Alexandria University
Faculty of Engineering
CSE212: Programming I

Name: Mohamed Saber Abaas Elsayed
ID: 24010617

Name: Fares Tahseen Mohamed Nageeb
ID: 24010493

Chess Project Report

Contents

ISLAMIC CALLIGRAPHY

1 Application Description and Features

We have developed a robust, cross-platform Chess application that represents a sophisticated fusion of a high-performance logical engine and a modern graphical interface. This project was engineered from the ground up to serve as a comprehensive simulation of international chess, providing a professional environment suitable for both competitive gameplay and tactical strategy testing. Our primary focus was on achieving 100% logical accuracy while maintaining a responsive and immersive user experience.

1.1 Key Features

1.1.1 Full FIDE Rule Compliance

Implementing the "physics" of chess required a deep dive into the official FIDE laws. We engineered a complete rule-enforcement system:

- **Special Maneuvers:** We successfully implemented **Castling**, which requires checking the state of multiple pieces and verifying that the King does not pass through squares under attack.
- **En Passant:** This specialized pawn capture was integrated by tracking the "ghost square" left behind by a double-stepping pawn.
- **Pawn Promotion:** We built a dynamic promotion system that allows players to transform pawns into Queens, Rooks, Bishops, or Knights upon reaching the eighth rank.

1.1.2 Graphical User Interface (GUI)

The visual layer was developed using the **SDL2** framework, chosen for its low-level efficiency and cross-platform reliability.

- **High-Resolution Rendering:** We utilized an optimized rendering pipeline that draws the 64-square grid and piece textures with sub-pixel precision.
- **Custom Identity:** The application is personalized with a customized window title: **"Mohamed & Fares 's chess"**.

1.1.3 Persistence Engine (FEN Standardization)

To ensure that the game state is never lost, we implemented a sophisticated persistence layer based on **Forsyth-Edwards Notation (FEN)**.

This allows the engine to represent a complete 64-square board and game metadata in a single line of text.

FEN Example and Analysis: For a standard starting position, the FEN string generated by our engine is:

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

We designed our parser to interpret this string as follows:

- **Piece Placement:** The first part (`rnbqkbnr/...`) describes the pieces from the 8th rank to the 1st rank. Uppercase letters denote White, and lowercase denote Black. Digits (like `8`) represent consecutive empty squares.
- **Active Color:** The `w` indicates that it is currently White's turn to move.
- **Castling Rights:** `KQkq` signifies that both sides retain both Kingside and Queenside castling rights.
- **En Passant Target:** The `-` indicates there is currently no legal en passant target square.
- **Game Clocks:** The final digits `0 1` represent the Halfmove clock and the Fullmove number, respectively.

1.1.4 Visual & Auditory Feedback

We believe that a professional application should provide a multi-sensory experience:

- **Dynamic Highlights:** When a piece is selected, the engine precalculates all legal moves and renders them as translucent Green overlays.
- **Auditory Environment:** Using `SDL_mixer`, we integrated a customized sound library where captures, checks, and moves have distinct acoustic signatures.

1.1.5 Advanced State Detection

The engine constantly monitors the board for terminal conditions using continuous scanning logic:

- **Checkmate and Stalemate:** Detected by analyzing the King's safety in conjunction with the availability of legal moves.
- **Draw Conditions:** We implemented logic for **Insufficient Material** and **Threefold Repetition**, ensuring the game follows the highest standards of competitive play.

2 Data Architecture and Core Definitions (board.h)

In this project, we built a chess engine from scratch using the C language and the SDL2 framework. Our primary goal was to create a robust, memory-efficient, and scalable architecture. We chose to separate the game into three distinct logical layers: the Atomic layer (the piece), the Entity layer (the player), and the Global layer (the board). The `board.h` file serves as the technical blueprint, defining the data structures and the functional API that govern the "physics" of the game.

2.1 Memory Management and Data Types

We utilized Enumerations to define the basic properties of the game. This approach ensures type safety and enhances code readability by replacing "magic numbers" with descriptive constants:

- **Color Enum:** We defined WHITE and BLACK to manage turns and piece ownership. Technically, these map to 0 and 1, which we strategically use for array indexing, allowing for $O(1)$ access to player-specific data.
- **Type Enum:** Defines the identity of each piece (PAWN through KING). This classification is the core of our move generation logic, as it allows the engine to branch into different algorithmic paths based on the piece type.

2.2 Efficient Movement Modeling

One of the most critical challenges in chess programming is how to handle a list of potential moves efficiently.

- **Move Struct:** We designed this as a simple coordinate pair (row, col).
- **MoveList Struct:** Instead of using dynamic linked lists which require frequent `malloc` and `free` calls, we implemented a static buffer of size 27.

Design Choice: We chose the number 27 because it represents the maximum theoretical moves a Queen can have. By using a fixed-size array, we guaranteed high cache locality and eliminated the risk of memory leaks during the move generation phase, making the engine significantly faster.

2.3 Structural Encapsulation: The Piece Entity

The `Piece` struct is the fundamental building block of our engine. We designed it to store not just the identity, but the complete "historical state" of each unit:

- **State Tracking:** The `has_moved` flag is essential for our implementation of Castling and the Pawn's double-step. Without this, the engine would not be able to distinguish between a Rook in its starting position and one that has moved and returned.
- **UI Integration:** The `is_attacked` and `selected` fields bridge the logic with the SDL2 renderer, allowing for real-time visual feedback like highlighting legal moves.
- **Indexing Logic:** By including `index_in_player`, we created a fast mapping system. The board knows which piece occupies a square, and the player knows exactly where that piece is in their inventory, optimizing search times.

2.4 The Player and Board Global States

The hierarchy of our data structures culminates in the `Board` struct, designed as the "Single Source of Truth."

- **The Player Struct:** To optimize the "Check Detection" algorithm, we cached the King's position (`king_row`, `king_col`) directly within the Player's data. This allows our engine to verify King safety instantly without scanning all 64 squares.
- **The Board Struct:** This is the master controller. We integrated the **Logical Grid** (`board_places`) with the **Graphical Grid** (`SDL_Rect`

`chessboard`). This dual-representation allows for seamless translation between pixel coordinates (mouse clicks) and board coordinates.

- **Advanced Rule Tracking:** We included specific fields for En Passant (tracking the "ghost" square), the Halfmove Clock (for the 50-move draw rule), and Fullmove Numbers to ensure full compliance with official FIDE rules.

2.5 Functional API and Rule Enforcement

We declared a comprehensive set of functions that act as the engine's brain:

- **Modular Move Generation:** Each piece has its own specialized function (e.g., `knight_moves`, `bishop_moves`). This modularity simplifies debugging of specific piece behaviors.
- **Legal Move Validation:** The `is_it_llegal_move` function performs a "look-ahead" simulation to ensure that a move does not leave the King in check.
- **Serialization (FEN):** We implemented `board_to_fen` and `fen_to_board` to ensure compatibility with industry-standard Forsyth-Edwards Notation.
- **Terminal State Detection:** We included logic for detecting Stalemate, Checkmate, and Insufficient Material, ensuring the game handles all ending scenarios correctly.

3 Core Logic and Game Physics (board.c)

In this module, we have implemented the complete physical and logical rules of Chess. Spanning over 800 lines of code, `board.c` handles everything from initial deployment to complex rule detection. Our design focuses on **Deterministic Logic**, ensuring that every move is validated against the FIDE laws of chess before execution.

3.1 Board Initialization & Data Mapping

The foundation of the game starts with the `init_board` function. We designed this to be a "Master Reset" for the entire system:

- **Matrix Setup:** We utilize a nested loop to clear all 64 squares, ensuring that properties like `has_moved` and `captured` are zeroed out to prevent logic errors between sessions.

- **Strategic Deployment:** Pieces are placed using a hard-coded mapping that assigns unique IDs. This creates a high-speed link between the visual board and the player's internal piece array, allowing for $O(1)$ updates during high-speed move calculations.

3.2 Movement Physics Engine

We developed individual "Move Generators" for each piece type, focusing on algorithmic efficiency:

- **Sliding Logic (Rooks, Bishops, Queens):** We implemented a **Directional Vector Algorithm**. The engine projects a ray in valid directions until it hits an obstacle. If the obstacle is an opponent, it marks a capture; if it is a teammate, it stops. This ensures the engine respects the "blocking" nature of chess pieces.
- **Leaping Logic (Knights):** Using a pre-computed offset array of the 8 possible "L-shapes," we implemented a fast-check system that validates target squares in a single pass without the need for iterative loops.
- **The Pawn Complex:** We programmed unique behavior including the initial double-step, diagonal captures, and the **En Passant** rule, which requires tracking the "ghost square" from the immediate previous move.

3.3 Virtual Simulation Layer

One of the most advanced features we implemented is the Virtual Board Simulation within `is_it_llegal_move`.

- **The Challenge:** A move might be "pseudo-legal" based on piece mechanics but "illegal" if it leaves the King in a state of check.
- **The Solution:** We used dynamic memory allocation (`malloc`) to clone the entire board state. We perform the move in this "shadow world" and execute an attack scan. Only if the King remains safe is the move confirmed as legal. This prevents "pinned" pieces from moving if they would expose the King.

3.4 King Safety and Threat Detection

The `is_square_attacked` function acts as the engine's security guard. We implemented a **Reverse Scanning Method**: To see if a square is at-

tacked, we look outwards from that square as if it were a Queen, a Knight, or a Pawn. If we find an enemy piece that could "look back" at this square, we flag it as attacked. This is essential for Castling logic, where the King cannot pass through or land on a square under fire.

3.5 Terminal State Intelligence (Win/Loss/Draw)

We programmed the engine to recognize terminal scenarios to ensure full compliance with official rules:

- **Checkmate vs. Stalemate:** By combining `get_total_possible_moves` with the King's check status, the engine distinguishes between a decisive win and a draw.
- **Insufficient Material:** We implemented a counting algorithm that detects "dead positions" where checkmate is mathematically impossible (e.g., King and Knight vs. King). We even added logic to check bishop square colors using the formula $(row + col)\%2$.
- **Threefold Repetition:** Using FEN serialization, we created a history-scanning loop that compares the current position with previous moves to trigger a draw if the position repeats three times.

3.6 Move Execution and Sensory Integration

The `move_piece` function serves as the command center. Beyond updating coordinates, it:

- **Manages Special Moves:** Automatically moves the Rook during Castling and removes the correct pawn during En Passant.
- **Triggers Multimedia:** We integrated `SDL_mixer` to trigger distinct sounds for captures, checks, and standard moves, ensuring a tactile and professional feel.

4 Visual and Interactive Hub (main.c)

In this final module, we integrated the core chess logic with the SDL2 library to create a fully interactive graphical user interface. `main.c` acts as the orchestrator of the program, managing the Game Loop, user input, asset rendering, and real-time state synchronization. Our goal was to build a seamless experience where the complex backend logic is presented through an intuitive and responsive front-end.

4.1 Graphics and Media Architecture

We utilized multiple SDL2 extensions to enhance the sensory experience and performance of the application:

- **SDL_image & Texture Mapping:** We implemented a 2D texture array
  `piece_textures[color][piece_type]` for $O(1)$ access. This ensures that during the rendering phase, the engine does not perform expensive file searches, but instead pulls pre-loaded textures directly from GPU memory.
- **SDL_ttf (Typography):** Integrated for dynamic text rendering. We use this to display real-time coordinate labels (A-H, 1-8), turn indicators, and the system command console, ensuring the interface is both informative and legible.
- **SDL_mixer (Audio Orchestration):** We mapped specific game events to distinct audio channels. This provides the player with professional auditory feedback for every action, from the subtle click of a move to the urgent alert of a King in check.

4.2 The Event-Driven Game Loop

The heart of our application is a structured `while(running)` loop. We designed this loop to handle three critical phases in every frame to maintain a smooth 60 FPS experience:

- **Phase 1: Event Handling:** We utilize `SDL_PollEvent` to intercept user interactions. We engineered a coordinate translation algorithm that calculates which board square was clicked based on the raw mouse coordinates (x, y), effectively mapping screen pixels to our 8×8 logical matrix.
- **Phase 2: State Update:** After an event, we synchronize the visual state with the backend. This includes querying `board.c` for game-ending conditions, updating the move history, and managing the active turn transitions.
- **Phase 3: Multi-Layer Rendering:** We implement a "Painter's Algorithm" approach. We render the board grid first, followed by translucent move highlights, then the chess pieces, and finally the UI overlays and text. This prevents visual artifacts and ensures a clean composition.

4.3 Interactive Features: Selection and Promotion

We implemented a two-step interaction model for making moves to ensure precision:

- **Highlighting System:** When a player selects a piece, we query `get_possible_moves` and populate a `highlighted_squares` list. These are rendered as soft-glow overlays, providing the user with immediate visual confirmation of legal destinations.
- **Pawn Promotion State-Machine:** We designed a specific modal state to handle promotion. When a pawn reaches the 8th rank, we pause standard game-loop processing and enter a "Selection Mode," waiting for specific keyboard triggers (Q, R, B, N) to finalize the piece transformation.

4.4 UI Real-Time Feedback and Material Tracking

To improve the user's tactical awareness, we dedicated screen real estate to real-time status updates:

- **Captured Material Sidebars:** We implemented logic to scan the `captured_piece` arrays and render those pieces in designated zones outside the main board, allowing players to track material advantage at a glance.
- **The Command Console:** A dedicated text area provides textual status updates, such as "White is in Check" or "Draw by Threefold Repetition," bridging the gap between graphical events and the underlying engine logic.

4.5 Memory Lifecycle Safety

We implemented a rigorous cleanup routine in the final stage of `main.c`. Every texture, font, and audio chunk is explicitly destroyed using `SDL_DestroyTexture`, `TTF_CloseFont`, and `Mix_FreeChunk`. This ensures that the application terminates gracefully without leaving a memory footprint on the host operating system.

5 Memory and Storage System (file.c)

In this module, we developed the essential bridge between the game's volatile memory and permanent storage. By centering our persistence

layer around the **FEN (Forsyth–Edwards Notation)** standard, we ensured that our chess engine is not only capable of saving its own state but is also technically compatible with international chess databases and external analysis software.

5.1 Advanced FEN Encoding (Board to String)

We engineered a sophisticated `board_to_fen` function that translates the high-level `Board` structure into a compact, standardized string. This process involves several critical sub-logics:

- **Dynamic Piece Decoding:** We implemented a `piece_decoder` that maps internal piece types and colors to their standard ASCII characters (e.g., uppercase 'K' for White King, lowercase 'q' for Black Queen).
- **Empty Square Compression:** Following FIDE standards, we implemented logic to count consecutive empty squares and represent them as integers (1-8). This significantly reduces the storage footprint of the board state.
- **State Serialization:** Beyond piece coordinates, we ensured the string captures the complete game context, including the active turn, castling rights (K/Q/k/q), En Passant target squares, and the Halfmove/Fullmove clocks.

5.2 Robust Parsing and Reconstruction (String to Board)

The `fen_to_board` function acts as our reconstruction engine. We designed it to be a "destructive-safe" re-initializer:

- **State Reset Protocols:** Before loading, the engine clears all 64 squares and resets player statistics (captured pieces, check status) to prevent data "bleeding" from previous game states.
- **Piece Encoding:** Using our `piece_encoder`, we reconstruct each `Piece` object individually. We assign each unit a unique ID and restore its `index_in_player` to maintain the integrity of our piece-tracking arrays.
- **Coordinate Synchronization:** We integrated `SDL_Rect` initialization directly into the loading process, ensuring the graphical board is ready for immediate rendering upon the completion of the file read.

5.3 Security and Validation Logic

To prevent system crashes and ensure stability when handling user-modified files, we implemented a multi-layered validation system:

- **The 6-Part Syntax Check:** The `is_valid_fen` function performs a rigorous analysis of the input string. It validates the number of ranks, ensures each rank totals exactly 8 squares, and verifies that the active color and move clocks are numerically sound.
- **File I/O Safety:** We built an `is_file_found` utility to prevent the program from attempting to read non-existent data. Additionally, our `save_file` function features an automated naming algorithm (e.g., `Game(1)`, `Game(2)`) to prevent accidental overwriting of user progress.

5.4 Persistence Architecture and Memory Efficiency

We established a dedicated storage hierarchy to manage user data professionally:

- **Directory Isolation:** All saved states are directed to a specific `Saved_Games` folder, isolating user data from the core source code and binary files.
- **High-Speed Conversion:** We utilized `sscanf` and `sprintf` for high-speed string-to-integer conversions. This ensures that saving or loading a game happens in near-instantaneous time, $O(n)$ where n is the length of the FEN string, providing a smooth user experience even on lower-end hardware.

6 Automated Build System (Makefile)

In this module, we implemented a structured build system using **GNU Make**. The primary objective was to automate the compilation process of our multi-file C project, ensuring that all dependencies, compiler flags, and external libraries are linked correctly. By using a `Makefile`, we eliminated the need for manual command-line compilation, providing a consistent, reproducible, and efficient development workflow.

6.1 Compiler Configuration and Optimization

We defined several variables within the build script to maintain flexibility and granular control over the compilation process:

- **The Compiler (CC):** We designated `gcc` as our primary compiler, ensuring compatibility with standard C development tools and POSIX environments.
- **Compiler Flags (CFLAGS):**
  - **-Wall:** We enabled all compiler warnings to maintain high code quality. This was critical during the development of `board.c` for catching potential logical errors and type mismatches.
  - **-g:** We included debugging symbols to allow tools like GDB or Valgrind to trace memory issues and perform runtime analysis.
  - **sdl2-config –cflags:** We integrated this command to automatically locate the necessary SDL2 header files on any host system, ensuring the build is not hardcoded to a specific path.

6.2 Dynamic Library Linking and Portability

Because our chess engine relies on multiple multimedia extensions, we carefully managed the linker flags (`LDFLAGS`) to bridge the engine with external shared libraries:

- **SDL2_image:** Required for rendering the PNG textures of the chess pieces.
- **SDL2_ttf:** Essential for the TrueType font engine used in our UI and coordinates.
- **SDL2_mixer:** Powering the high-fidelity sound effects for move alerts.
- **Linker Automation:** By using `sdl2-config --libs`, we ensure that the core SDL2 library is linked with the correct system-specific paths, making our project portable across different Linux and Unix-like environments.

6.3 Dependency Mapping and Project Structure

We organized the build rules to reflect the modular nature of our project, separating the source files (`SRC`) from the final executable (`OUT`):

- **Source Tracking:** We explicitly listed `main.c`, `file.c`, and `board.c`. This ensures that any modification to the core logic or the file-handling system is captured, triggering a re-compilation of only the necessary components.

- **Build Target:** We established the `all` rule as the default target. This rule orchestrates the assembly of our object files and libraries into a single, high-performance binary optimized for the host architecture.

6.4 Environment Maintenance and Reliability

To keep the development environment clean and prevent binary conflicts, we implemented a `clean` utility:

- **Cleanup Rule:** This command removes the compiled executable (`rm -f $(OUT)`).
- **Consistency Check:** We utilized this rule to ensure that every major architectural update is tested from a "fresh" build. This prevents old binary artifacts from interfering with new logic, particularly after modifying the shared `board.h` header file.

7 Multimedia Assets (Assets)

In this final module, we managed the integration of the external media resources that transform the chess engine from a purely logical back-end into a complete, user-friendly application. We organized our assets into three primary categories: typography, graphical textures, and auditory feedback. By carefully selecting and linking these files, we ensured that every move and state change in the engine provides immediate visual and acoustic confirmation to the player, enhancing both gameplay clarity and immersion.

7.1 Typography and UI Clarity

For the graphical interface's text elements, we integrated the `arial.ttf` TrueType font.

- **Dynamic Rendering:** We utilized this font to render the board's coordinate system (A-H and 1-8) and the real-time command console.
- **Scaling and Legibility:** We configured the font settings within the code to maintain high legibility across different UI components, such as the turn indicator and the action buttons (Undo, Save, etc.), ensuring the interface remains professional and readable regardless of window scaling.

7.2 Graphical Asset Mapping (Piece Textures)

We implemented a comprehensive set of high-fidelity `.png` images for the chessboard's visual representation.

- **Complete Set:** We included individual textures for all 12 unique chess pieces (Pawn, Rook, Knight, Bishop, Queen, and King for both White and Black colors).
- **Transparent Layering:** The images were formatted with per-pixel alpha transparency. This allows them to sit perfectly atop the grid squares generated by the SDL2 renderer, preventing any "clipping" effect and ensuring that piece movement looks fluid as it transitions across different colored squares.
- **O(1) Access Pattern:** By loading these into a structured 2D array, we ensured that the renderer can retrieve the correct texture in constant time during the drawing loop.

7.3 Auditory Feedback System (SFX)

To enhance the tactical feel of the game, we integrated a library of `.wav` audio files. We mapped these sounds to specific logical triggers within the engine to provide a multi-sensory experience:

- **Movement Cues:** We distinguished between standard moves (`move-self.wav`) and critical threats (`move-check.wav`). This helps the player recognize a check through sound even before visually analyzing the board.
- **Special Rules:** We added unique audio signatures for complex moves like `castle.wav` and `promote.wav`, giving the player clear feedback when these rare events occur.
- **Game Flow:** We implemented `game-start.wav` and `game-end.wav` to define the boundaries of a session, along with an `illegal.wav` sound to immediately notify the player of a rule violation.
- **Combat Feedback:** The `capture.wav` sound provides a satisfying acoustic confirmation when a piece is removed from the board, reinforcing the impact of the tactical exchange.

7.4 Resource Lifecycle and Memory Optimization

We focused on the efficient management of these assets to prevent memory leaks or unnecessary system strain:

- **Centralized Loading:** All assets are loaded into RAM during the initial boot sequence using `SDL_image` and `SDL_mixer`. This avoids stuttering that would occur if files were read from the disk during active gameplay.
- **Safe Disposal:** We implemented a rigorous shutdown sequence. Every loaded font, texture, and sound chunk is properly freed from memory when the application is closed using `SDL_DestroyTexture` and `Mix_FreeChunk`, maintaining the integrity of the host operating system's resources.

8 User Manual and Implementation Assumptions

To ensure a seamless interaction between the user and the engine, we established a clear set of operational guidelines and technical assumptions. This section provides a visual and descriptive guide to the application's interface.

8.1 Implementation and Environmental Assumptions

We engineered the engine with specific architectural assumptions to maintain performance:

- **Resource Localization:** The `assets/` directory must reside in the same root directory as the binary.
- **File System Permissions:** The application requires write-access to create the `Saved_Games/` directory.

8.2 Operating Instructions and Visual Guide

We designed the user interface to be intuitive, focusing on high-fidelity visual feedback.

8.2.1 Main Game Window

Upon launching the application, the user is presented with the primary chessboard interface. The window is titled **"Mohamed & Fares 's chess"** and features a high-resolution 8x8 grid.

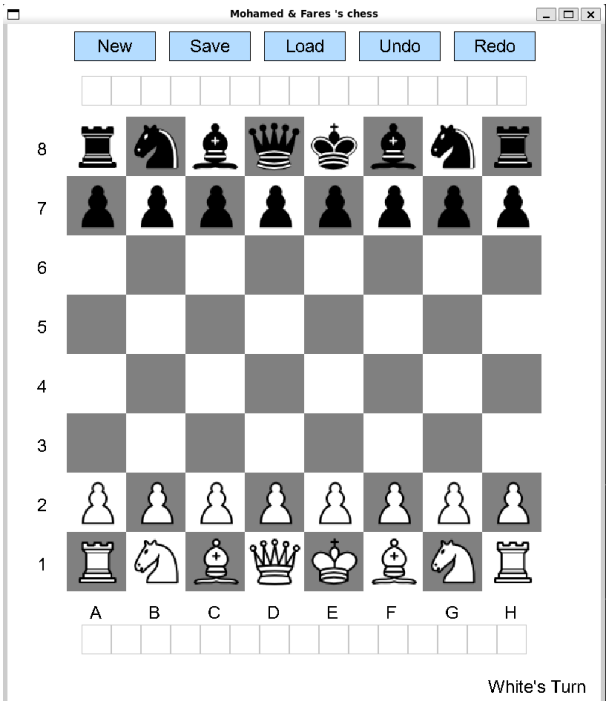

Figure 1: Initial game state and Graphical User Interface.

8.2.2 Movement and Highlighting

Left-click a piece to activate the move generation logic. As shown in Figure 2, legal destination squares are marked with translucent indicators (Green Color).

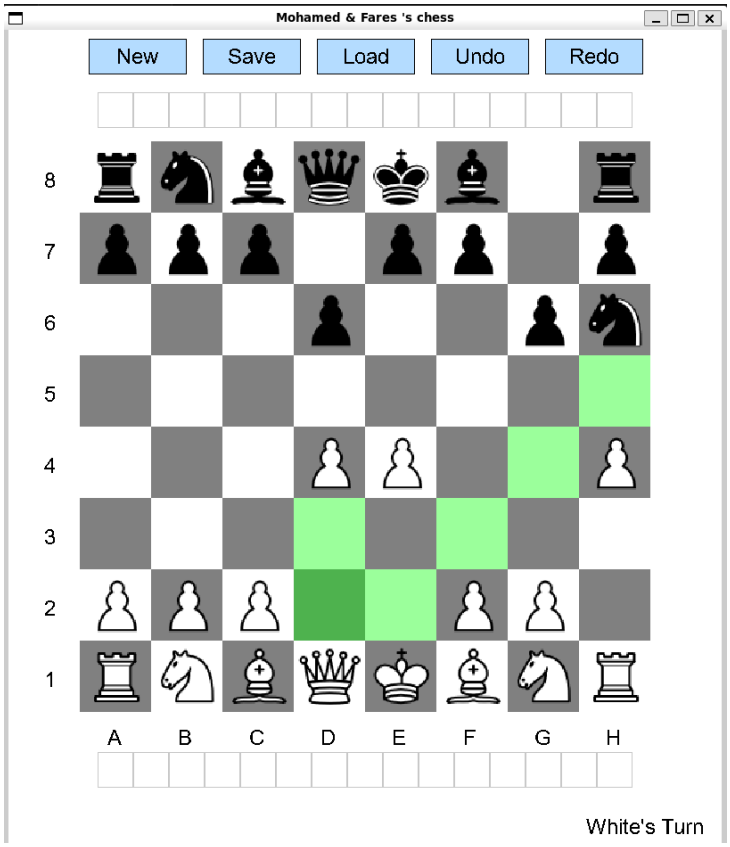

Figure 2: Visual feedback system showing legal moves for the selected piece.

8.2.3 Material Tracking (Captured Pieces)

We implemented a real-time tracking system for captured pieces. Whenever a piece is captured, its texture is removed from the board and dynamically rendered in the empty designated squares located in the sidebar (Figure 3). This allows players to visually track the material advantage.

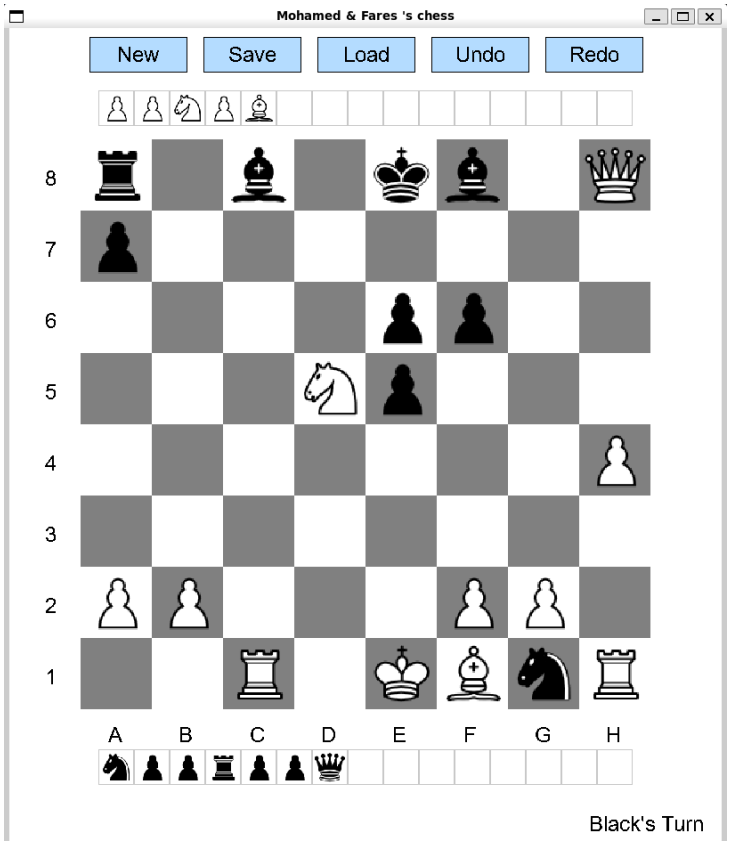

Figure 3: The Sidebar display showing captured pieces in dedicated empty slots.

8.2.4 Pawn Promotion and Special States

When a pawn reaches the 8th rank, the game enters a promotion state. The user must use the keyboard to select the new piece type as illustrated in Figure 4.

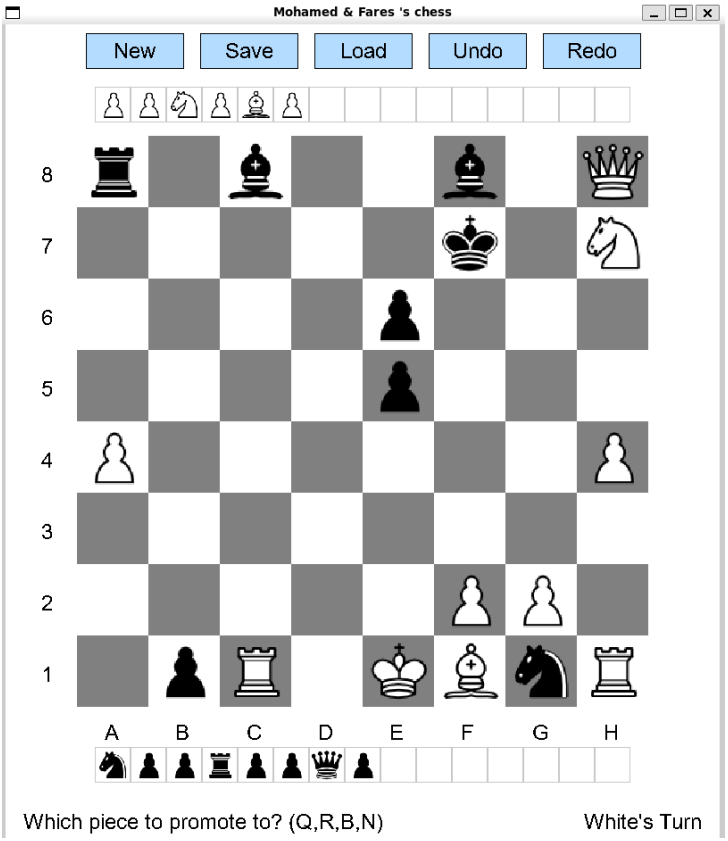

Figure 4: Pawn promotion interface requiring keyboard input (Q, R, B, N).

8.2.5 Game Persistence (Saving and Loading)

By clicking the **"Save"** button, the current state is serialized into a FEN string. The terminal provides confirmation as seen in Figure 5.

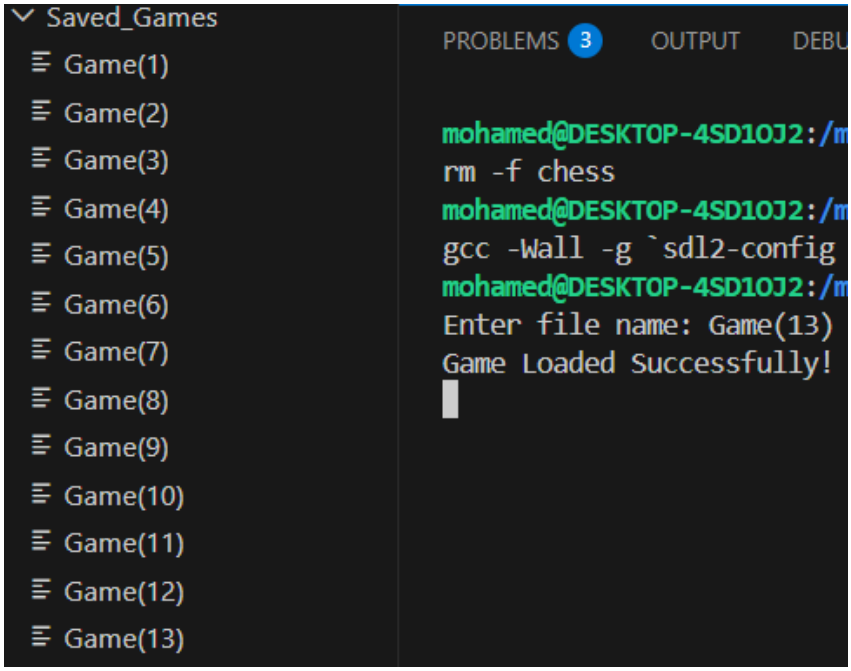

Figure 5: Terminal output confirming data serialization and file persistence.

9 References

In the development of this chess engine, we relied on a variety of technical documentations, international standards, and specialized chess programming archives. Below is the comprehensive list of resources used:

- **TalkChess Forum:**
  The primary hub for computer chess engine discussions: www.talkchess.com/forum/index.php
- **Chess Programming Wiki:**
  The most comprehensive knowledge base for chess logic: www.chessprogramming.org
- **Bruce Moreland's Programming Topics:**
  Fundamental algorithms for chess engines: web.archive.org/web/20070607231311/http://www.brucemo.com/compchess/programming/index.htm
- **Dr. Robert Hyatt's Publications:**
  Research and papers on engine architecture: web.archive.org/web/20110629052846/http://www.cis.uab.edu/hyatt/pubs.html
- **Crafty Engine Home Page (Dr. Hyatt):**
  Historical archives of the Crafty engine: web.archive.org/web/20110118221705/

http://www.cis.uab.edu/hyatt/

- **CCRL (Computer Chess Rating Lists):**
  The definitive engine performance and testing list: www.computerchess.org.uk/ccrl/4040/rating_list_all.html
- **Stockfish Chess Engine:**
  The highest-rated open-source engine documentation: www.stockfishchess.org
- **TSCP (Tom's Simple Chess Program):**
  A classic reference for entry-level C engine logic: www.tckerrigan.com/Chess/TSCP
- **Arena Chess GUI:**
  Professional engine testing and interface standards: www.playwitharena.de
- **UCI (Universal Chess Interface):**
  The official specification for engine-GUI communication: www.shredderchess.com/chess-features/uci-universal-chess-interface.html
- **SDL2 Library Ecosystem:**
  Graphics, Audio, and Typography documentation: www.libsdl.org
- **FIDE Laws of Chess:**
  Official International Chess Rules: handbook.fide.com