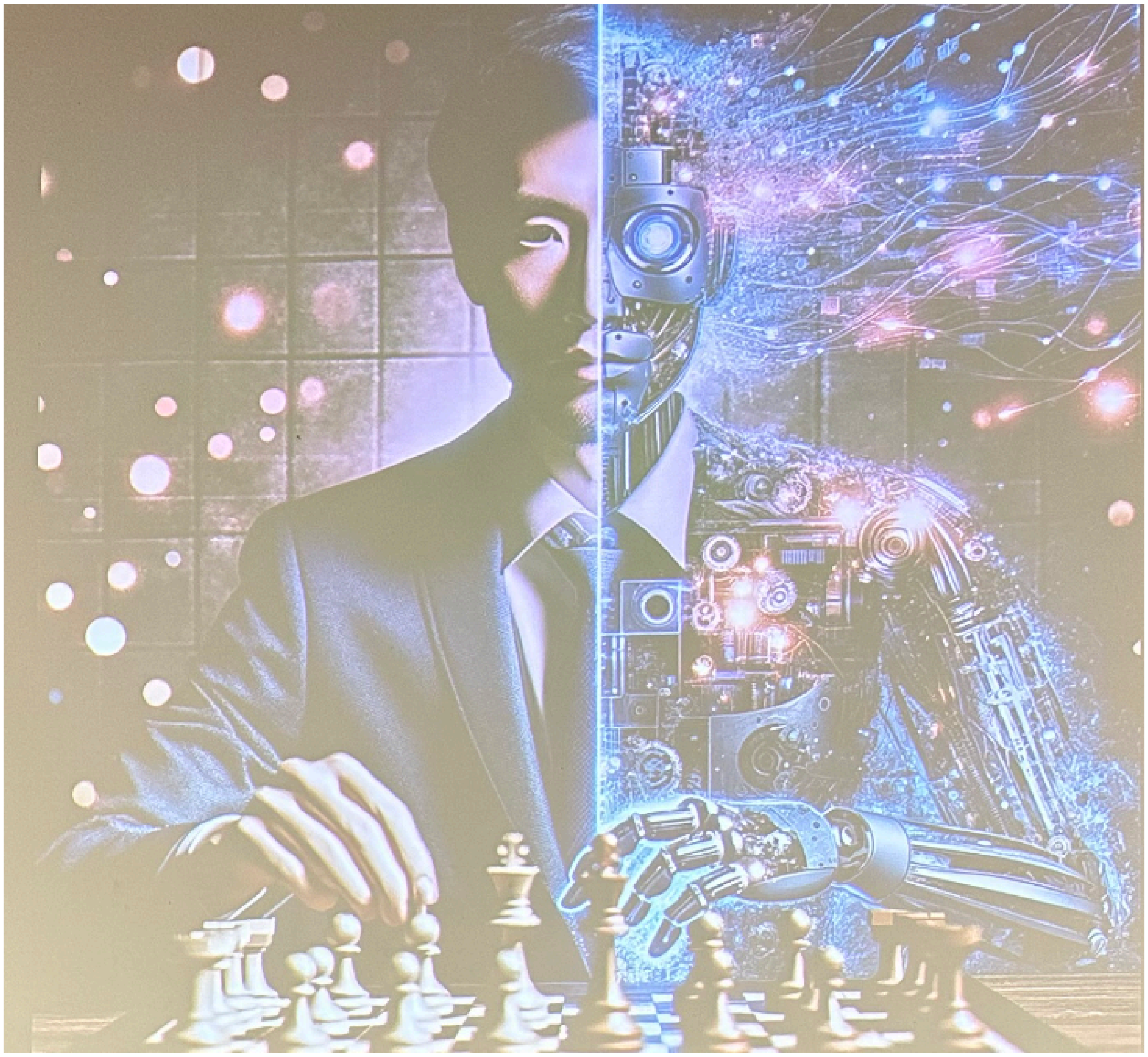
Martin Bot Chess Engine - Software Specification

V1.0



Authors: Kent Mencel, Albert Huang, Aakarsh Pasi, Johnny Wu, Liam Fricker, and Jimmy Le

In affiliation with the UCI EECS department and the EECS 22L class

April 08, 2024

Table of Contents

Glossary

A

Algorithm - Step-by-step procedure or formula for solving a problem or accomplishing a task, within a finite number of steps. They are sets of instructions that guide computers through the processes necessary to input, manipulate, and output data; the chess bot will use a variety of these to determine the best moves to make.

Application Programming Interface (API) - A set of rules and protocols for building and interacting with software applications. They define the methods and data formats that developers can use to communicate with programming libraries, software components, etc. They abstract complex code into simpler, reusable components.

Array - Fundamental data structure in programming that consists of a collection of elements, each identified by at least one array index or key.

Artificial Intelligence - Systems that can perform tasks which typically require human intelligence with the goal to augment/automate decision making. The chess bot that the player will play against is a prime example of this, given how it will make informed decisions based on a variety of factors (board states) and algorithms.

C

Character (char) - A data type used to store individual characters, such as letters, numbers, or symbols, and typically requires a single byte of memory.

Chess Algebraic Notation - A standardized method used to record and describe the moves in the game of chess. Each chess piece is denoted by their initials (K for King, Q for Queen, etc). Each square on the board represents a unique coordinate made up of a letter (a-h) representing the file and a number (1-8) representing the rank. A move is recorded by combining the piece's initial with the target square's coordinate, and captures are indicated with an x (example: Bxf5 means a bishop captures on f5).

D

Data Type - A classification that specifies the type of data a variable can hold and determines the operations that can be performed on it.

Dependencies - The external libraries, modules, or packages that a software project requires to function properly. They are pieces of code written and maintained by third parties that developers incorporate into their projects to reduce redundancy and to avoid reinventing the wheel.

Doubly Linked List - A type of data structure that consists of a sequence of elements called nodes where each node contains three components: data, a pointer to the next node, and a pointer to the previous node. It is called "doubly linked" because it allows traversal in both directions. The head of the list points to the first node and the tail points to the last.

E

Enumerate (enum) - A user-defined data type that provides the developer a way to assign symbolic names to a set of related values.

F

Function - A self-contained block of code designed to perform a specific task; it can be called by various parts of the program based on the hierarchy. They have unique names and can optionally accept parameters and return a value.

G

Graphical User Interface (GUI) - A type of user interface that allows users to interact with digital devices through graphical icons and visual indicators.

I

Input - Any data or instruction that is sent to a computer system for processing.

Integer (int) - A primitive data type used to represent whole numbers that can be positive, negative or zero.

L

Linked List - A fundamental data structure that consists of a series of connected nodes, where each node contains a data value and pointer to the next node in the sequence. The first node is the head and the last node is the tail.

M

Module - A self-contained unit of code that encapsulates a specific part of a system's functionality.

Module Hierarchy - Structured organization of code into a hierarchy of modules, reflecting the functional or logical decomposition of a system into smaller, manageable parts.

O

Output - Data that a computer system produces as a result of processing inputs.

P

Packages - A bundle of software components or modules that are grouped together under a common namespace, allowing for organized and reusable code.

Pointer - Variables that store the memory addresses of other variables. Allows for dynamic memory allocation and creation of complex data structures (linked lists, etc.).

S

Source Code - Collection of written instructions and statements written by a programmer using a human-readable programming language.

Struct - A composite data type that groups together variables under one name. These variables can be of different types.

T

Two-Dimensional Array - Essentially an array of arrays. A way to store data in a table-like structure. Useful to organize information in a grid, such as a chess board.

U

User Interface (UI) - The interface through which the user interacts with the program. It includes parts of how the program responds to user inputs, and all the other elements that the user can choose to interact with.

V

Void - The absence of a data type; it can be used in function definitions to specify that the function is not returning any value explicitly.

W

Whitespace- A character generally referring to spaces, indents, as well as newlines.

1 Software Architecture Overview.

1.1 Main data types and structures.

Log: list of all chess moves that have been made

- Data Type: Doubly-linked list
  - Move
  - First: Pointer to First Move (NULL if Log is Empty)
  - Last: Pointer to Last Move (NULL if Log is Empty)
  - Prev: points to previous Move
  - Next: points to next Move
  - Length: unsigned int indicating number of moves

Move:

- Data type: char array
  - Algebraic notation; i.e. Qa4
- Coordinate: int
  - 4 digits that represents the from and the to
- Capture: int
  - Represents whether the move was a capture or en passant
- Special: int
  - Represents whether the move was a Castle or Promotion

Piece: data structure containing Color, PieceType, PieceValue, Position, SquarePointer.

- Data Structure:
  - Color
  - PieceType
  - PieceValue: the numerical value determined by evaluation algorithms
  - Position (Same as Square Position)
  - SquarePointer: links back to Square data structure

Color: integer representation of the color of the player.

- Data Type: int
  - White is 0 and Black is 1.

PieceType: type of chess piece

- Data Structure:
  - Enum: Names of the Different Chess Pieces

Position: AlgebraicNotation representing chess location; i.e. “a4”

- Data Type: char array

Square: data structure containing a Pointer to Piece.

- Struct Data Structure
    - Pointers to Piece
    - Position

Board: 8x8 2D array of Squares.

PossibleMoves: List of Possible Moves Upon Selection of a Piece

- Doubly Linked List
    - Containing a String (of the coordinates of the available moves)

ThreatsList: identifies threats and moves that will lead to threats

- Doubly linked list of Threats
- Second Doubly Linked List
    - Contains legal moves for the selected piece that will lead to danger

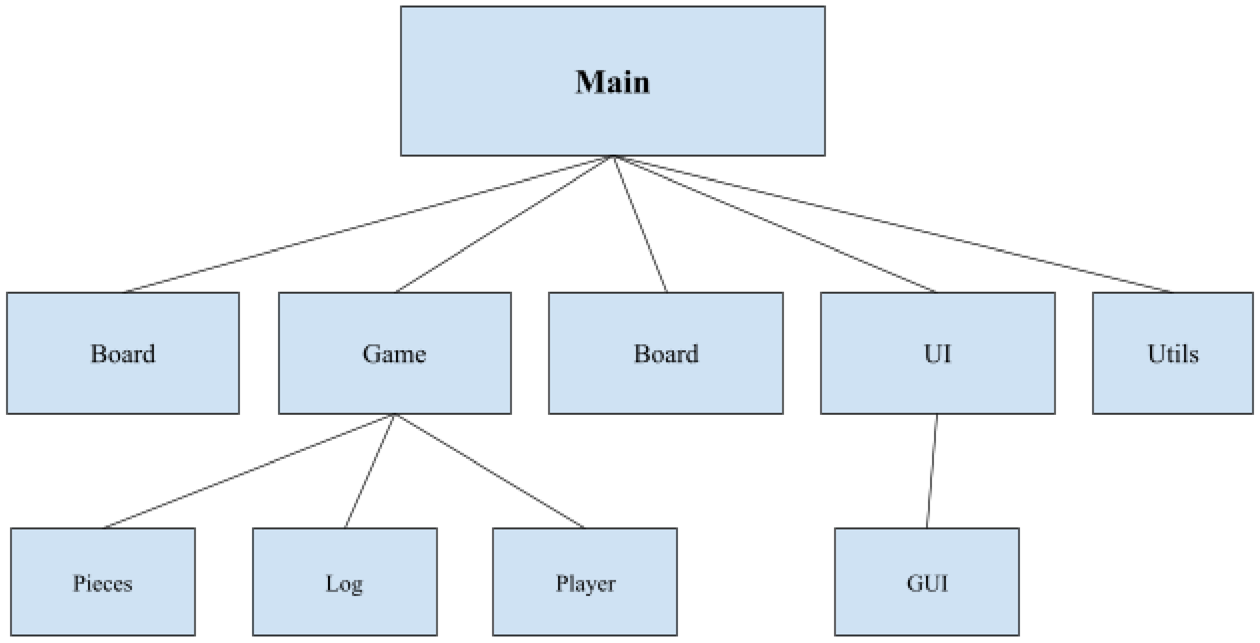
Threats

- Data Type: Struct
    - Positions of enemy pieces that can capture the selected square/piece
    - Enemy Piece Type

1.2 Major software components.

I. Board
    A. Squares
        1. Position
        2. Piece
            a) Color, PieceType, PieceValue, Position, SquarePointer
II. Log
    A. Moves
III. Possible Moves
    A. Moves
IV. Threats List
    A. Threats
V. Moves
VI. UI
    A. GUI

Diagram of module hierarchy:



1.3 Module interfaces.

Main

-Module Dependencies:

- Provides: Chess program
- Requires: all files

-Exported Functions:

- Main function
- void boardSetup(PieceType *pieceOrder)
    - Arguments: **pieceOrder**
    - Result: sets up the board according to the user's preferences

Board

-Module Dependencies:

- Provides: information regarding the board and location of pieces
- Requires: pieces

-Exported Functions:

- void CreateBoard()
    - Arguments: void

    - Result: create the board
- void DeleteBoard()
    - Arguments: void
    - Result: delete the board
- void EditBoard(Position,chessPiece)
    - Arguments: **Position**, **chessPiece**
    - Result: insert the chessPiece at the position
    - Note: if chessPiece is null, the position becomes empty

Game

-Module Dependencies:

- Provides: Setup of chess game
- Requires: board, log, pieces, player, chess engine

-Exported Functions:

- void Initialize(int option, int option2, int option3)
    - Arguments: **option1**, **option2**, **option3**
    - Result: initialize the game state
    - Note: option1, option2, option3, etc… are parameters that define whether the game is computer vs computer, player vs player, whether there is a timer, difficulty of the computer, etc…
- PossibleMoves legalMoves(const Move *move, const Board *board)
    - Arguments: **move**, **board**
    - Result: generates a list of legal moves
    - Note: has variables to specify what the move will result in; checks if game is in stalemate
- void Move(chessPiece, Position)
    - Arguments: **chessPiece**, **Position**
    - Result: move the chessPiece to the given position
    - Note: user will be prompted an error message if move is illegal
- void SeePossibleMoves(chessPiece)
    - Arguments: **chessPiece**
    - Result: a list of all possible moves of the chessPiece will be displayed

Log

-Module Dependencies:

- Provides: list of all previous moves
- Requires: none

-Exported Functions:

- void DisplayLog()
    - Arguments: void
    - Result: log of moves
- void AddLog()
    - Arguments: void
    - Result: add move to a log
- void RemoveLog()

    - Arguments: void
    - Result: remove move from a log
- void CreateLog()
    - Arguments: void
    - Result: create the log
- void DeleteLog()
    - Arguments: void
    - Result: delete the log

Pieces

-Module Dependencies:

- Provides: data structure definition of pieces
- Requires: nothing

-Exported Functions:

- PossibleMoves generateMoves(const Board *board, PlayerColor playerColor, Move moves[], int *moveCount)
    - Arguments: **board, playerColor, moves[], moveCount**
    - Results: outputs all the possible moves that a piece can make

ChessEngine

-Module Dependencies:

- Provides: chess engineer algorithms
- Requires: Pieces, Board, Log

-Exported Functions:

- move BestMove()
    - Arguments: void
    - Result: the best move according to the algorithm
- move goodMove()
    - Arguments: void
    - Result: a good move (top 5) according to the algorithm
- move badMove()
    - Arguments: void
    - Result: a bad move according to the algorithm
    - Note: this function is designed for the easy difficulty level
- int Evaluate(Board *board)
    - Arguments: **board**
    - Result: an integer representing the game state: positive if white is winning and negative if black is winning. The magnitude represents how much the side is winning by.

UI

-Module Dependencies:

- Provides: The user interface for starting the game, showing the end results, and managing the visual board state
- Requires: board, game

-Exported Functions:

- void showInstructions()
  - Arguments: void
  - Result: displays the game instructions
- void startGameUI()
  - Arguments: void
  - Result: starts the game by bringing up the game interface
- void showResult(PlayerColor winner)
  - Arguments: **winner** (color of player that won the game)
  - Result: displays a message depending on who won the game

GUI

-Module Dependencies:

- Provides: The graphical user interface for viewing the game; allows the user to either press and click or drag to make moves
- Requires: board, game, UI

-Exported Functions:

- void guiInitialize()
  - Arguments: void
  - Result: sets up the GUI
- void guiStartMenu()
  - Arguments: void
  - Result: displays the graphical start menu
- void guiGameBoard()
  - Arguments: void
  - Result: displays the game board
- void enterMovePress()
  - Arguments: void
  - Result: checks for a move made by pressing and clicking, and makes it if legal
- void enterMoveDrag()
  - Arguments: void
  - Result: checks for a move made by dragging, and makes it if legal

Utils

-Module Dependencies:

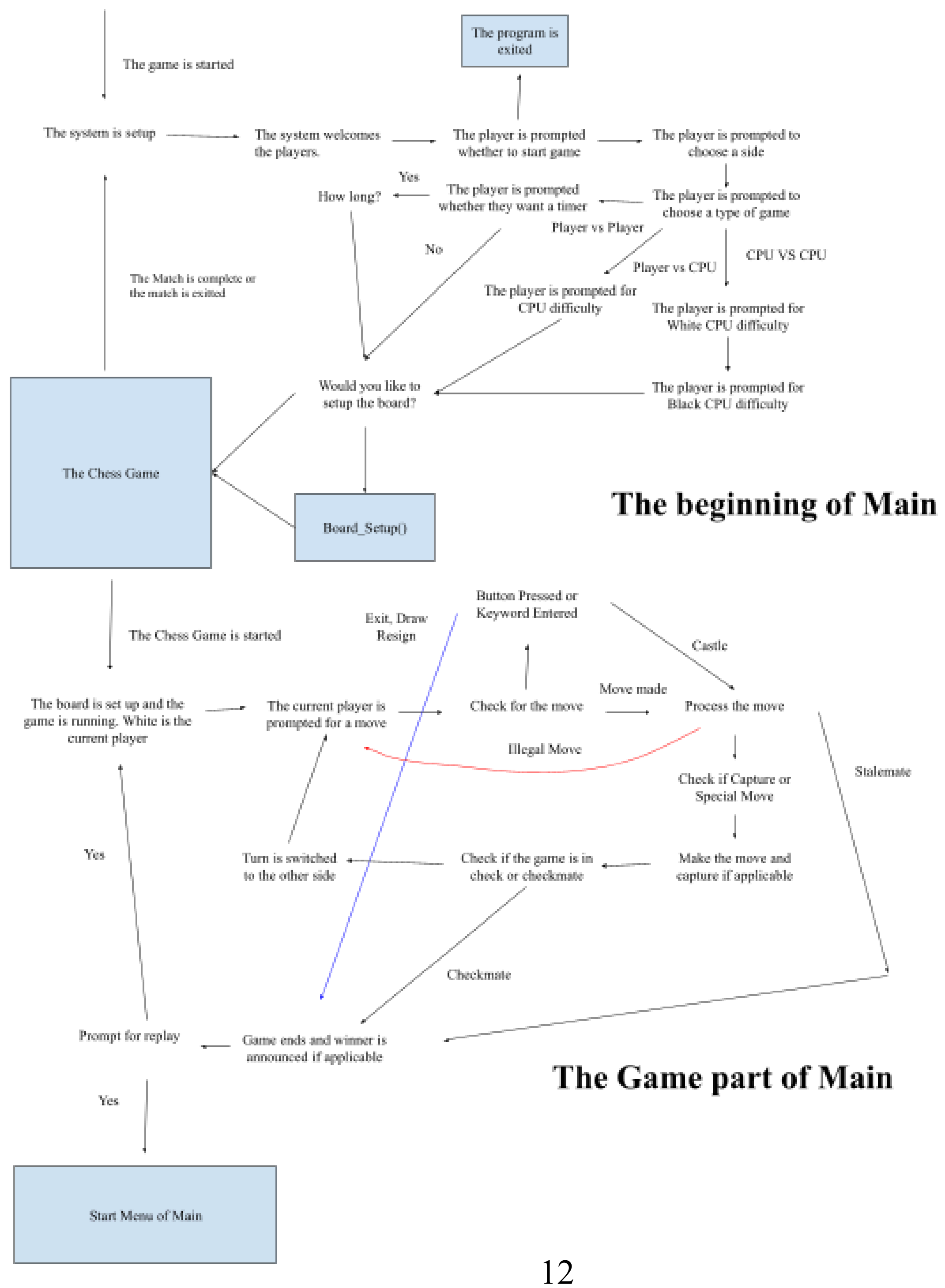
- Provides: utility functions such as checking if a given position lies within the bounds of the board, and trimming whitespace from a string
- Requires: none

-Exported Functions:

- int isValidPosition(int x, int y)
  - Arguments: **x**, **y** (x and y coordinates of a position)
  - Result: true/false depending on if the position (x, y) is valid or not
- char* trimWhitespace(char* str)
  - Arguments: **str** (any string)
  - Result: str, modified to have the whitespace removed

- void convertAlgebraic(Move *move)
  - Arguments: **move**
  - Result: converts the coordinates of a move to algebraic notation
- void convertCoordinate(Move *move)
  - Arguments: **move**
  - Result: converts the algebraic notation of a move to coordinates

1.4 Overall program control flow.


The game is started
The program is exited
The system is setup
The system welcomes the players.
The player is prompted whether to start game
The player is prompted to choose a side
Yes
How long?
The player is prompted whether they want a timer
The player is prompted to choose a type of game
Player vs Player
No
Player vs CPU
CPU VS CPU
The Match is complete or the match is exited
The player is prompted for CPU difficulty
The player is prompted for White CPU difficulty
Would you like to setup the board?
The player is prompted for Black CPU difficulty
The Chess Game
Board_Setup()
The beginning of Main
Button Pressed or Keyword Entered
Exit, Draw Resign
The Chess Game is started
Castle
Move made
The board is set up and the game is running. White is the current player
The current player is prompted for a move
Check for the move
Process the move
Illegal Move
Stalemate
Check if Capture or Special Move
Yes
Turn is switched to the other side
Check if the game is in check or checkmate
Make the move and capture if applicable
Checkmate
Prompt for replay
Game ends and winner is announced if applicable
The Game part of Main
Yes
Start Menu of Main

2 Installation.

2.1 System requirements, compatibility.

The recommended system requirements are:

- OS: Linux (64 bit)
- Processor: Anything that can run at at least 2 GHz
- Memory: 2 GB of RAM
- Graphic Card: Integrated graphics or dedicated GPU with basic capabilities
- Compatibility: Any computer, just needs to install or access Linux somehow

2.2 Setup and configuration.

The developer must have access to a machine running Linux OS or access to a server with Linux OS installed on it.

- The chess program is inside the '**Chess_V1.0_src.tar.gz**' file that contains all the source files needed to run the program.
- They can extract the archive by running "**gtar xvzf Chess_V1.0_src.tar.gz**".
- Go into the source file directory by using "cd **Chess_V1.0_src**"
- The developer can then run "**make**" and the chess executable should be in the "**bin**" directory.
- To run the chess program, the user enters the following command in the linux terminal: '**./chess_program**'. Then, all configurations of the chess game can be done via the start settings in the program.
- Alternatively, the user can also run "**make clean all test**" to compile and run the program.

Below are the contents of the makefile. It serves to compile all the files together. It also serves to provide for some shortcuts for git commands as well as for unit testing.

```
C/C++

CC=gcc
CFLAGS=-Wall -std=c11 -Iinclude
OBJDIR=bin
SRCDIR=src

OBJ=$(SRCDIR)/main.o $(SRCDIR)/chess.o $(SRCDIR)/chess_ai.o $(SRCDIR)/log.o
$(SRCDIR)/evalhelper.o $(SRCDIR)/evaluate.o $(SRCDIR)/midgame.o
$(SRCDIR)/boardControl.o $(SRCDIR)/earlygame.o $(SRCDIR)/endgame.o
$(SRCDIR)/GUI.o $(SRCDIR)/game.o

# Target to create the chess program
```

```
chess_program: $(OBJ)
    $(CC) $(CFLAGS) -o $(OBJDIR)/chess_program $(OBJ) -lSDL
    ./$(OBJDIR)/chess_program

all: chess_program

# Test targets
test_rulecheck.o: $(SRCDIR)/test_rulecheck.c
    $(CC) $(CFLAGS) -c $(SRCDIR)/test_rulecheck.c -o $(SRCDIR)/test_rulecheck.o

test_boarddisplay.o: $(SRCDIR)/test_boarddisplay.c
    $(CC) $(CFLAGS) -c $(SRCDIR)/test_boarddisplay.c -o
$(SRCDIR)/test_boarddisplay.o

testBoard: test_boarddisplay.o
    $(CC) $(CFLAGS) -o $(SRCDIR)/testBoard $(SRCDIR)/test_boarddisplay.o
    ./$(SRCDIR)/testBoard

testRule: test_rulecheck.o
    $(CC) $(CFLAGS) -o $(SRCDIR)/testRule $(SRCDIR)/test_rulecheck.o
    ./$(SRCDIR)/testRule

test: testBoard testRule

$(SRCDIR)/%.o: $(SRCDIR)/%.c
    $(CC) $(CFLAGS) -c $< -o $@

# Clean the build directory
clean:
    rm -f $(SRCDIR)/*.o $(OBJDIR)/chess_program $(SRCDIR)/testBoard
$(SRCDIR)/testRule

# Phony targets
.PHONY: clean all
```

2.3 Building, compilation, installation.

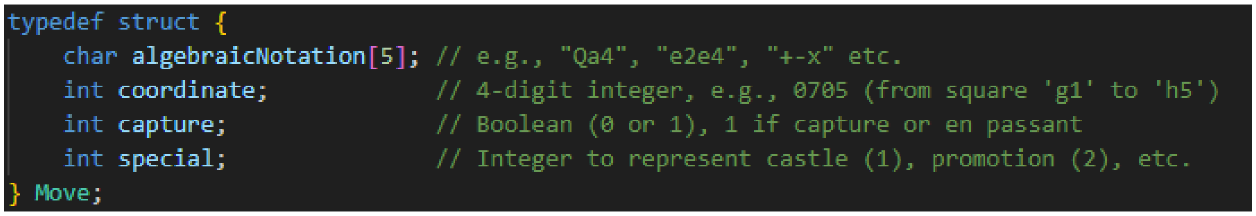
The program can be uninstalled by heading to the directory where the files extracted from ‘tar.gz’ file reside. Delete everything in this directory. Then, delete the ‘tar.gz’ file.

3 Documentation of packages, modules, interfaces.

3.1 Detailed description of data structures

Move:

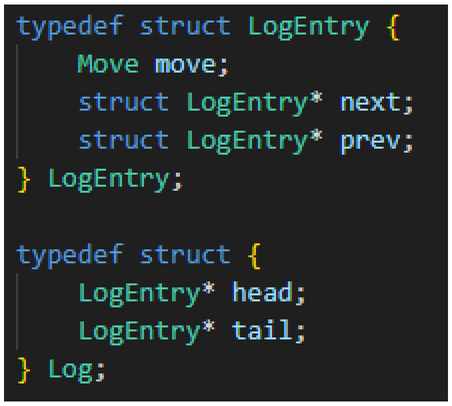
- Data type: char array
  - Algebraic notation; i.e. Qa4
- Coordinate: int
  - 4 digits that represents the from and the to
- Capture: int
  - Represents whether the move was a capture or en passant
- Special: int
  - Represents whether the move was a Castle or Promotion



```c
typedef struct {
    char algebraicNotation[5]; // e.g., "Qa4", "e2e4", "+-x" etc.
    int coordinate;            // 4-digit integer, e.g., 0705 (from square 'g1' to 'h5')
    int capture;               // Boolean (0 or 1), 1 if capture or en passant
    int special;               // Integer to represent castle (1), promotion (2), etc.
} Move;
```

Log: list of all chess moves that have been made

- Data type: doubly-linked list



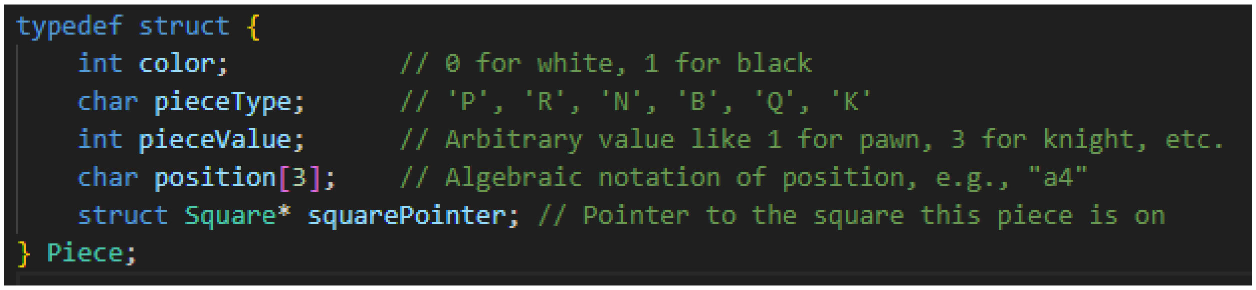
```c
typedef struct LogEntry {
    Move move;
    struct LogEntry* next;
    struct LogEntry* prev;
} LogEntry;

typedef struct {
    LogEntry* head;
    LogEntry* tail;
} Log;
```

Piece: data structure containing Color, PieceType, PieceValue, Position, SquarePointer.

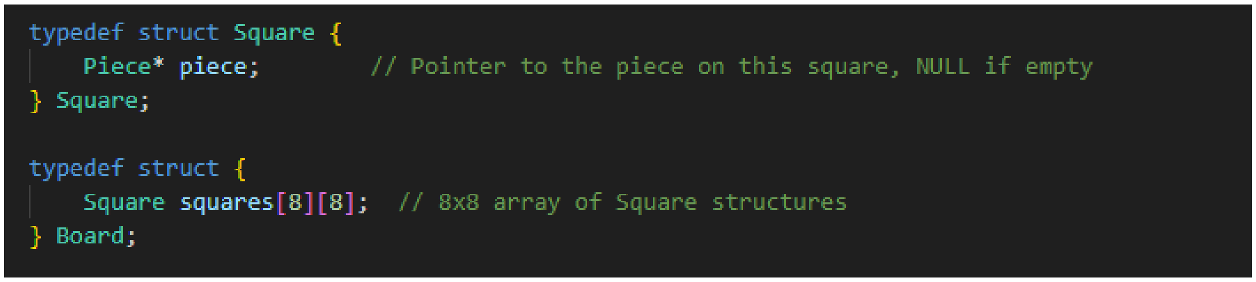
- Data Structure: struct
  - Color: integer representation of the color of the player.
    - Data Type: char
  - PieceType: type of chess piece
    - Data Type: char
  - Position: AlgebraicNotation representing chess location; i.e. “a4”
    - Data Type: char array
  - Square: Data structure containing a Pointer to Piece.

■ Data Structure: struct



```c
typedef struct {
    int color;            // 0 for white, 1 for black
    char pieceType;       // 'P', 'R', 'N', 'B', 'Q', 'K'
    int pieceValue;       // Arbitrary value like 1 for pawn, 3 for knight, etc.
    char position[3];     // Algebraic notation of position, e.g., "a4"
    struct Square* squarePointer; // Pointer to the square this piece is on
} Piece;
```

Board: 8 by 8 2D array of Squares.

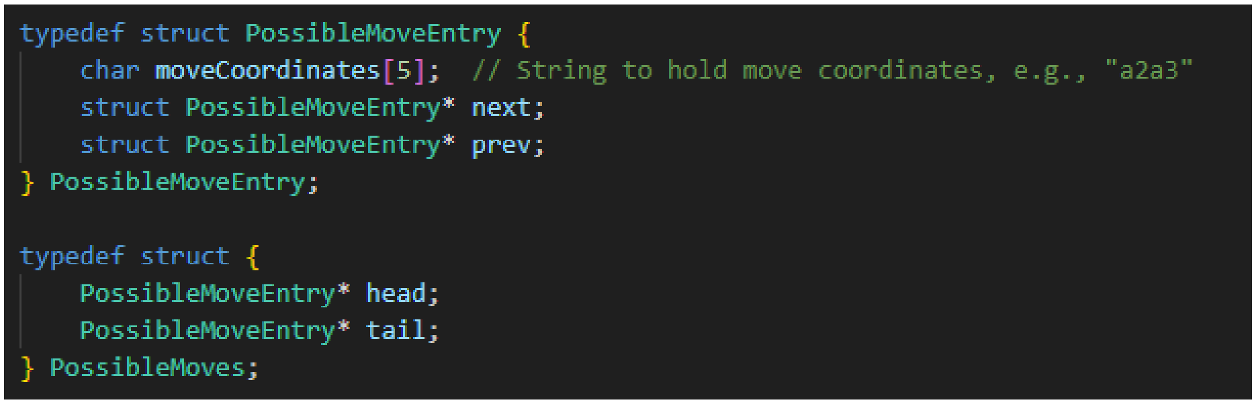


```c
typedef struct Square {
    Piece* piece;        // Pointer to the piece on this square, NULL if empty
} Square;

typedef struct {
    Square squares[8][8];  // 8x8 array of Square structures
} Board;
```

PossibleMoves: List of Possible Moves Upon Selection of a Piece

- Doubly Linked List
    - Containing a String (of the coordinates of the available moves)

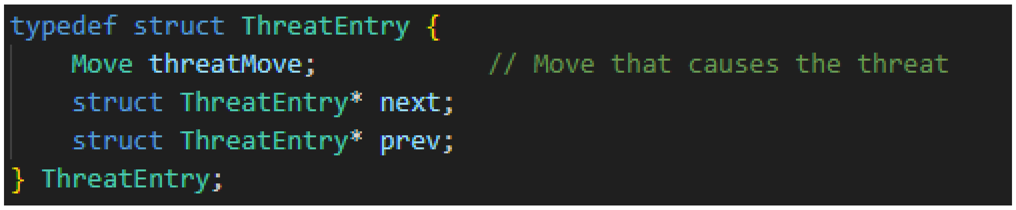


```c
typedef struct PossibleMoveEntry {
    char moveCoordinates[5];  // String to hold move coordinates, e.g., "a2a3"
    struct PossibleMoveEntry* next;
    struct PossibleMoveEntry* prev;
} PossibleMoveEntry;

typedef struct {
    PossibleMoveEntry* head;
    PossibleMoveEntry* tail;
} PossibleMoves;
```

Threats

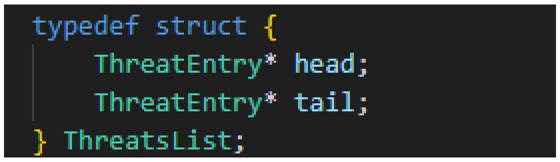
- Data Type: Struct



```c
typedef struct ThreatEntry {
    Move threatMove;            // Move that causes the threat
    struct ThreatEntry* next;
    struct ThreatEntry* prev;
} ThreatEntry;
```

ThreatsList: identifies threats and moves that will lead to threats

- Doubly linked list of Threats



```
typedef struct {
    ThreatEntry* head;
    ThreatEntry* tail;
} ThreatsList;
```

3.2 Detailed description of functions and parameters

Main

int main();

- The main function
- Setups the game
- Prompts the user for settings to setup the game
    - Which side they'd like to play
    - What mode they'd like to play
    - If a CPU is included in the mode, what difficulty will the CPU be.
    - If the mode is player vs player, will there be a timer
    - Would the player like to make any adjustments to the board
- Uses the information to call the various setup functions
- Begins the game
- Uses the functions from the Game module to process the game.
- Reacts to changes in game states

void boardSetup(PieceType *pieceOrder);

- Prompts the user for how they would like to set up the board
- Sets the piece order to how the user set the pieces as for each side
- Outputs a short error message if the user attempts to put a piece out of bounds, start a game without a King, or set pieces on the other side of the board.

Game

PossibleMoves legalMoves(const Move *move, const Board *board);

- Generates a list of legal moves.
- Also has a variable to specify whether that move would specify in a capture or an en passant
- Also has a variable to specify whether that move is a Castle or a Promotion
- Also checks if the game is in a stalemate state

int validateMove(const Move *move, const Board *board, int isCastle);

- Checks if a move is legal
- Also checks if a Castle move is legal

int isPathClear(const Board *board, const Move *move);

- Checks if there's any obstructions in a piece's path

int isCheck(const Board *board, PlayerColor playerColor);

- Checks if the game is in a Check state

int isCheckmate(const Board *board, PlayerColor playerColor);

- Checks if the game is in a Stalemate state

void makeMove(const Move *move, Board *board, PlayerColor *currentTurn);

- Moves the piece and switches the turn. It sets the spot where it was before to EMPTY

void makeCapture(const Move *move, Board *board, int enPassant);

- Removes the piece at the end of the move
- If the move is an en passant, removes the piece where that is captured by the en passant

void makeCastle(const Move *move, Board *board);

- Performs a castle

void makePromotion(const Move *move, Board *board);

- Performs a promotion.
- Will also prompt the user to choose what piece they'd like to promote it to.

Board

Board initBoard();

- Used to set up the board.
- Meant to be called initially by game or main.

void clearBoard(Board *board);

- Sets every position and color on the board to EMPTY values

void setupInitialPosition(Board *board);

- Sets up the initial position of every piece of a conventional chess game void setupInitialPosition(Board *board, PieceType *pieceOrder, int rowSize);

- Sets up the initial position of every piece as specified by the user.

void displayBoard(const Board *board);

- A prototype version of the display in the Terminal in case the GUI fails.

Log

void moveToString(const Move *move, char *moveStr);

- Converts a move to a string

void logMove(const Move *move, const char* fileName);

- Adds the move to the log file

void displayLog();

- Displays the history of the moves to the terminal or the GUI

void saveLogToFile(const char* fileName);

- If the user wants to save their log after a game

void addCapturedPieces(PieceType piece);

- Adds and records a captured piece

void getCapturedPieces();

- Displays the captured pieces.

Pieces

PossibleMoves generateMoves(const Board *board, PlayerColor playerColor, Move moves[], int *moveCount);

- Generates the moves that a piece can make

PossibleMoves addPawnMoves(const Board *board, int x, int y, Move moves[], int *moveCount);

- Generates the moves that a pawn can make
- This includes en passant

PossibleMoves addBishopMoves(const Board *board, int x, int y, Move moves[], int *moveCount);

- Generates the moves that a bishop can make

PossibleMoves addKnightMoves(const Board *board, int x, int y, Move moves[], int *moveCount);

- Generates the moves that a Knight can make

PossibleMoves addRookMoves(const Board *board, int x, int y, Move moves[], int *moveCount);

- Generates the moves that a Rook can make
- This does not include Castling

PossibleMoves addQueenMoves(const Board *board, int x, int y, Move moves[], int *moveCount);

- Generates the moves that a Queen can make

PossibleMoves addKingMoves(const Board *board, int x, int y, Move moves[], int *moveCount);

- Generates the moves that a King can make
- This does include Castling

Player

Move getPlayerMove(const Player *player, const Board *board);

- Prompts the player for a move
- Returns the move the player made

void computerMove(const Board *board, Move *move, PlayerColor color);

- Generates a random move.
- Meant to be a placeholder or a easy computer.

void competantComputerMove(const Board *board, Move *move, PlayerColor color);

- The Computer analyzes the board and makes a reasonable move
- It predicts the outcome of its moves as well as the value of the move.

void advancedComputerMove(const Board *board, Move *move, PlayerColor color);

- The Computer analyzes the board and makes a strong move
- It predicts the outcome of its moves and future moves as well as the value of the move and subsequent moves.

UI

void showInstructions();

- Displays the instructions

void startGameUI();

- Displays the start menu

void showResult(PlayerColor winner);

- Displays the winner

void displayBoard(const Board *board);

- Displays the board

void updateBoard(const Board *board, Move *move, int isCapture, int isCastlePromotion);

- Makes a move on the board

void displayMoveHint(PossibleMoves movelist);

- Displays the list of possible moves on the board

GUI

void guiInitialize():

- Initializes the GUI
- Setups the components of the GUI

void guiStartMenu():

- Setups the GUI of the start menu
- Displays the startMenu

void guiGameBoard():

- Setups the GUI of the game
- Displays the game

void enterMovePress();

- Checks for a Move made through press
- Displays the possible moves
- Makes the move if legal
    - Outputs a message if not.

void enterMoveDrag();

- Checks for a Move made through drag
- Displays the possible moves
- Make the move if legal.
    - Outputs a message if not.

Utils

int isValidPosition(int x, int y);

- Checks if a position in the board is in the board's dimensions

char* trimWhitespace(char* str);

- Trims Whitespace from the beginning and end of a string

void convertAlgebraic(Move *move);

- Takes the coordinates of a moves and converts it to algebraic

void convertCoordinate(Move *move);

- Takes the algebraic notation of a move and converts it to coordinates

3.3 Detailed description of input and output formats

Coordinates or Chess Algebraic Notation:

INPUT: In the terminal: "W" "White" "1" "Player vs CPU" "Player vs Player"

In the start menu, the player will be prompted to select certain options/settings to calibrate the game. For example, choosing White or Black or choosing Player vs CPU, Player vs Player, or CPU vs CPU.

If an invalid option is selected, a brief error message will appear in the terminal and the player will be prompted again.

The input will be processed and stored in main and then used during set up of the board and the game.

INPUT: In the terminal: "A5 A6"

Choose a piece at A5 and move it one space forwards to A6.

This input format is one of the ways the user can input a move.

If an invalid location or an illegal move is made, an error message will state this and then re-prompt the user for an input.

A castle can also be performed this way i.e. **"D1 B1"** given that the King is at location D1 and then the program will check if a castle is applicable.

A move input will be stored in the Move type. If the move was not made in algebraic notation, it will be converted to algebraic notation and logged in the log file. This will be the same for all move inputs.

INPUT: In the terminal: "a6" or "aa6" or "5a6"

Move a pawn to A6.

This input format is one of the ways the user can input a move.

If an invalid location or an illegal move is made, an error message will state this and then re-prompt the user for an input. If there are two pieces that can be moved with the same algebraic notation, the program will re-prompt the user to specify which piece is to be moved.

A castle can also be performed this way i.e. **"O-O" or "O-O-O"** and then the program will check if a castle is applicable.

INPUT: In the terminal: "HISTORY"

The terminal will also check for certain keywords in the input if the input entered wasn't applicable. This keyword would bring up the history of all the moves made.

A castle can also be performed this way by entering "CASTLE". This performs a castle if applicable

A keyword input will cause a function to be called. It will not be stored in the log file unless a Castle was performed or Draw/Resign/Exit was called. This will be the same for all keyword type inputs.

INPUT: Button Press: PIECE LOCATION move to ENDING LOCATION

Clicking on a piece allows the user to click a different location to move the piece. The piece clicked will be highlighted in green.

If the player selects a piece, the piece will be highlighted in a dull green modulate as well as the possible locations it can be moved to. If the user clicks an invalid location, the original piece will then be highlighted in red and no change in position will be made. An error message will then appear in the terminal.

A castle can also be performed this way i.e. **"[PRESS KING] move to [CASTLE LOCATION]"** and then the program will check if a castle is applicable.

INPUT: Button Press: (BUTTON DOWN - PIECE LOCATION) move cursor to (ENDING LOCATION - BUTTON UP)

Left clicking on a piece and holding without releasing allows the User to have the piece follow the mouse cursor. The user can then release the left mouse button to drop the piece at that location.

If the player selects a piece, the piece will be highlighted in a dull green modulate as well as the possible locations it can be moved to. If the user clicks an invalid location, the original piece will then be highlighted in red and no change in position will be made. An error message will then appear in the terminal.

INPUT: Button Press: RESIGN

There will also be some buttons with functions on the GUI. For example, this button would resign the game if pressed. This button would also have a confirmation before resigning as resigning is a big task and may be performed accidentally.

OUTPUT: Terminal:

The Terminal will often output a message in relation to an input being received. For example, for an option choice, the terminal will restate that option choice before outputting the effects of the choice. For a move choice, the terminal will state what piece was moved and where it was moved to, whether it captured any units or caused any game state changes such as check or checkmate.

OUTPUT: GUI

The GUI will show the results of the board state with the positions of all the pieces, the timer, as well as what pieces have been captured. The GUI will also show the pieces being moved as well as the effects the moves may make.

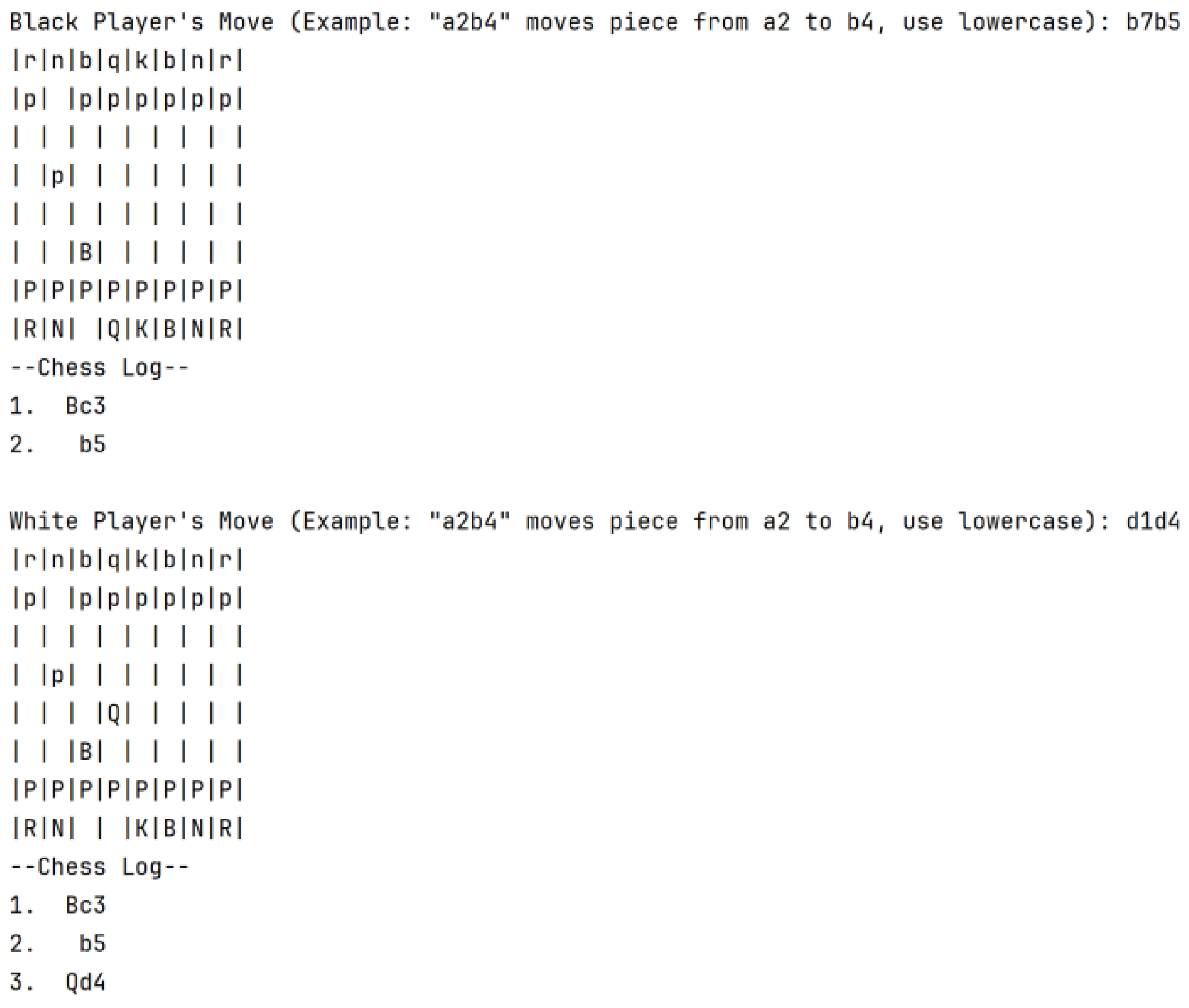
OUTPUT: Sound

There will occasionally be sound being outputted, whether from a piece move or winning a game.

OUTPUT: Log

The Log of the options and the move choices will be recorded in the log file. This can also be accessed through the view history option.

The moves will be logged in algebraic notation, in order of when the moves were made. An example of this is shown in the image below, where the moves made in the game are registered in the log file and can be outputted in game.



```
Black Player's Move (Example: "a2b4" moves piece from a2 to b4, use lowercase): b7b5
|r|n|b|q|k|b|n|r|
|p| |p|p|p|p|p|p|
| | | | | | | | |
| |p| | | | | | |
| | | | | | | | |
| | |B| | | | | |
|P|P|P|P|P|P|P|P|
|R|N| |Q|K|B|N|R|
--Chess Log--
1.  Bc3
2.   b5

White Player's Move (Example: "a2b4" moves piece from a2 to b4, use lowercase): d1d4
|r|n|b|q|k|b|n|r|
|p| |p|p|p|p|p|p|
| | | | | | | | |
| |p| | | | | | |
| | | |Q| | | | |
| | |B| | | | | |
|P|P|P|P|P|P|P|P|
|R|N| | |K|B|N|R|
--Chess Log--
1.  Bc3
2.   b5
3.  Qd4
```

4 Development plan and timeline (7)

4.1 Partitioning of tasks

End of Week 1: Basic Blueprint and outline of idea

End of Week 2: Rudimentary Code Development of blueprint and partitioning of responsibilities

End of Week 3: Projected Basic Engine Completion and Presentation Readiness

- A Chess Engine that can play a game of chess according to the FIDE rules.
    - This includes setting up the pieces correctly
    - This includes making moves and capturing possible
    - Making sure these moves are legal
    - Having functionality of special moves such as Castling, Promotion, and Enpassant.
    - Making check, stalemate, and checkmate correct and possible.
- Having a start menu
- Developing a CPU that can play against a player without a huge wait time or performing illegal moves.
- Developing a responsive and intuitive GUI
    - This includes a GUI for the start menu as well as the board
    - This also includes having buttons for the various functions such as "draw"
    - Having the board show where you can move pieces to

End of Week 4: Engine Development complete and Completion of Project.

- Engine needs to work correctly and needs to be able to be executed.
- The most advanced CPU needs to be functioning
- The GUI has to be compatible with the Chess Engine
- Complete User Manual
- Complete Software Specification
- Complete Documentation

4.2 Team Member Responsibilities

The main three sections we will be partitioning our work into are

- The Underlying Chess Engine
- The CPU
- The GUI

Based on the intensities of each section we will be dividing our group members to work on each section:

The Underlying Chess Engine (Aakarsh Pasi, Johnny Wu, Jimmy Le)

The responsibilities of the people working on the Chess Engine will include

- Developing a Chess Engine that can play a game of chess according to the FIDE rules.
    - This includes setting up the pieces correctly
    - This includes making moves and capturing possible
    - Making sure these moves are legal
    - Having functionality of special moves such as Castling, Promotion, and En Passant.
    - Making check, stalemate, and checkmate correct and possible.
- Having a start menu
- Adding additional options to the game such as “Draw”, “Undo”, “Pause”, “History”, “Evaluation”, and “Rules”
- Adding additional modes/options to the game:
    - Choosing Side
    - Versus Modes
        - Player vs CPU
        - Player vs Player
        - CPU vs CPU
    - Timer
    - Board Arrangement.

The CPU (Albert Huang and Kent Mencel)

The responsibilities of the people working on the Chess Engine will include

- Developing a CPU that can play against a player without a huge wait time or performing illegal moves.
- Developing a more Advanced CPU that can perform special moves as well as make informed, educated decisions
- Developing a powerful CPU that can predict moves in the future as well as stand a chance to win the tournament.
- Make an option to choose between the three CPUs
- Make the CPU compatible with different board sets (or perhaps have a CPU by itself that can handle different board sets)

The GUI (Liam Fricker)

The responsibilities of the people working on the GUI will include

- Developing a responsive and intuitive GUI
    - This includes a GUI for the start menu as well as the board
    - This also includes having buttons for the various functions such as “draw”
    - Having the board show where you can move pieces to
- Having the visuals meet the expectations of a professional Chess Engine
- Adding minor quirks such as Piece Movement and Capture Tweens and Sound Effects.
- Adding Ease of Access.

Back matter

Copyright.

References

https://www.fide.com/FIDE/handbook/LawsOfChess.pdf

https://newport.eecs.uci.edu/~doemer/eee_uci_edu/18w/18020/Lecture1.pdf

https://newport.eecs.uci.edu/~doemer/eee_uci_edu/18w/18020/Lecture2.pdf

https://newport.eecs.uci.edu/~doemer/eee_uci_edu/18w/18020/Lecture3.pdf

https://docs.google.com/presentation/d/1-0ltlNXjAMsfNlFpEchcK5KIYdjSlYaweOmc7aBANbQ/edit#slide=id.p

https://docs.google.com/presentation/d/1r9yeeTayXnU5S40c1_we8AkyN3bpdwyCPycjZ8EQ8uI/edit#slide=id.p

Index