Alexandria University
Faculty of Engineering
Computer and Systems Dept.

DR: Marwan Torky
TA: Karim Alaa
CS221: Programming I

CS221: Computer Programming I

Course Project: Chess

Team Members

Name	ID
Youssef Mohamed Mahmoud Abdelhamid	24010873
Adham Bahaa Eldin Mohamed Mohamed Anwer Hamada	24010092

Contents

1 Application Description

Our project is a **sample Chess application** for **two players**, implemented entirely in the **C programming language**.

The **uppercase letters represent White pieces**, while **lowercase letters represent Black pieces**.

We designed the board to be **easy to read** by adding visual separators, making it more **pleasant for the eyes** and **user-friendly**.

All features are implemented using **C code only**, without any external libraries for GUI or graphics.

These features include:

- **Save and Load** game functionality
- **Undo and Redo** moves
- **Exit** the game safely

Advanced chess rules implemented:

- **En Passant** captures
- **Stalemate** and **Checkmate** detection
- **Draw by insufficient material**
- **Castling**

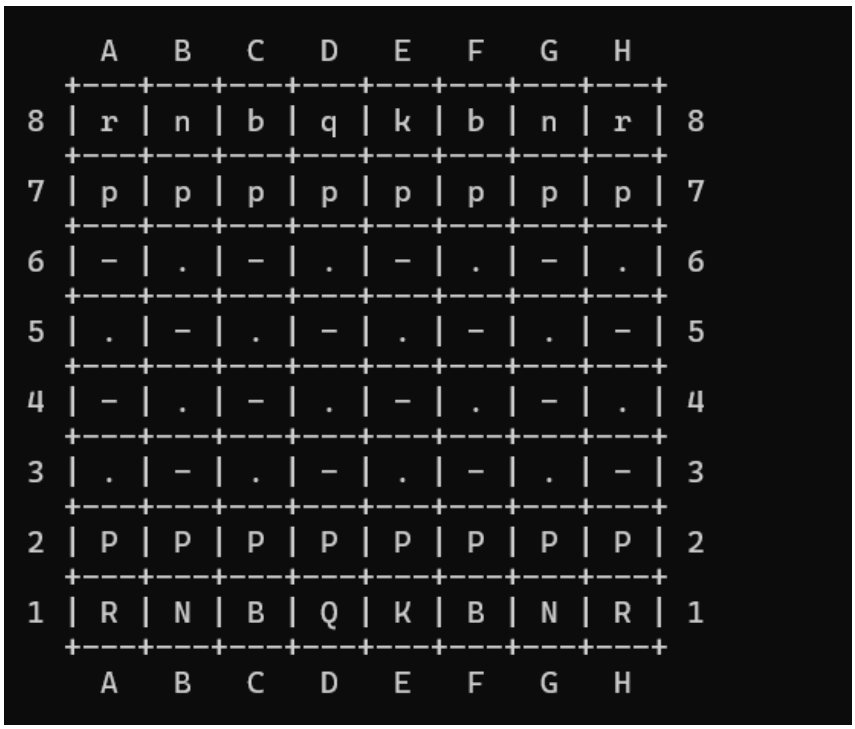
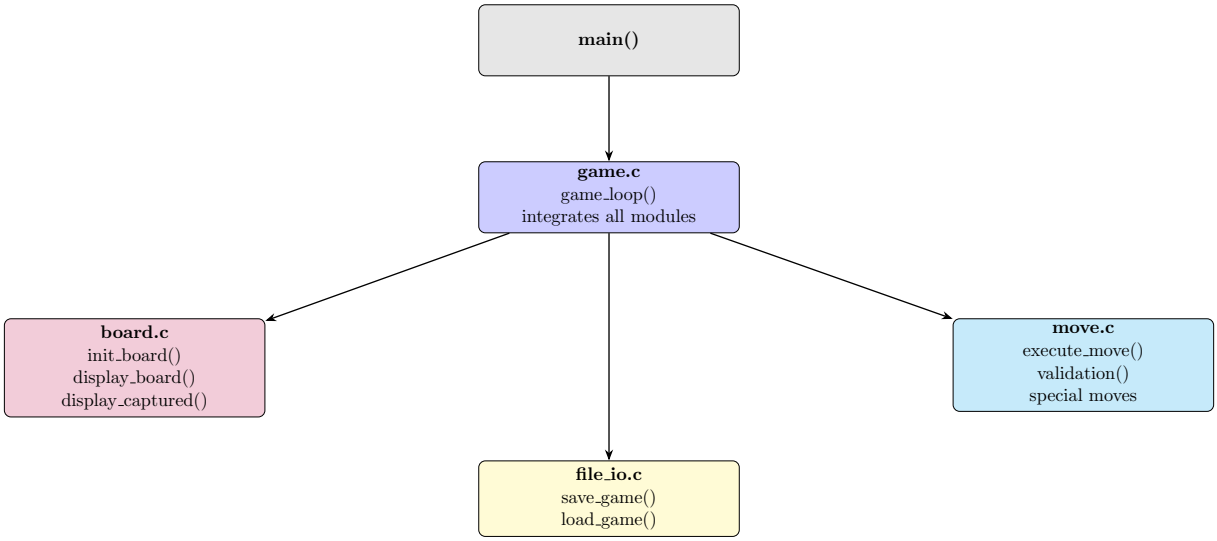


Figure 1: The Board

2 Overview of the Design

The Chess project is organized into **four essential files** to maintain modularity, clarity, and functionality:

1. **board.c / board.h:** Responsible for the board structure and piece positions:
   - `init_board()` – initializes all pieces in their correct starting positions.
   - `display_board()` – prints the board in a clear and readable format.
   - `display_captured()` – shows captured pieces for both White and Black.
2. **file_io.c / file_io.h:** Handles all input/output operations:
   - `save_game()` – saves the current game state to a file.
   - `load_game()` – loads a previously saved game.
3. **move.c / move.h:** Implements all movement logic and special rules:
   - Validation of legal moves for all pieces.
   - Special moves: **En Passant**, **Castling**.
   - Checkmate and stalemate detection.
   - Undo and redo functionality.
   - `execute_move()` – updates the board according to the move.
4. **game.c / game.h:** Central file to manage gameplay:
   - Integrates all modules into a `game_loop()` for continuous play.
   - Calls functions from `board`, `move`, and `file_io` to implement complete game logic.



3 Assumptions

To ensure smooth gameplay and simplify user interaction, we made the following assumptions regarding user input and game behavior:

- **Move Input Format:** All moves are assumed to be entered in the format: `from_column from_row to_column to_row`, similar to standard chess notation (e.g., `e2e4`). Both uppercase and lowercase letters are accepted (e.g., `E2E4` or `e2e4`).
- **Board Reference:** To assist the user, each row is labeled from 1 to 8 and each column from A to H. This makes it easier to determine positions while entering moves.
- **Excess Input:** If the user enters more than 4 characters, only the first 4 characters are considered for the move.
- **Invalid Moves:** Any move that is not valid according to chess rules will result in a message: `Invalid Move`, and the user will be prompted to enter a new move.
- **Castling Moves:** For castling, the user should enter:
  - White King-side: `e1g1`, White Queen-side: `e1c1`
  - Black King-side: `e8g8`, Black Queen-side: `e8c8` Both uppercase and lowercase letters are accepted (e.g., `E1G1` or `e1g1`).
- **Pawn Promotion:** After a pawn reaches the last row, the user will be prompted to choose a piece to replace the pawn (Queen, Rook, Bishop, or Knight). Input can be entered in uppercase or lowercase letters (e.g., `q` or `Q`).
- **Special Commands:**
  - To save the game, type: `save` (also `SAVE` or `Save`)
  - To load a game, type: `load` (also `LOAD` or `Load`)
  - To undo a move, type: `undo` (also `UNDO` or `Undo`)
  - To redo a move, type: `redo` (only available if at least one undo has been performed; also `REDO` or `Redo`)
  - To exit the game, type: `exit` (also `EXIT` or `Exit`)

4 Data Structures Description

This section describes the data structures and constants used in the chess game implementation.

4.1 Piece Representation

4.1.1 Piece Type Constants

The following constants define the type of a chess piece:

- `KING (0)`
- `QUEEN (1)`
- `ROOK (2)`
- `BISHOP (3)`
- `KNIGHT (4)`
- `PAWN (5)`
- `EMPTY (6)` — indicates an empty square

4.1.2 Piece Color Constants

The color of each piece is defined using the following constants:

- `WHITE (1)`
- `BLACK (2)`

4.1.3 Piece Structure

```
typedef struct{
    int type;
    int color;
} Piece;
```

Description The `Piece` structure represents a single chess piece.

- `type`: Specifies the type of the piece (King, Queen, etc.)
- `color`: Specifies the color of the piece (White or Black)

If `type = EMPTY`, the square contains no piece and the color field is ignored.
—

4.2 Board Structure

```
typedef struct
{
    Piece square[bsize][bsize];
    Piece whitecaptured[16];
    Piece blackcaptured[16];
    int whitecapturedcount;
    int blackcapturedcount;
    int whitekingmoved;
    int blackkingmoved;
    int whiterook_amoved;
    int whiterook_hmoved;
    int blackrook_amoved;
    int blackrook_hmoved;
    int enpassen_row;
    int enpassen_col;
    int wkingsq[2];
    int bkingsq[2];
} Board;
```

Description The `Board` structure stores the complete state of the chessboard.

- `square[bsize][bsize]`: 2D array representing the chessboard squares
- `whitecaptured[16], blackcaptured[16]`: Lists of captured pieces
- `whitecapturedcount, blackcapturedcount`: Number of captured pieces

Castling State

- `whitekingmoved, blackkingmoved`: Track king movement
- `whiterook_amoved, whiterook_hmoved`: Track white rooks movement
- `blackrook_amoved, blackrook_hmoved`: Track black rooks movement

These flags are used to validate castling rights.

En Passant State

- `enpassen_row, enpassen_col`: Target square for en passant capture

King Positions

- `wkingsq[2]`: White king position (row, column)
- `bkingsq[2]`: Black king position (row, column)

—

4.3 Game Structure

```
typedef struct {
    Board board;
    int current_player;
    int state;
    int flag;
} Game;
```

Description The `Game` structure represents the overall game state.

- `board`: Current chessboard configuration
- `current_player`: Player to move (WHITE or BLACK)
- `state`: Game result status
- `flag`: Auxiliary control flag (e.g., special conditions)

Game States

- `ONGOING (0)`
- `WHITE_WIN (1)`
- `BLACK_WIN (2)`
- `DRAW (3)`

—

4.4 MoveRecord Structure

```
typedef struct {
    int from_row, from_col;
    int to_row, to_col;
    int enpassant_row, enpassant_col;
    int is_en_passant;
    int white_king_moved, black_king_moved;
    int white_rook_a_moved, white_rook_h_moved;
    int black_rook_a_moved, black_rook_h_moved;
    Piece moved_piece;
    Piece captured_piece;
    Piece promoted_piece;
} MoveRecord;
```

Description The `MoveRecord` structure stores all information required to undo or redo a move.

- Source and destination squares of the move
- En passant information
- Previous castling rights
- The moved piece
- Any captured piece
- Promoted piece (if pawn promotion occurred)

—

4.5 Move History Structure

```
typedef struct {
    MoveRecord history[100];
    int move_count;
    int undo_count;
    int from_row;
    int from_col;
    int to_row;
    int to_col;
    Piece moved_piece;
    Piece captured_piece;
} Move;
```

Description The `Move` structure manages move history and undo/redo functionality.

- `history[100]`: Array storing past moves
- `move_count`: Number of executed moves
- `undo_count`: Number of undone moves
- Temporary fields storing the current move information

This structure enables reliable undo and redo operations while preserving full game state integrity.

5 Important Functions

This section explains the most important functions used in the project, focusing on move validation, special rules, and game-ending conditions.

5.1 board.c — Board Initialization and State Management

The file `board.c` is responsible for initializing the chessboard, maintaining board consistency, and providing helper functions that manage board-related state updates throughout the game.

5.1.1 Board Initialization

The function `init_board()` sets up the initial chess position according to standard chess rules.

This function performs the following operations:

- Initializes all board squares to empty.
- Places all white and black pieces in their standard starting positions.
- Initializes captured piece arrays and their counters.
- Sets initial castling flags for both kings and rooks.
- Initializes en passant target values to an invalid state.
- Stores the initial positions of both kings.

This function is called once at the beginning of the game and guarantees a valid starting configuration.

—

5.1.2 Board Display and Debugging

The function `print_board()` is used to display the current board state in a human-readable format.

It:

- Prints rank and file labels for easier user interaction.
- Displays uppercase characters for white pieces and lowercase characters for black pieces.
- Uses a placeholder symbol to represent empty squares.

This function is primarily used for debugging and user interaction in the console-based interface.

—

5.1.3 Board Copying and Simulation

Several functions in `board.c` rely on board copying to safely simulate moves.

A helper function such as `copy_board()` creates a deep copy of the board structure, allowing:

- Move simulation without altering the actual game state.
- Safe validation of moves involving checks or special rules.

This approach is heavily used in check, checkmate, and stalemate detection logic.

—

5.1.4 Captured Piece Management

Captured pieces are managed directly through board-level functions.

Whenever a capture occurs:

- The captured piece is added to the appropriate captured array.
- The corresponding captured count is incremented.

These arrays are also updated during undo and redo operations to ensure consistency across game state transitions.

—

5.2 move.c — Move Validation and Game Logic

The file `move.c` is responsible for validating moves, handling special chess rules, and determining game states such as check, checkmate, stalemate, and draw.

5.2.1 Basic Piece Movement Validation

Each chess piece has its own movement logic implemented in a separate function:

- `rook()` verifies that either the row or the column remains unchanged.
- `bishop()` checks that the absolute difference between rows equals the absolute difference between columns.
- `queen()` combines the logic of both rook and bishop.
- `knight()` checks that the move is in an L-shape, where the difference between rows is 2 and columns is 1, or vice versa.
- `king()` ensures that the king moves only one square in any direction, and that the distance between the two kings is at least one square.

Pawn movement is more complex and is handled separately due to special cases such as double-step movement, captures, en passant, and promotion.

5.2.2 Path Checking Logic

For pieces that move through multiple squares (rook, bishop, queen), we implemented two helper functions:

- `path_check()` verifies that all squares between the source and destination are empty.
- `dest_check()` ensures that the destination square does not contain a piece of the same color.

To simplify path traversal, we use a helper function called `sign()`, which determines the movement direction:

- Positive value for upward or right movement.
- Negative value for downward or left movement.

Using this direction, the function loops through all intermediate squares. If any square is occupied, the function returns false; otherwise, it returns true.

5.2.3 Main Validation Function

The main function `validation()` controls the entire move validation process:

1. Check that the input coordinates are within the board range.
2. Ensure the selected square contains a piece belonging to the current player.
3. Check that the destination square is valid using `dest_check()`.
4. Identify the selected piece type using a `switch` statement.
5. For each piece:
    - Check if the path is clear (if required).
    - Call the corresponding piece validation function.

If all conditions are satisfied, the move is considered valid.

5.2.4 Pawn Movement and Promotion

Pawn logic is handled carefully due to its special rules:

- A pawn may move one square forward.
- On its first move, it may move two squares forward if the path is clear.
- Diagonal captures are validated separately.

Pawn Promotion: After a pawn reaches the last rank, the function `pawn_promotion()` checks whether promotion is required. If true, `change_pawn()` asks the user to choose a replacement piece (Queen, Rook, Bishop, or Knight), accepting both uppercase and lowercase input. The promoted piece is stored in the move history for undo functionality.

5.2.5 En Passant

The `enpasswn()` function handles the en passant rule:

- It checks whether the destination square matches the stored en passant target.
- If valid, the captured pawn is removed from the board and stored in the move history.
- If a pawn moves two squares forward, the en passant target square is updated.
- Otherwise, en passant values are reset.

5.2.6 Castling Logic

Castling is handled using a dedicated path-checking function `castling_path_clear()`.

The castling function follows these steps:

1. Ensure the moved piece is a king.
2. Verify that neither the king nor the rook has moved before.
3. Confirm that the path between the king and rook is clear.
4. Execute the move temporarily on a copied board.
5. Check that the king is not in check before, during, or after castling.

If all conditions are satisfied, the castling move is executed and recorded.

5.2.7 King Check Detection

The function `is_king_checked()` locates both kings on the board and checks whether the current player's king is under attack.

This is done by iterating over all opponent pieces and calling `can_attack()` to determine if any piece can legally attack the king's square.

5.2.8 Checkmate Detection

Checkmate is determined using two functions:

- `no_check_blocked_orCaptured()` checks whether any legal move can block the check or capture the attacking piece.
- `checkmate()` confirms checkmate if the king is in check and no valid escape move exists.

Temporary board copies are used to simulate moves safely without affecting the actual game state.

5.2.9 Stalemate and Draw Conditions

Stalemate: The function `stalmate()` checks whether the current player has no legal moves while the king is not in check.

Draw by Insufficient Material: The `draw()` function detects draw situations where only kings or a king with a single bishop or knight remain on the board.

If no piece capable of forcing checkmate exists, the game is declared a draw.

5.2.10 Move Execution

Once a move has been validated, the function `execute_move()` is responsible for applying the move to the board.

This function performs the following tasks:

- Moves the selected piece from the source square to the destination square.
- Handles captures by removing the opponent piece from the board.
- Updates the king's position if the moved piece is a king.
- Updates castling-related flags when a king or rook is moved.
- Handles special moves such as castling and en passant by performing the necessary additional square updates.

The function updates only the board state and does not perform validation, as all legality checks are guaranteed to have been completed beforehand.

—

5.2.11 Move Recording

To support undo and redo functionality, every executed move is stored using the function `record_move()`.

This function creates a `MoveRecord` entry that captures the complete state change caused by the move, including:

- Source and destination coordinates.
- The moved piece.
- Any captured piece.
- En passant information.
- Previous castling rights.
- Promotion details if a pawn was promoted.

By storing both positional and state-related information, the move history ensures that the game can be accurately restored to any previous state.

—

5.2.12 Move Reversal Logic

The function `reverse_move()` is used internally to undo a previously executed move by restoring the board to its former state.

Its responsibilities include:

- Returning the moved piece to its original square.
- Restoring any captured piece to the board.
- Reverting en passant captures when applicable.
- Restoring castling flags and king positions to their previous values.
- Reverting pawn promotions by replacing the promoted piece with a pawn.

This function operates solely on stored move history data and does not require additional validation.

—

5.2.13 Undo Functionality

The `undo_move()` function allows the user to revert the most recent move.

The undo process follows these steps:

1. Ensure that at least one move exists in the move history.
2. Retrieve the most recent `MoveRecord`.
3. Call `reverse_move()` to restore the board state.
4. Decrement the move counter and update the current player.

Undo operations also update captured-piece lists and restore all game state flags, ensuring full consistency after reversal.

—

5.2.14 Redo Functionality

The `redo_move()` function reapplies a previously undone move.

This function operates by:

1. Verifying that a redo operation is possible.
2. Retrieving the next move from the move history.
3. Re-executing the move using `execute_move()`.
4. Updating captured pieces, castling rights, and king positions accordingly.

Redo functionality allows seamless navigation through the game history without recomputing move legality.

—

5.2.15 Move History Management

The move history system ensures correct synchronization between undo and redo operations.

- When a new move is executed after an undo, all redoable moves are discarded.
- The move counter tracks the current position in the history array.
- Each move is applied or reversed using the stored `MoveRecord`, guaranteeing deterministic behavior.

This design allows reliable traversal through the game timeline while preserving all special rule states.

—

5.2.16 Integration with Game State

All move-related functions interact closely with the overall game state.

After each move execution, undo, or redo operation:

- The current player is toggled.
- The board is checked for check, checkmate, stalemate, or draw conditions.
- The game state is updated accordingly.

This ensures that the game remains consistent and responsive to all player actions.

5.3 game.c — Game Control and State Management

The file `game.c` is responsible for controlling the overall game flow, managing turns, handling user commands, and coordinating between move validation and board updates.

5.3.1 game_init()

The function `game_init()` initializes the game state at the beginning of the program. It performs the following tasks:

- Sets the current player to White.
- Initializes move counters.
- Resets all game flags such as check, checkmate, and draw.
- Prepares the game for user input.

This function ensures that the game always starts from a clean and well-defined state.

5.3.2 game_loop()

The `game_loop()` function represents the main execution loop of the game. It continues running until the game ends due to checkmate, draw, or user exit.

Within each iteration, the function:

1. Displays the current board.
2. Prompts the current player for input.
3. Reads and processes user commands.
4. Calls move validation and execution functions.
5. Updates the game state and switches turns when needed.

This function acts as the central controller of the entire game.

5.3.3 handle_input()

The function `handle_input()` processes the user's input string. It supports both chess moves and special commands.

The function:

- Converts input to a consistent case (uppercase or lowercase).
- Extracts the first four characters to interpret a move.
- Detects special commands such as:
  - `save`
  - `load`
  - `undo`
  - `redo`
  - `exit`
- Redirects the input to the appropriate function based on its type.

If the input does not match any valid command or move format, an error message is displayed.

5.3.4 execute_move()

The `execute_move()` function applies a validated move to the board.

Its responsibilities include:

- Updating the board array.
- Recording the move in the move history stack.
- Handling special cases such as:
  - Castling
  - En passant
  - Pawn promotion

This function ensures that all board updates are consistent and reversible.

5.3.5 switch_turn()

The function `switch_turn()` changes the current player after a successful move.

- If the current player is White, it switches to Black.
- If the current player is Black, it switches to White.

This function guarantees correct turn alternation throughout the game.

5.3.6 check_game_state()

The function `check_game_state()` evaluates the game status after each move.
It checks for:

- Check
- Checkmate
- Stalemate
- Draw by insufficient material

Depending on the result, the function may end the game or allow play to continue.

5.3.7 undo_move()

The function `undo_move()` restores the previous game state by reverting the last move.

- Pops the last move from the undo stack.
- Restores captured pieces and special move states.
- Updates the board and current player.

Undo is only allowed if at least one move has already been made.

5.3.8 redo_move()

The function `redo_move()` reapplies a previously undone move.

- Pops a move from the redo stack.
- Reapplies the move to the board.
- Updates the game state accordingly.

Redo is only available after at least one undo operation.

5.3.9 save_game()

The function `save_game()` stores the current game state into a file.

Saved data includes:

- Board configuration.
- Current player.
- Move history.
- Special states (castling rights, en passant).

This allows the game to be resumed later without data loss.

5.3.10 load_game()

The `load_game()` function restores a previously saved game.

- Reads game data from a file.
- Restores the board and game state.
- Resets undo and redo stacks accordingly.

After loading, the game continues from the exact point it was saved.

5.3.11 exit_game()

The function `exit_game()` safely terminates the game.

- Frees dynamically allocated memory.
- Ensures no data corruption occurs.
- Exits the main game loop.

5.4 file_io.c — Saving and Loading Game State

The file `file_io.c` handles persistent storage of the game, allowing the user to save and load game states.

5.4.1 Saving the Game

The function `save_game()` writes the current game state to a file.

This includes:

- The complete board configuration.
- The current player turn.
- Castling rights and en passant information.
- Captured piece lists and counters.
- The full move history required for undo and redo operations.

All data is written in a structured format to ensure correct reconstruction when loading.

—

5.4.2 Loading the Game

The function `load_game()` restores a previously saved game.

It performs the inverse of the saving process by:

- Reading board data and reconstructing piece positions.
- Restoring captured pieces and counters.
- Restoring game flags such as castling and en passant state.
- Rebuilding the move history and counters.

After loading, the game can resume exactly from the saved position without loss of information.

—

5.4.3 File Validation and Error Handling

Basic validation checks are performed during file operations:

- Verification that the save file exists and can be opened.
- Graceful handling of corrupted or incomplete data.
- Prevention of undefined behavior when invalid files are encountered.

If an error occurs, the game safely aborts the loading process and preserves the current state.

—

5.4.4 Integration with Game Flow

The functions in `file_io.c` are integrated into the main game loop.

- Save operations can be triggered at any time during gameplay.
- Load operations reset the current game state and refresh the board display.

This allows users to pause and resume games while maintaining full consistency with the move validation and history systems.

6 Catalog

At the start of the game, a set of instructional messages is displayed to guide the user on how to interact with the Chess application. These messages explain the expected input format, available commands, and system responses.

6.1 Input Format

- The expected move input format is:

  `from_column from_row to_column to_row`

  which follows standard chess notation, such as:

  `e2e4`

- Both **uppercase and lowercase letters** are accepted for all inputs.
- For easier use, the board is labeled:
  - Rows from **1 to 8**
  - Columns from **A to H**

6.2 Special Moves and Commands

- **Pawn Promotion:** After a pawn reaches the last rank, the player is prompted to enter the piece to promote to (Queen, Rook, Bishop, or Knight).
- **Castling:** Castling is performed by entering the king's movement:
  - White king-side: `e1g1`
  - White queen-side: `e1c1`
  - Black king-side: `e8g8`
  - Black queen-side: `e8c8`
- **Undo Move:** Enter `undo` to revert the last move.
- **Redo Move:** Enter `redo` to reapply the last undone move. Redo is only available after at least one undo operation.
- **Save Game:** Enter:

  `save <filename>`
- **Load Game:** Enter:

  `load <filename>`

6.3 System Messages

During gameplay, the system provides clear feedback messages:

- If the player enters an illegal or invalid move, the message:

  `Invalid move`

  is displayed.
- If a player is in check, the system displays:

  `You are in check`

- If the player attempts a move that does not protect the king while in check, the message:

  `You must protect the king`

  is shown.

These messages ensure that the user clearly understands the game state and required actions at all times.

7 Sample Game

This section demonstrates a complete sample game using valid inputs. Each move is followed by a small snapshot of the board state.

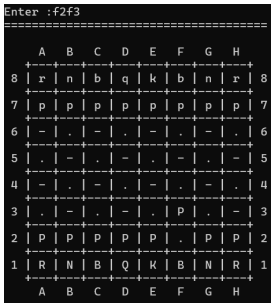


Figure 1: * **f2f3**

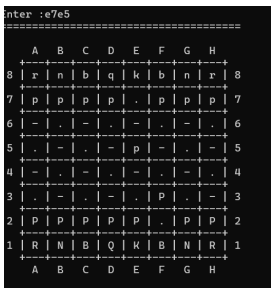


Figure 2: * **e7e5**

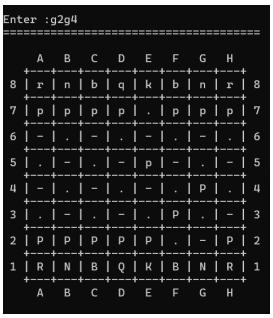


Figure 3: * **g2g4**

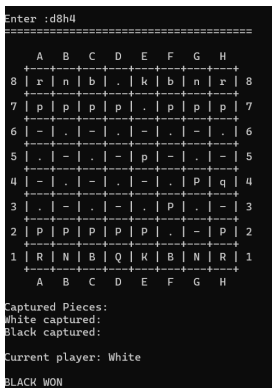


Figure 4: * **d8h4 (Checkmate)**

7.1 Result

The system detects a checkmate after the final move and terminates the game correctly.

8 Learning Outcomes

Through the development of this Chess project, the team acquired valuable technical and soft skills:

- Gained practical experience using **GitHub** for version control, collaboration, and file management.
- Learned to work with **C programming files, include statements, and modular code organization**.
- Developed problem-solving skills by analyzing and implementing complex game logic.
- Improved logical thinking and the ability to organize and plan tasks while programming.

- Enhanced teamwork and communication skills through collaborative project development.

9 References

References

[1] K. N. King, *C Programming: A Modern Approach*, 2nd Edition, W. W. Norton & Company, 2008.

[2] GitHub Tutorial, YouTube, `https://youtu.be/Q6G-J54vgKc?si=jC5Ur1saP7GgpAzQ`.

[3] Makefile Tutorial, `https://makefiletutorial.com/`.