization of the game state, the game has a graphical representation which is implemented using the **SDL2** library and the respective subsystem **SDL2_Mixer** for sounds, **SDL2_TTF** for text generation and **SDL2_Image** for importing images. This allows us to implement a complex graphical representation of the board and manage a variety of events. To add extra features and enable the board personalization, loading positions, reading the game history and checking crucial information, a graphical user interface is implemented using the **Dear ImGui** library.

The game was structured with the concepts of **encapsulation**, **data hiding** (partially), **abstraction**, **inheritance** and **polymorphism** in mind, with the goal of creating a strong modular, expandable, and maintainable system. Each class models a designated object with the appropriate methods and attributes.

2.2 External Libraries

As described, the project relies on the following third-party libraries for graphics, sound, text rendering, and graphical user interface implementation:

- **SDL2 v2.32.4**: <https://www.libsdl.org>, [2] [3]
- **SDL2_Image v2.8.8**: https://github.com/libsdl-org/SDL_image
- **SDL2_TTF v2.24.0**: https://github.com/libsdl-org/SDL_ttf
- **SDL2_Mixer v2.8.1_1**: https://github.com/libsdl-org/SDL_mixer
- **Dear ImGui v1.91.9**: <https://github.com/ocornut/imgui>, [4]

3 Classes

This section describes all classes used in the program, their relationships and their roles within the system architecture. The following subsections provide detailed specifications for each class, including its purpose, structure, and interactions with other components.

3.1 Overview

The program contains a total of **12 classes** implementing core chess functionalities. The classes can be divided into three groups:

- **Game State Classes:** responsible for representing the board, turn logic, move validation and rule enforcement. In this case, these are the **ChessGame** and **Board**
- **Piece Classes:** there is a base **Piece** class with each type being a derived class (**King**, **Queen**, etc...). The movement generation occurs by means of polymorphism with the use of a pure virtual function.
- **Interface Classes:** there are responsible for handling graphics, GUI — **Graphics**, **Texture** and **ChessGUI**

The **Unified Modelling Language (UML)** diagram below demonstrates the relationships between the classes [5] at a high level.

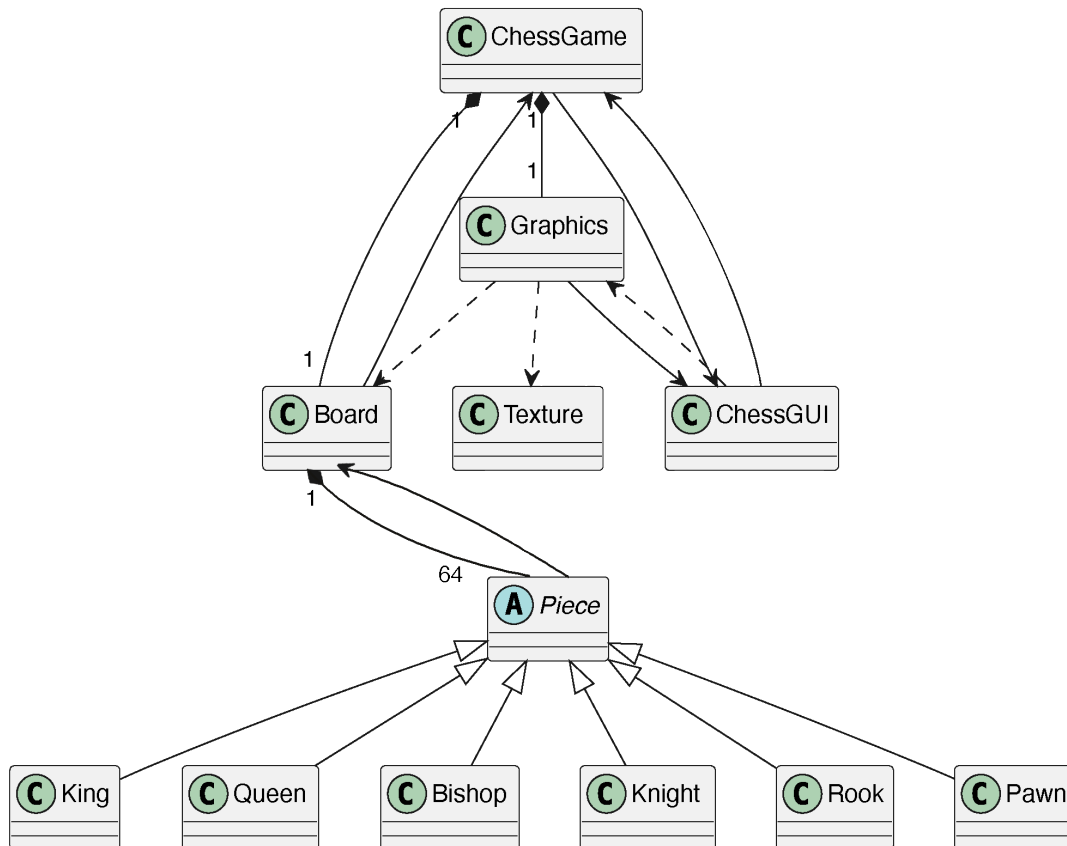


Figure 1: UML Diagram of the program (using PlantText)

Notation: Solid arrow = association, solid diamond = composition, dashed arrow = dependency, triangle arrow = inheritance.

3.2 Game

The `ChessGame` class serves as the central class for gameplay logic and state management in the application. It integrates the board representation (`Board board`) with the user interface components (`Graphics graphics` and `ChessGUI * gui`) while enforcing the rules and flow of a chess game. The class maintains the game's state, processes player moves and manages interactions between the graphical interface and core game mechanics.

This class contains several critical aspects of game management through its methods and member variables. It tracks the current game state using the `GameState` enumeration class, which has states like `Idle` when waiting for a player input, `PieceSelected` when a given piece has been chosen, `Dragging` to render the piece, `Processing` to validate, register and check for end of game conditions, and `GameOver` to handle when the game is over. These states shape the game's behavior. Turn management is done by maintaining a `turn` variable which alternates between `Color::White` and `Color::Black`, while at the same time, monitor and increment the `Half Move Clock`, used for the 50-move rule and `Full Move Clock` tracking the number of completed moves in the game.

Move processing and validation occur through coordination with the `Board` instance. When a player selects a piece, `focusIndex` records its position and potential moves are calculated ad hoc. And after the user selects a destination square, the `targetIndex` stores the destination square. The moves are performing the actual move and updating board states, turn switching and move history recording. All moves are recorded in `Standard Long Algebraic` notation through `std::vector<std::string> moveList`, enabling `Portable Game Notation` (PGN) file generation, which can be later used in more advanced platforms for game analysis.

User input processing happens through the `handleEvent()` method, which is called in `main`. It interprets SDL events to track mouse actions and piece selections. Alongside, the `handleRender()` method allows to use the `Graphics` to render the game states and animations at a steady frame rate (before, it was a reactive rendering which was triggered with events). A pointer to the `ChessGUI` allows communication with the GUI for displaying game information and user prompts. The game can be initialized with the standard chess position or by loading custom positions via the Forsyth-Edwards Notation (FEN) strings through the `loadFEN()` method and `resetGame()` functionality provides a way to restart the game.

The `handleGameOver()` method determines and provides notification about the outcome of game, while writing this information to the `moveList` and generating a PGN. In terms of design, it facilitates maintenance and future expansion, although some refactor would be useful to organize better the code.

3.3 Board

The `Board` class serves as the fundamental representation of the chessboard, encapsulating the piece placement and the logical rules to govern their movement. In its core, it uses a flattened `std::array<Piece *, 64>` to model the 8×8 grid through row-major index ($0 = a1$ to $63 = h8$). Alongside this, it maintains references to the white and black kings, which simplifies the tracking of the Kings for detecting check and checkmate con-

ditions. It also contains two attack boards, `whiteAttackBoard` and `blackAttackBoard`, which store information about the squares controlled by each side.

The `Board` is responsible for computing all legal moves across the board using `computeAllMove`, and for identifying which squares are attacked using `computeAttackBoards()`. When a piece is selected, `validateMovesForPiece()` determines which of its potential moves are legal, by performing a deep copy of the board using a copy constructor and executing the move in the temporary board. The main method, `movePiece()` performs after a move is determined valid, updating the board state accordingly and execute any special conditions like en passant or castling. The class can also detect whether the king is in check or whether any legal moves exist for a given color, which is used to determining game-end conditions (checkmate/stalemate).

Additionally, the board can be initialized by using the standard FEN string and it provided static attributes for conversion between index, row and column values, and standard algebraic notation to index and vice-versa

3.4 Piece and the Derived Classes

The `Piece` class serves as an **abstract base class** that defines the fundamental interface and common properties for all chess pieces, such as position, type and color. Using object-oriented **polymorphism**, it establishes a unified interface while allowing each piece type to implement its specific movement rules. The class includes the following protected attributes: a board position (0 to 63 index), a color (black or white), a piece type (pawn, knight, etc...), a pointer to the parent `Board` class and a `hasMoved` flag which is used for special moves like castling and pawn double-moves.

The class contains two pure virtual functions that all derived pieces must implement: the `computeMoves()` for generating possible moves based on each piece movement, and `clone()`, which clones a piece with all of its attribute to another location on the heap, to allow the deep copy of the board pieces and states, which is used for **move validation**. The public methods provide getters for all key properties and a shared method to verify whether a move is possible. A virtual destructor ensures proper cleanup of derived class instances.

Each piece class — `King`, `Queen`, `Rook`, `Knight`, `Bishop` and `Pawn`, inherits from this base class and implements type-specific behavior:

- Movement patterns defined via offsets
- Special move logic (en passant, castling)
- For the King, additional attributes and methods are implemented due to the more complex nature of the King according to the rules of the game

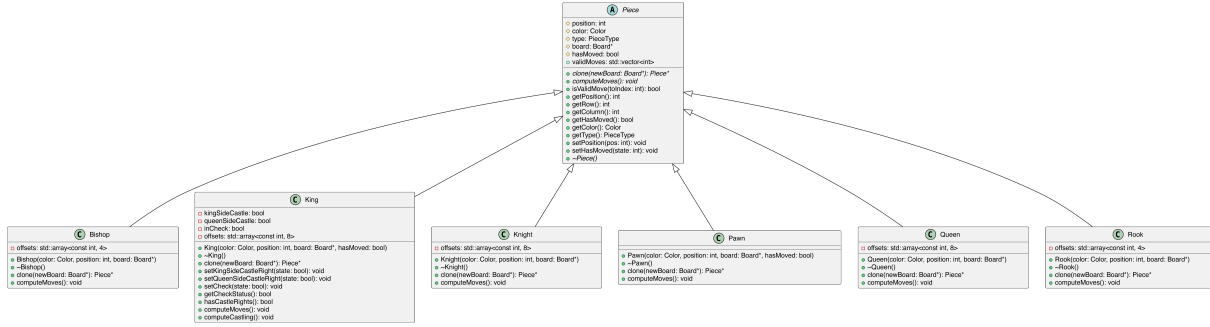


Figure 2: UML diagram of Piece and its derived classes, with attributes and methods. Access it in [PlantText](#)

3.5 Graphics

The **Graphics** class encapsulates and manages all the involved SDL2 subsystems. The class is designed considering the **Resource Allocation is Initialization** (RAII) principle, where the constructor creates and initializes all necessary SDL subsystems, while creating the window and renderer. Correspondingly, the destructor automatically handles the destruction of the window and renderer, deallocates resources and quits the SDL subsystems, ensuring no memory leaks or subsystem occur during program termination.

The class maintains essential SDL resources as private members, including pointers to the application window and renderer, alongside a pointer to **ChessGUI**, allowing it to render the GUI in certain scenarios. A key optimization is the use of an `std::array<SDL_Rect, 64>` **squares** to precompute and store the screen coordinates of all chessboard squares, which simplifies the rendering of the pieces and enables to easily flip the board by recomputing the positions without affecting the underlying game logic.

There are methods for rendering each visual element such as the pieces, the board squares, board markings, move highlight, capture highlight, square highlight and king in check markings. Due to the lack of layering in SDL, each method should be called in a specific order to create a desired effect. Additionally functionality includes methods for rendering events such as animating the piece moving from one square to another, render a selected piece on the same position of the mouse cursor or to flip the board.

3.6 Texture

The **Texture** class is responsible for facilitating the management of the SDL textures resources, handling both image assets and dynamically generated text surfaces. It automatically frees any loaded texture to prevent memory leaks. It has as methods, a `SDL_Texture` to a GPU loaded texture and the respective width and height dimensions in pixels.

It contains methods which create a GPU texture from a given image or text using a provided font. Then, by calling the respective method, it is possible to render the texture in a given position of the screen. This design allows to easily reuse textures such as of pieces. It is important to note that most of the textures are defined as **global variables**

in `Graphics.cpp`.

Example:

```
Texture pawn;  
pawn.loadTexture("White_Pawn.png", renderer);  
pawn.renderTexture(renderer, x, y);
```

3.7 ChessGUI

Manages and creates Graphical User Interface (GUI) subsystem using the **Dear ImGui** library, managing the complete lifecycle of ImGui resources including initialization and shutdown procedures. It contains a pointer to `ChessGame` to access information and perform some operations such as reading the move history, resetting, and loading the board with a position.

It provided three simple functions: initializing the ImGui-SDL rendering backend, rendering the primary interface with all interactive elements (menus, configuration options, and game controls), and displaying specialized end-game dialogs that prompt users to restart or exit.

4 Data Structures

4.1 Flattened Arrays

Instead of the more intuitive 2D Matrix, the chessboard is represented using a **flattened array** of `Piece*` with 64 elements, which converts the natural 8×8 matrix into a linear sequence through row-major ordering. This approach maps each square row and column to a single *index* via the formula $index = row \times 8 + column$. The inverse transformation, retrieving a square's coordinates from its index is simply done by using the formulas $row = index \div 8$ and $column = index \bmod 8$.


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

Figure 3: Indexes of the flattened array and the respective squares in the board

This approach allows offers a superior performance over a 2D matrix due to its **memory locality** and **computational efficiency**. It avoids the use of double pointers and it simplifies the move generation. It optimizes sequential memory access, making loops over the flat array faster [6].

4.2 Attacking Boards

Similarly to the approach in 4.1, the attacking board is a flattened array of `int` which contains the information of the squares being attacked by the pieces of a color. Two separate arrays, `whiteAttackBoard` and `blackAttackBoard` use binary values to encode attack states: `1` indicates a square under attack by the respective colors, while `0` denotes if square is safe. It is updated after each move and it is used to detect threats to the king and if the king is being checked using the `Board::isKingInCheck` method, where it verifies if the position of the King is marked as 1 in the enemy attack board. It is important that this can be later swapped to **bitboards**.

4.3 Use of Vectors

The `std::vector` container serves as the primary dynamic array implementation throughout the application.

- In **Piece** class: each derived piece (e.g. `Pawn`, `Knight`) has a `std::vector<int> validMoves` member, which is populated during the move generation to store target square indices. It allows pieces to report their pseudolegal moves to the board validation system and accomodating the variable amount of possible moves (e.g. a queen in general has more possible moves then a knight or a pawn).
- In **Game** class: it records the game's move history in **Extended Algebraic Notation**, which is later exported to a **PGN** file.

5 Key Algorithms

5.1 FEN Parser

The **Forsyth-Edwards Notation** (FEN) string initializes the board state by encoding piece positions, game metadata, and move rules [7]. A valid FEN must include all six fields and exactly one king per side. It contains the fields: piece placement, active color, castling rights, en passant, halfmove clock and fullmove counter. For example, the starting position has the following FEN:

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

The parsing is done by separating each field into a `std::string` and reading their contents:

1. **Piece Placement**: there are 8 rows separated by `/`, with lowercase letters representing black pieces and uppercase for white pieces. The digits 1 to 8 tells to the parser the amount of columns to be skipped. When a valid character (Kk, Qq, Rr, Nn, Bb, Pp) is read, the piece is **dynamically allocated** into the `std::array<Piece*, 64> board` with the respective color, type and position.

2. **Active Color:** **w** tells that is white's turn and **b** is black's turn.
3. **Castling Rights:** format **KQkq** (White/Black kingside/queenside) or **-** if none. A validation is performed to check if the pieces are in-place
4. **En Passant Target:** a square (e.g. **e3**) or **-** if none
5. **Halfmove Clock:** count of moves since last capture/pawn advance (for 50-move rule)
6. **Fullmove Counter:** increments after black's turn. Starts from 1

The parser has **exception handling** and it will fall back for the default board initialization. A visual example of the FEN can be seen in the figure below:



Figure 4: Piece placement in a FEN String [7]

5.2 Move Generation

As specified in the Class section, the Piece class is an abstract class containing the `computeMoves()` pure virtual function. This enables to each piece to generate the pieces in their own. As mentioned before, the chessboard is represented by using a flattened array and allows to easily generate the possible moves of a piece by using an `std::array` of `offsets`.

+7	+8	+9
-1	0	+1
-9	-8	-7

Figure 5: Offset values according to destination square [6]

So, for example, a bishop has offsets $[+9, +7, -9, -7]$ and it loops through every direction till the board is reached, a enemy piece is attacked or it collides with a friendly piece. Similarly, the rook has offsets $[+8, +1, -1, -8]$ and the queen has both of these offsets. One special piece might be the Knight, which has offsets $[-17, -15, -10, -6, 6, 10, 15, 17]$.

However, we should be careful with the side effect of using a flat array which is **wrapping** (when a piece can move to the complete opposite side of the board, e.g., from index 7 to index 8) or invalid address accessing.

								+9
+1						+7	+8	
-7						-1	0	
						-9	-8	

Figure 6: Index overflow [6]

This is easily fixed by calculating the difference between the initial column and the destination column, verifying if it does not have a jump in the value

5.3 Move Validation

To implement a range of rules and dynamics such as piece pinning, double checks, forks and even checkmate/stalemate detection, the system must generate fully **legal moves**, moves that do not leave the King in danger or in attack. To validate a move, several steps must be made in sequence:

1. **Pseudolegal move generation:** each individual piece type generates their pseudolegal moves, which are the moves that a piece can make without considering if the move leaves the king in check. However, only a subset of these moves are legal. It is important to note, to save some computations, the valid moves are generated when a piece is selected by the user.
2. **Test the Move:** after generating the list of moves for the pieces, we can select a **origin square** (the piece position) and **destination square** to test. The move is **executed in a copy** of the Board class, which is created dynamically with different addresses so no changes will occur to the actual game. The move is executed in the copy and then the attack boards are re-generated
3. **Verify if it leaves the King in Check:** after the execution of the move in the copy, it is verified that the move left or not the King in check. If so, the move is considered **invalid**, otherwise, the move is **valid**.

The move validation is implemented in the `Board` class through the following group of methods:

- `validateMove(int fromIndex, int toIndex)`: validates a single move
- `movePiece(int fromIndex, int toIndex)`: executes the move, by performing the necessary changes in the `std::array<Piece *, 64> board`, mostly by removing the piece from the origin square and placing into the destination square, and recomputing the attack boards and generate the pseudo-moves for every piece.
- `isKingInCheck(Color color)`: checks if the position of the King is marked as attacked in the enemy attacking board. Returns true if yes, otherwise it returns false
- `validateMovesForPiece(int position)`: goes through each possible move of the piece in `position` and validates it

5.4 Checkmate/Draw Detection

It is essential to be able to detect checkmate as it is a fundamental core concept and what ensures that the game always ends with some kind of result, instead of running forever. To detect checkmate or a possible draw condition, we must:

1. Check if the King has any possible moves. This is mostly done due to the fact that in most cases, the King will probably have a valid move and it will resolve right away

2. If there is no valid King moves, check if there is any legal moves by friendly pieces (pieces of same color). This is done by looping through the board searching for friendly pieces, computing their legal moves and checking if there is at least some legal move. If it exist, then it means that the it is still possible to play the game.
3. If there is no legal move remaining, the last step is to check if the King is in check or not. If it is, then it is a **checkmate**, otherwise it can be a **stalemate** (no legal moves but the King is not in check).

6 Testing and Verification

To ensure that the game is logically correct and follows the rules, testing and verification is essential. Due to the dynamic nature of the game, the testing and verification were primarily done through **visual inspection** and **interactive debugging**.

6.1 Defensive Programming

One major factor to avoid runtime errors is the application of a **defensive programming mindset**, which includes:

- A careful input/parameter validation and state checking before events, such as checking if the indexes are in bounds and avoiding nullptr dereferencing
- Structured error handling to prevent crashes or undefined behavior
- Early returns in case of unexpected states or invalid inputs

6.2 Manual Testing

Since the project included a graphical user interface using SDL and ImGui, the state of the board, piece positions, legal moves and possible states are displayed and can be easily visually verified, specially for anomalies (e.g. moves wrapping across the board).

The following approaches were used:

- **Move Testing:** Each chess piece's movement rules were manually tested by selecting the piece and observe what are the possible valid moves displayed on the screen.
- **Wrapping & Edge Cases:** board boundaries and piece interactions also were checked during interaction by moving the pieces around similarly to a real game and ensure they followed the rules of chess
- **States Validation:** each state and they corresponding functionality and interactions were manually tested
- **Game Outcome:** the different outcomes of the game were tested by manually playing the games to reach that state (e.g. checkmate, stalemate)



Figure 7: Invalid move of a King capturing the enemy Queen in the other side of the board, indicating issues with the move generation

7 Improvements and Extensions

7.1 Limitations

This implementation of chess has certain limitations resulting from some design choices and practical constraints during the development. Mainly, the constraints included a strict deadline, which necessitated prioritizing developing the core gameplay mechanics over advanced features, as well as the complexity of certain chess rules that would have required a disproportionate implementation effort relatively to their educational value. Some of the main limitations are:

- **Pawn Promotion:** currently, the pawn is automatically promoted to a queen and there is no support for underpromotion (to a knight, rook or bishop). This feature would require the implementation of additional user interface elements and state management.
- **Repetition Draw:** in Chess, a repetition can happen if the same position and conditions are repeated three times. Although sounds theoretically easy, it is not as simple to implement as it would require to keep track of pasts positions and compare if three positions repeated.

- **Clock/Time System:** an interested feature would be adding time control for each player. That would require a deep use of time-related libraries, which are not famous to be easy to use.
- **File Path Problems:** currently, there are some limitations of the PGN generator due to the way of how file paths are being handling.

7.2 Future Work

Due to the modular nature of the project, it has a strong potential to be expanded further with new functionalities and adding the previously mentioned missing features. Given the additional time, some of the plans are:

1. **Complete Core Rule Implementations:** as described in 7.1, first it is important to have a complete game, so finishing all essential game mechanics is essential to all for implementing even more advanced features
2. **Artificial Intelligence:** one of the biggest objectives is to implement a basic AI opponent which uses searching algorithms, move generation and evaluation function, while applying advanced optimization techniques. This will require significant research, learning complex algorithms and techniques.
3. **Expand UI capabilities:** there is room to implement more user interface features similarly to what is available in Chess.com or Lichess, such as move reply and a board editor
4. **Cross-platform:** it is possible to expand the game to be acceptable in other platforms. However that would require first to solve the problems in my own machine regarding the external libraries, and then add the necessary compilation instructions into the **makefile** to compile in macOS, Windows or Linux systems.

All of this possibilities will serve as a practical platform for exploring new advanced c++ techniques, learning fundamental of Game AI where it includes learning new algorithms and data structures, and new design techniques.

8 Difficulties Encountered

Many difficulties were encountered during the execution of this program, ranging from learning new concepts, libraries to finding different bugs.

- **High DPI Screen Rendering:** One of the first difficulties encountered were making all the pieces and font render correctly in the screen of my laptop, which is a **Macbook Pro M2 14"**, as it has a **high DPI** (Dots per Inch) screen. After searching for many hours and trying, I could fix the textures and render it correctly to the screen, considering the differences between the logical and physical resolutions
- **Segmentation Faults:** many were the cases where accidentally missed to check first if it is nullptr pointer before trying to access a method. Due to this, many hours were expend finding where a nullptr de-referencing was happening

- **Cyclical Dependencies:** due to the separation of different classes in different files, it is necessary to include them in each other files. It was unknown to the actor that cyclical dependency exist, i.e., if two headers include each other, the compiler will read each other include, resulting in a loop. This was solved by only including a header if it is really necessary in the header file, otherwise, it should be included in the implementation. In case of classes, a simple forward declaration solves most of the problems
- **Infinite Function Calls:** during the implementation of the move validation algorithm, the biggest problem was that the way the methods were implemented led later to create a **infinite function call** problem, which ultimately results in segmentation fault, and well, good luck to the developer to find what caused it.
- **Implementing the GUI:** while the Dear ImGui library was not a struggle, the code itself was, as the implementation had a more of a reactive rendering instead of a constant rendering. Initially, that imposed a challenge to implement the GUI on top, which required rewriting the rendering logic to allow the game itself to be rendered constantly

9 Conclusion

The project successfully represents a comprehensive implementation of a chess program that merges object-oriented design principles with the complexities of chess. Through its modular design, with low-level state management in the Board class, gameplay organization in ChessGame, and the use of inheritance/polymorphism to model different objects with same characteristics, the system demonstrates how encapsulation, inheritance, and polymorphism can create a maintainable and extensible software. Technical innovation such as the flattened array representation for board indexing to ensure performance, while the FEN parser and PGN export provides ways to use other chess tools. The project also came with different development challenges, from ensuring a sharp and pleasant looking rendering in a high-DPI screen to avoiding cyclical dependencies, all of which offered pragmatic lessons in programming. The implementation of external libraries like SDL2 and Dear ImGui showed me how to use them and the potential of using such libraries. Although this document terminates here, the project has still areas of improvement and the need for implementing new features to be a full and complete program.

10 Glossary

- **Flattened Array:** a one dimensional representation of a two dimensional board
- **FEN:** acronym for Forsyth-Edwards Notation, standardized string encoding chess positions (e.g. `rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1`).
- **Pseudolegal Move:** a move which is valid by piece rules, but ignores checks to the king.
- **RAII:** Resource Acquisition is Initialization
- **PGN:** Portable Game Notation, which is a standard format for recording chess games.
- **Algebraic Notation:** standard system for recording chess moves (e.g `e4`, `Nf3`).
- **Fullmove Clock:** counts the played moves in the game
- **Halfmove Clock:** counts played moves since last capture/pawn advance for the 50-move rule.
- **Legal Move:** a move is valid under all chess rules (including pins, forks, etc...).
- **Castling:** special move when the king swaps places with the rook, provided the pieces have not moved before
- **En Passant:** special pawn capture move which happens when a pawn moves twice and there is a pawn adjacent to it
- **Checkmate:** termination condition when a player's king is in check with no legal moves (capturing the attacking piece, moving to a safe square or covering the check)
- **Stalemate:** a player does not have legal move but is not in check
- **High DPI:** high number of Dots per Inch
- **Move Generation:** calculating the valid moves according to piece-specific offsets

References

- [1] Wikipedia contributors, *Chess — Wikipedia, the free encyclopedia*, <https://en.wikipedia.org/w/index.php?title=Chess&oldid=1288462374>, [Online; accessed 07.05.2025], 2025.
- [2] *Sdl2 documentation*, <https://wiki.libsdl.org/SDL2/FrontPage>.
- [3] L. F. Productions, *Beginning game programming v2.0*, <https://lazyfoo.net/tutorials/SDL/index.php>, 2025.
- [4] *Dear imgui documentation*, <https://github.com/ocornut/imgui/wiki/>.
- [5] GeekForGeeks, *Class diagram — unified modeling language (uml)*, <https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/>, [Online; accessed 11.05.2025], 2025.
- [6] likeawizard, *Review of different board representations in computer chess*. <https://lichess.org/@/likeawizard/blog/review-of-different-board-representations-in-computer-chess/S9eQCAWa>, [Online; accessed 10.05.2025], 2022.
- [7] chess.com, *Forsyth-edwards notation (fen)*, <https://www.chess.com/terms/fen-chess>, [Online; accessed 11.05.2025].