**Console-based Chess Game Proposal**

Ezequiel N. Buck Martínez

Texas Tech University

CS 1412: Programming Principles

Dr. Juan Carlos Rojas

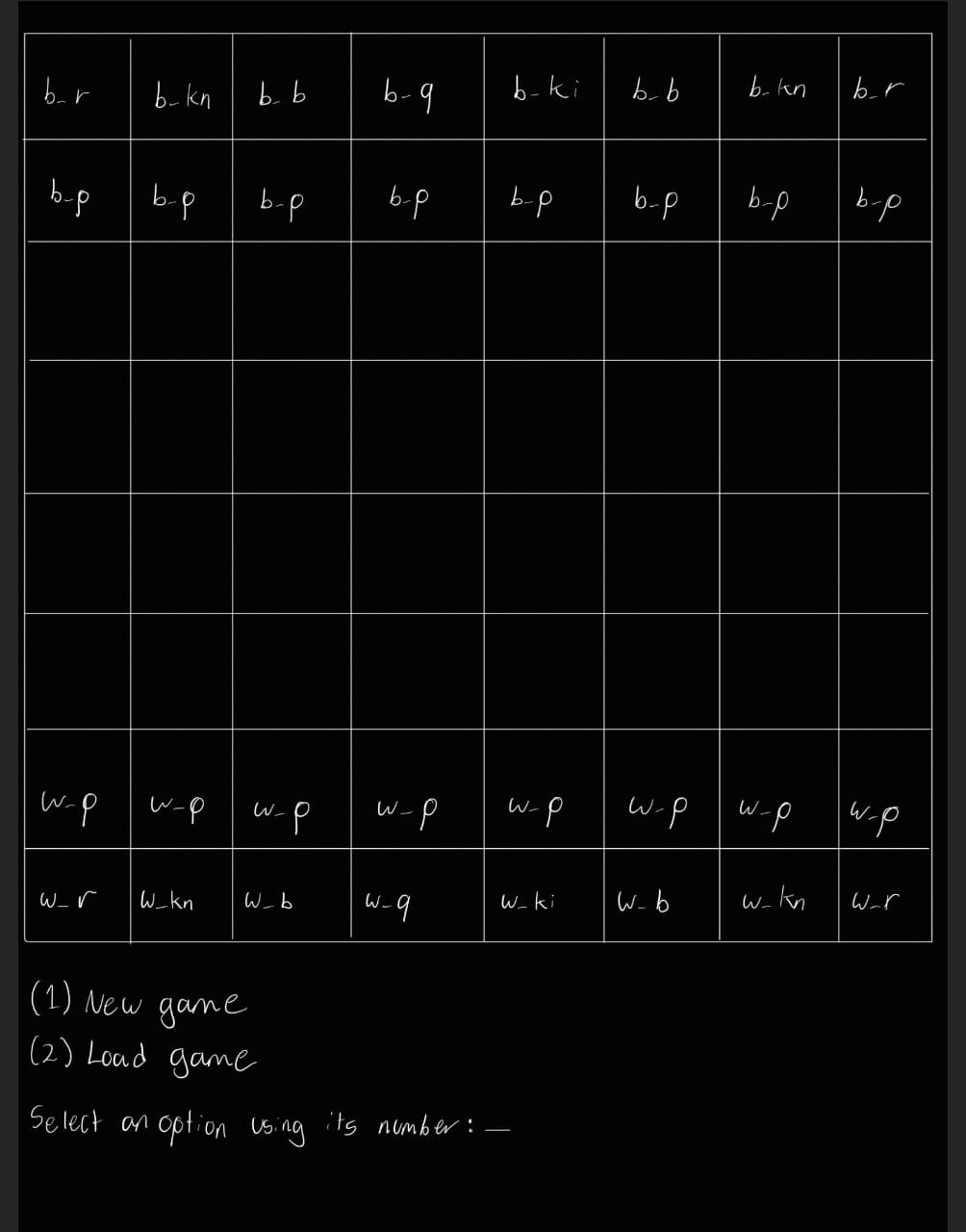
November 10, 2024

**Overview**

This project aims to create a console-based chess game that implements basic chess rules and game mechanics. When launched, the program will display a chess board showing all pieces in their starting positions, followed by a simple menu interface below the board for user interaction.

**Menu System**

The game starts by presenting players with a numbered menu offering two main options: "Start New Game" and "Load Game". Players make their selection by entering the corresponding menu number. The load game option will only work if there's an existing binary save file in the same directory as the game executable. If no save file exists, the game will display an error message and return to the main menu.



If time permits, selecting "New Game" will lead to a secondary menu where players can choose between playing against an AI opponent or entering a two-player mode where both players share the same computer (At the moment, only the 2-player mode is intended). In the case of choosing AI mode, one final prompt will ask the player to select whether they want to play as white or black pieces before beginning the game.

**Game Mechanics**

The core gameplay centers around a piece movement system where players specify their moves through text commands. Players identify pieces using a notation system - for example, "kn\_c3" represents a knight at position c3 and 'kn' stands for knight. After entering the piece identifier and pressing enter, the game will prompt for the desired destination square. Another option is that with a single console input, the piece and the destination can be specified in the same line using an arrow like this: “kn\_c3->d5”, but this would take more string processing.

The game will implement fundamental piece movement validation to ensure all moves follow chess rules. If a player attempts an invalid move, they'll be prompted to try again until they input a valid move. The game will maintain a history of moves, particularly important for special moves like castling that depend on whether pieces have moved previously.

For the initial version, the winning condition will be simplified to capturing the king, rather than implementing the more complex checkmate detection system. While this deviates from official chess rules, it provides a more manageable scope for the project's first iteration.

**Display System**

The game board will be displayed using text characters in the terminal, with pieces represented by their text notation (like w\_kn for white knight). While using actual chess piece icons in the terminal presents some challenges, there's potential to explore using Nerd Fonts for improved piece visualization. However, piece notation will still be needed internally and for the players to indicate the desired move.



*Chess nerd fonts. would need a console with nerd fonts too.*

**Program Structure (tentative)**

**1. Custom Data Types**

* Enums:
  + *PieceType*: Types of chess pieces (e.g., King, Queen, Rook, etc.).
  + *PlayerColor*: Enum for player colors (White, Black).
  + *GameStatus*: Game states (Playing, Check, Checkmate, Draw).
* Structs:
  + *Piece*: Represents each piece with its type, color, and position.
  + *Move*: Stores move information, including source and destination, piece moved, and any captured piece.
  + *Board*: A struct containing an 8x8 array of Piece pointers.
  + *Game*: Stores overall game state, including board, player turns, move history, etc.
* Unions (optional, depending on design):
  + For storing moves, using a union can help with handling different move types (e.g., regular, capture, en passant).

**2. Functions**

* Initialization Module
  + *initBoard*: Initializes the board and places pieces.
  + *initPiece*: Initializes a single piece.
  + *createGame*: Allocates and initializes the game struct.
* Input and Validation Module
  + *parseMove:* Parses user input for moves.
  + *validateMove*: Checks if a move is valid.
* Movement Module
  + *makeMove*: Updates the board with the given move.
  + *isCheck*: Determines if a player is in check.
* File Handling Module
  + *saveGame*: Saves the game state to a file.
  + *loadGame*: Loads the game state from a file.
* Utility Module
  + *printBoard*: Displays the board in the terminal.
  + *printMoveHistory*: Displays the history of moves.
  + *freeGame*: Frees all dynamically allocated memory.
  + *sort*: Some kind of sorting may be necessary.
* Main Game Loop
  + *playGame*: Handles the main game loop, switching turns and handling user input.
  + *checkGameStatus:* Checks if the game is over.