**Console-based Chess Game**

Ezequiel N. Buck Martínez

Texas Tech University

CS 1412: Programming Principles

Dr. Juan Carlos Rojas

December 9, 2024

**Objectives**

This project aims to develop a console-based chess game that implements fundamental mechanics while showcasing the programming principles learned in class. The project aims to create an engaging two-player experience, dealing with the constraints of a console interface to design an intuitive and visually appealing user interface. The game emphasizes simplicity and clarity, allowing players to interact smoothly while adhering to essential chess rules such as piece movement and turn-based gameplay. By focusing on a robust implementation of fundamental features, the project demonstrates key concepts such as modular programming, data structure management, and dynamic memory allocation, culminating in a functional and educational application that highlights both the challenges and creative possibilities of console-based game development.

**Inputs and Outputs to the System**

The chess game system manages interaction through a combination of structured inputs and outputs, ensuring a cohesive user experience within the console environment. The primary inputs include a menu selection where players choose between starting a new game or loading a previously saved game. If the load option is selected, the saved game file becomes an input to the system, allowing it to reconstruct the board state, captured pieces, and move history. During gameplay, players provide two critical inputs: the coordinates of the piece they wish to move and the destination coordinates for the move. Additionally, the system periodically prompts players to decide whether to save the game, requiring input to confirm or decline.

The outputs of the system are designed to provide clarity and feedback to the players. The chessboard is displayed dynamically from both players' perspectives, complete with intuitive icons representing the pieces and labeled coordinates to guide the players. When saving a game, the system generates a binary file containing all necessary game data, including the board state, captured pieces, and move history, ensuring the match can be resumed seamlessly. Furthermore, the system outputs immediate feedback whenever an invalid or illegal input is detected, such as attempting to move an opponent's piece or providing incorrectly formatted coordinates, helping players correct their actions and continue the game without disruption.

**Structure Chart**

Diagrama

Descripción generada automáticamente

**Function Descriptions**

**main**

* Description: The entry point for the chess game program. It initializes the game environment by setting up the chessboard, linked lists for captures and move history, and configuring the locale for Unicode display. The function presents the user with a menu to start a new game or load a saved one. Based on the choice, it either initializes a fresh board or retrieves a saved game state. After setting up, it enters the game loop, allowing players to alternate turns until the game concludes. Upon exit, it cleans up all dynamically allocated memory and terminates the program.
* Parameters: None.
* Return Value: Returns 0 to indicate successful execution.

**init\_board**

* Description: Initializes the chessboard with the standard starting positions for all chess pieces. The function uses a temporary 8x8 array to define the initial layout, with black pieces occupying the first two rows, white pieces the last two rows, and the middle rows left empty. Once the layout is defined, it copies the temporary array into the main chessboard using memcpy, ensuring that the board is ready for gameplay.
* Parameters:
  + Piece\_t board[8][8]: The 8x8 array representing the chessboard.
* Return Value: None.

**free\_history**

* Description: Frees the memory allocated for the linked list of move history. The function iterates through the linked list, freeing each node individually to prevent memory leaks. It continues until all nodes are deallocated and the list is emptied, ensuring the system's memory remains clean.
* Parameters:
  + History\_node\_t \*head: Pointer to the head of the move history linked list.
* Return Value: None.

**free\_captures**

* Description: Frees the memory allocated for the linked list of captured pieces. It iterates through the linked list, releasing each node in turn until all are deallocated. This ensures that all dynamically allocated memory for captured pieces is properly freed, maintaining the program’s memory integrity.
* Parameters:
  + Captures\_node\_t \*head: Pointer to the head of the captures linked list.
* Return Value: None.

**print\_board\_white**

* Description: Prints the chessboard from the perspective of the white player. It displays column labels (a to h) at the top and bottom for orientation, while iterating through the rows from 8 to 1 to show the chessboard in ascending order from the white player’s point of view. Each square of the board is displayed with its corresponding piece or left empty, and row numbers are printed alongside for clarity. The board layout is enclosed in a grid format, providing an intuitive visual representation for the players.
* Parameters:
  + Piece\_t board[8][8]: The 8x8 array representing the chessboard.
* Return Value: None.

**print\_board\_black**

* Description: Prints the chessboard from the perspective of the black player. The function mirrors the white perspective, starting with column labels (h to a) at the top and bottom for orientation. Rows are displayed in descending order (8 to 1), presenting the board as seen by the black player. Each square displays its content or remains empty, with row numbers printed for reference. The grid format ensures a clear and organized representation of the game state.
* Parameters:
  + Piece\_t board[8][8]: The 8x8 array representing the chessboard.
* Return Value: None.

**print\_captures**

* Description: Prints the captured pieces for a player. It iterates through the linked list of captures, printing the icon of each captured piece separated by spaces. If there are no captured pieces, no output is displayed. The function concludes with a newline for proper formatting, offering a quick summary of the captured pieces during the game.
* Parameters:
  + Captures\_node\_t \*p\_captures\_head: Pointer to the head of the captures linked list.
* Return Value: None.

**print\_history**

* Description: Displays the move history in a tabular format. The function begins by printing a header that includes column titles for the previous and next positions of each move. It then iterates through the linked list of moves, printing each one in a formatted row. After all moves have been displayed, it prints a footer to close the table, providing players with a comprehensive view of all moves made during the game.
* Parameters:
  + History\_node\_t \*p\_history\_head: Pointer to the head of the move history linked list.
* Return Value: None.

**update\_captures**

* Description: Adds a captured piece to the linked list of captures. The function dynamically allocates memory for a new node, initializing it with the captured piece's data. If the list is empty, the new node becomes the head of the list. Otherwise, the function traverses to the end of the list and appends the new node, ensuring the capture history is maintained in sequential order.
* Parameters:
  + Captures\_node\_t \*\*pp\_captures\_head: Pointer to the pointer to the head of the captures linked list.
  + Piece\_t piece: The captured piece to be added.
* Return Value: None.

**find\_piece\_coordinates**

* Description: Locates the coordinates of a piece on the chessboard based on its position string. The function iterates through every cell of the board, comparing each piece's position string with the given input. If a match is found, it stores the corresponding row and column indices in the provided pointers and returns success. If no match is found after searching the entire board, it returns failure.
* Parameters:
  + Piece\_t board[8][8]: The 8x8 array representing the chessboard.
  + char pos[3]: The position string (e.g., "e2") to locate.
  + int \*i: Pointer to store the row index of the piece.
  + int \*j: Pointer to store the column index of the piece.
* Return Value: Returns 1 if the position is found; otherwise, returns 0.

**update\_board**

* Description: Moves a piece from one position on the chessboard to another. The function transfers the piece data (icon, color, and type) from the source position to the target position and clears the source position by setting it to an empty state. This ensures the board reflects the current game state after a move.
* Parameters:
  + Piece\_t board[8][8]: The 8x8 array representing the chessboard.
  + int prev\_i, prev\_j: The row and column indices of the piece's current position.
  + int next\_i, next\_j: The row and column indices of the piece's destination position.
* Return Value: None.

**update\_history**

* Description: Records a new move in the move history linked list. The function dynamically allocates memory for a new history node, initializing it with the move's starting and ending positions. If the list is empty, the new node becomes the head. Otherwise, the function traverses to the end of the list and appends the new node, maintaining the chronological order of moves.
* Parameters:
  + History\_node\_t \*\*pp\_history\_head: Pointer to the pointer to the head of the move history linked list.
  + char prev\_pos[3]: The starting position of the piece (e.g., "e2").
  + char next\_pos[3]: The ending position of the piece (e.g., "e4").
* Return Value: None.

**is\_valid\_move**

* Description: Validates a proposed move based on the rules of chess for the specific piece type. The function checks if the target position is occupied by a piece of the same color and ensures that the move adheres to the movement rules for each piece (e.g., pawns moving forward, knights moving in an L-shape, etc.). For moves involving paths, such as rooks and bishops, the function verifies that no other pieces obstruct the path. If the move is valid, it returns success; otherwise, it indicates failure.
* Parameters:
  + Piece\_t board[8][8]: The 8x8 array representing the chessboard.
  + int prev\_i, prev\_j: The row and column indices of the piece's current position.
  + int next\_i, next\_j: The row and column indices of the piece's destination position.
* Return Value: Returns 1 if the move is valid; otherwise, returns 0.

**get\_move**

* Description: Handles player input for making a move. The function prompts the player for the piece they wish to move and its destination, validating both inputs for format and legality. It checks whether the piece belongs to the current player, ensures the move complies with chess rules, and updates the board, captures, and move history if the move is valid. If the move is invalid, it provides feedback and prompts the player to try again. Additionally, it checks for the capture of a king to determine if the game should end.
* Parameters:
  + Piece\_t board[8][8]: The 8x8 array representing the chessboard.
  + Captures\_node\_t \*\*pp\_capture\_color\_head: Pointer to the pointer to the head of the current player's captures linked list.
  + History\_node\_t \*\*pp\_history\_head: Pointer to the pointer to the head of the move history linked list.
  + int \*captured\_king: Pointer to an integer indicating if a king has been captured.
  + int \*moves: Pointer to the integer tracking the number of moves made.
* Return Value: None.

**game\_loop**

* Description: Manages the main game loop, alternating turns between players and processing each move. For each turn, it displays the board from the current player’s perspective, shows captured pieces, and prompts the player to make a move. It periodically asks players if they wish to save the game and ends the loop if a king is captured. Upon conclusion, the function declares the winner, displays the final board, and prints the move history.
* Parameters:
  + Piece\_t board[8][8]: The 8x8 array representing the chessboard.
  + Captures\_node\_t \*p\_captures\_white\_head: Pointer to the head of the white captures linked list.
  + Captures\_node\_t \*p\_captures\_black\_head: Pointer to the head of the black captures linked list.
  + History\_node\_t \*p\_history\_head: Pointer to the head of the move history linked list.
  + int \*moves: Pointer to the integer tracking the number of moves made.
* Return Value: None.

**save\_game**

* Description: Saves the current game state to a binary file. The function writes the number of moves, board state, captured pieces for both players, and move history to the file. It uses end markers for the captures and history lists to indicate their conclusion. Upon successful saving, it prints a confirmation message.
* Parameters:
  + Piece\_t board[8][8]: The current state of the chessboard.
  + Captures\_node\_t \*p\_captures\_white\_head: Pointer to the head of the white captures linked list.
  + Captures\_node\_t \*p\_captures\_black\_head: Pointer to the head of the black captures linked list.
  + History\_node\_t \*p\_history\_head: Pointer to the head of the move history linked list.
  + int moves: The number of moves made.
* Return Value: Returns 1 on successful save; otherwise, returns 0.

**load\_game**

* Description: Loads a previously saved game state from a binary file. The function reads the number of moves, board state, captured pieces for both players and move history. It reconstructs the linked lists for captures and move history, ensuring the game state is accurately restored. On success, it prints a confirmation message; otherwise, it reports an error.
* Parameters:
  + Piece\_t board[8][8]: The chessboard to be populated with the saved state.
  + Captures\_node\_t \*\*p\_captures\_white\_head: Pointer to the pointer to the head of the white captures linked list.
  + Captures\_node\_t \*\*p\_captures\_black\_head: Pointer to the pointer to the head of the black captures linked list.
  + History\_node\_t \*\*p\_history\_head: Pointer to the pointer to the head of the move history linked list.
  + int \*moves: Pointer to the integer tracking the number of moves made.
* Return Value: Returns 1 on successful load; otherwise, returns 0.

**Test Plan**

To test my console chess game, I performed a series of focused checks to ensure the functionality and correctness of the implementation. First, I verified that the board array correctly initialized the pieces in their standard starting positions and that the board displayed properly for both players' perspectives. I tested the main menu to confirm it activated the load\_game function when selected and that it handled input validation, rejecting invalid options and prompting the user appropriately.

During gameplay, I ensured that the piece I input to move was the one that actually moved and that it reached the destination I specified. I tested that all pieces adhered to the chess rules implemented in the program, such as pawns being able to move two squares only on their first move, capturing diagonally, and other pieces obeying restrictions like not being able to "jump" over other pieces. For example, I tested scenarios where invalid moves were blocked and appropriate feedback was displayed.

Using the DEBUG definition, I was able to extensively test edge cases, such as capturing pieces, achieving checkmate, and displaying captured pieces. I validated the correctness of the capture mechanics and ensured the program accurately tracked and displayed the captured pieces.

To test the save and load functionality, I saved a game after performing a sequence of moves. I memorized the moves made, the captured pieces, and the turn order before saving. Then, I loaded the saved game and verified that the board state, captured pieces, and move history matched the state when the game was saved. Additionally, I confirmed that the program correctly maintained the turn order, ensuring that if the game was saved after Black's turn, it resumed with White's turn upon loading.

**Test Results**

**Observations**

One of the most challenging aspects of developing this console chess game was translating the rules of piece movement into code. Implementing the movement logic for each piece, such as pawns being able to move two squares on their first move or bishops moving diagonally without jumping over other pieces, required careful thought and precise control of conditions. These rules had to be translated into logic that worked seamlessly with the chessboard array, which added complexity. Understanding how to manipulate the 2D array that represents the board, iterating over it with loops, and placing the conditions correctly was difficult. Grasping these concepts and applying them effectively to track piece positions and enforcing game rules demanded significant effort.

For future projects, I recognize that there is room for improvement in how the code is organized. A key enhancement would be to separate the program into multiple .c and .h files, grouping related functionality into modules for better structure and maintainability. Additionally, automating the compilation process using a Makefile would streamline development and ensure all dependencies are properly handled. These changes would improve the readability, scalability, and overall organization of the code, making it easier to manage and build upon in future iterations.