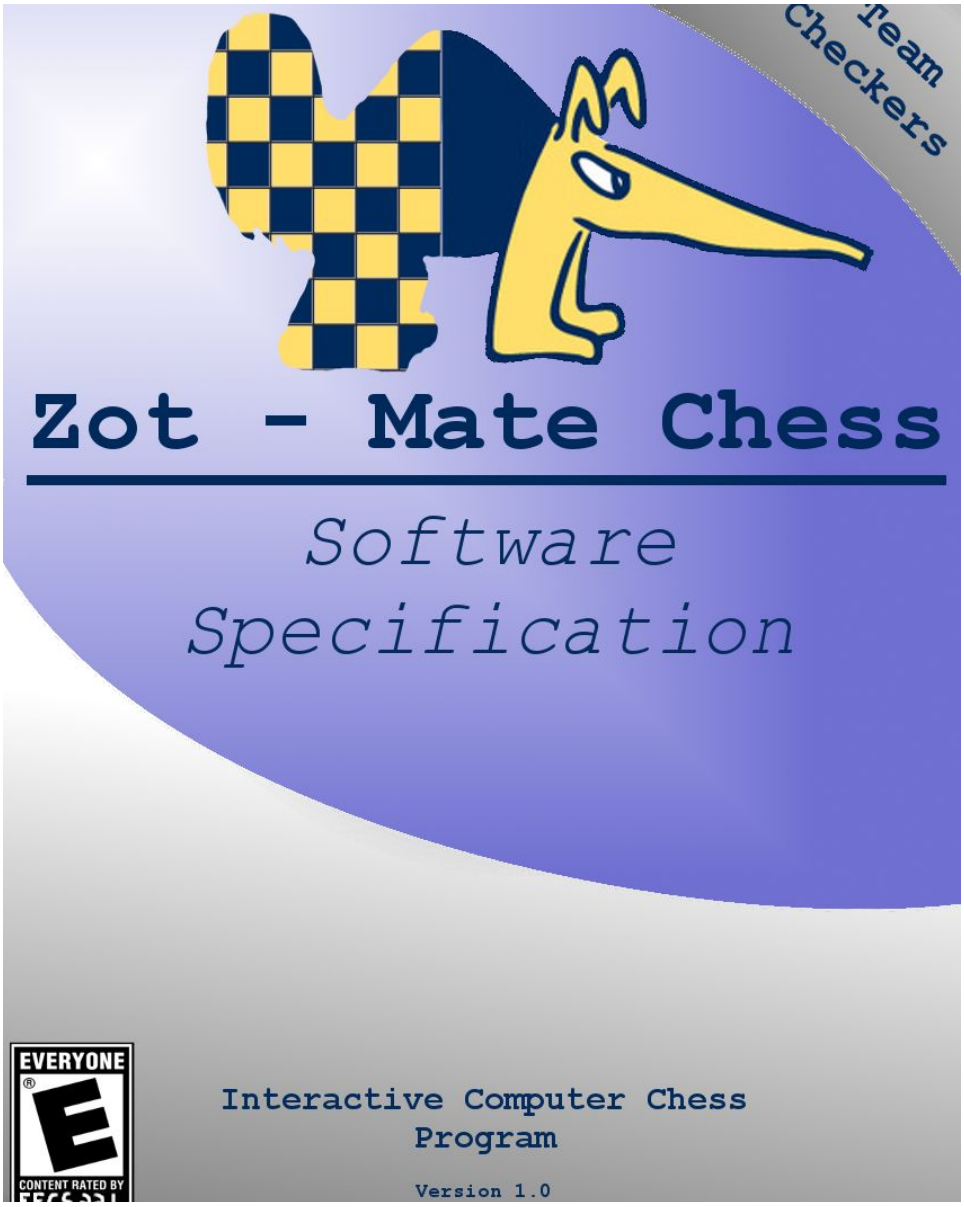
## Checkers

Bianca Corine Doronila
Cesar Alvarenga
Joshua Miller
Michael Reza Honar
Michael Barbosa
Tun Myat

In affiliation with: The Henry Samueli School of Engineering at The University of California Irvine




# Zot-Mate version 1.0

**22nd January 2018**

## Table of Contents

## Glossary

| | | |
|---|---|---|
| **array** | - | multiple data types, of the same type that can can be accessed through one call as a list |
| **char** | - | data type containing [-255, 255] |
| **cmd** | - | short for “command” |
| **data type** | - | declarations that determine the characteristic of the memory location or variable |
| **header file** | - | file containing function declarations only, ending in .h |
| **Makefile** | - | a script that can choose the appropriate files to be compiled and linked together |
| **pointer** | - | a variable that contains the address of another variable |
| **source file** | - | file containing function definitions only, ending in .c |
| **string** | - | array of characters (char) |
| **struct** | - | developer defined data type |
| **void function** | - | a function type that ends either by reaching the end of the function that does not need to return a value |

# 1 Software Architecture Overview

## 1.1 Main Data Types and Structures

- **Chess Layout.h**
    - InitializeBoard
        - 8x8 pointer array
        - Points to either null or ‘Piece’ struct
    - DeleteBoard
        - Deletes a board array
    - PrintBoard
        - Prints a board
    - Piece
        - char Type
        - char  Color
        - char  Value
        - char  Moved
        - char  Xpos
        - char  Ypos
    - theBoard
        - Piece board[8][8]
- **Move.h**
    - Move
        - char initX
        - char initY
        - *Piece Mover
        - char finX
        - char finY
        - *Piece Capture
    - MoveListEntry
        - *List List
        - *Entry Next
        - *Entry Prev
        - *Move Move
    - MoveList
        - *PieceMover
        - char Length

        - *Entry First
        - *Entry Last
- **Alive.h**
    - AliveList
        - unsigned int Length
        - unsigned int Color
        - *Entry First
        - *Entry Last
    - AliveListEntry
        - *List List
        - *Entry Next
        - *Entry Prev
        - *Piece AlivePiece
- **Check.h**
    - int TestCheckLegal
    - Int TestCheckMate
    - Piece TestCheckLegalAttacker
    - Piece TestCheckLegalBlocker
- **FileIO.h**
    - int WriteToFile
    - intStringToMove
    - int MoveToString
    - LogMove
    - LogCheck
    - LogMate
- **Legal.h**
    - *List PawnLegalMoveListGen
    - *List KingLegalMoveListGen
    - *List RookLegalMoveListGen
    - *ListBishopLegalMoveListGen
    - *ListQueenLegalMoveListGen
    - *ListKnightLegalMoveListGen
- **Locomotion.h**
    - movePiece
    - *Piece Promotion
    - capture
- **AI.h**
    - AliveList
        - char Length

        - char Color
        - *Entry First
        - *Entry Last
    - AliveListEntry
        - *AliveList List
        - *Entry Next
        - *Entry Prev
        - *Piece Piece

## 1.2 Major Software Components

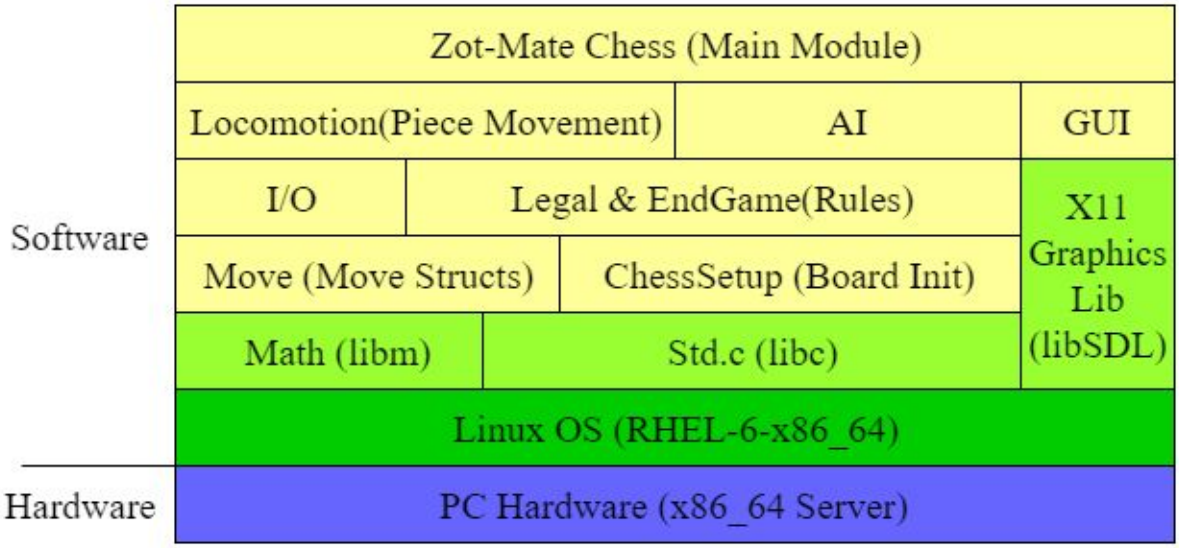
- There are four major software components in our game:
    1. Artificial Intelligence Decision Sequence
        - This sequence is based around the list of the AI player's pieces on the board. For every AI piece on the board, a double-linked list of legal moves will be created. Decision of which move to pick for each piece depends on the difficulty set for the AI.
            - On Easy, the move is made at random
            - On Medium, half of the moves are random, half of the moves are calculated by single-layer, points-based system
            - On Hard, all of the moves are calculated based on a dual layer (current and next move) system.
        - The points system is based on the current set: {255, 9, 5, 3, 1}, where the king is worth 255 points, the queen 9, the rook and bishop 5 each, the knight 3, and a pawn 1. Each possible move calculated in the function "GenLegalList(Piece Peice, board* Board, char isCheck);" is given a numerical value based upon the piece it may knock off the board.
        - After a move is assigned to each piece on the AI, the sequence will then select the piece with the most valuable points. For example, if the AI's first pawn can kill another pawn, and the AI's second pawn can kill a rook, then the AI will choose the latter move as it is more valuable.
        - After the AI's move, it is the player's turn

2. Input Legal Move Validator
    - This Component of our software is involved in making sure a user's input is correct. Our chess program will take in a user's input and process it as a move. The move "A2 A4" will move the pawn in the "A" column up two spaces, after going through a sequence:
        - Is there a piece in the starting location?
        - Is the piece the correct color?
    - If both hypothetical questions are validated as true, then the sequence proceeds with the current flag
        - Is the player making the move in check?
            - 0 or 1 (1 means in check) this parameter is recorded under the variable name "check"
    - With a flag for "check", the program now compares the passed move to the list of legal moves. If the passed move is identical to an entry of the legal moves, then the move is processed on the board.
    - If the player is in check, we validate that the move blocks the attack and takes the player out of check
    - After the move has been processed on the board, this sequence is complete

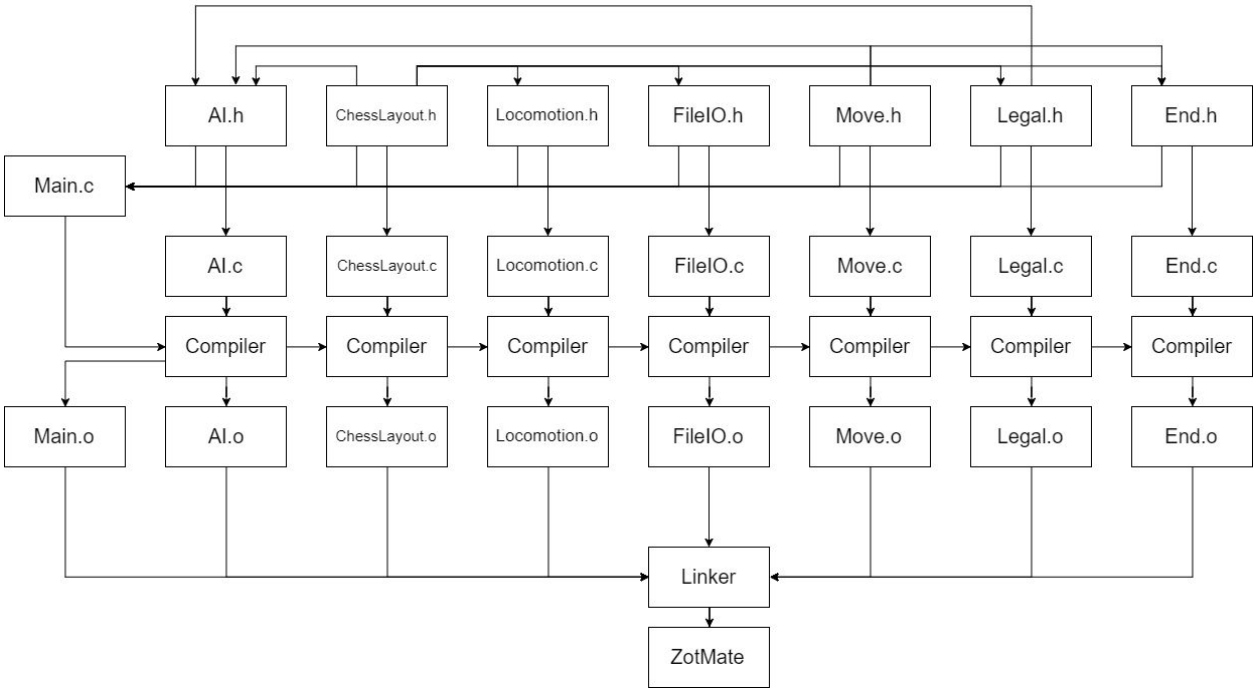
3. Check or Mate Checker Sequence
    - This Sequence checks to see if a player (Human or AI) is placed in check or checkmate. It works by first analyzing the checked King's possible moves. If none of the King's moves are valid (by looping through all the combinations), then the program considers blocking the piece that is attacking the King. This is done by looping through all the pieces and finding a move that is in the way of the line of attack.
        - Note: the attack of a Knight cannot be blocked - if a King is put under check by a knight and cannot move anywhere, the King is checkmated.

4. File Logger
    - After every move, a C-string stream consisting of the moves (in the format of [‘moveNumber’ ‘start’ ‘end’ - ‘color’ ‘time’] will be output onto a log.txt file.
    - Example entries include:
        - “1 A2 A4 - white pawn - 2s” to indicate a white pawn moved two spaces, and the move lasted 30 seconds
        - “567 Black King E8 in check by white bishop C6 - 7s”
        - “859 Black King F8 in checkmate by white Queen D6 - 5s -Game Over! ”

### 1.2.1 Diagram of Module Hierarchy



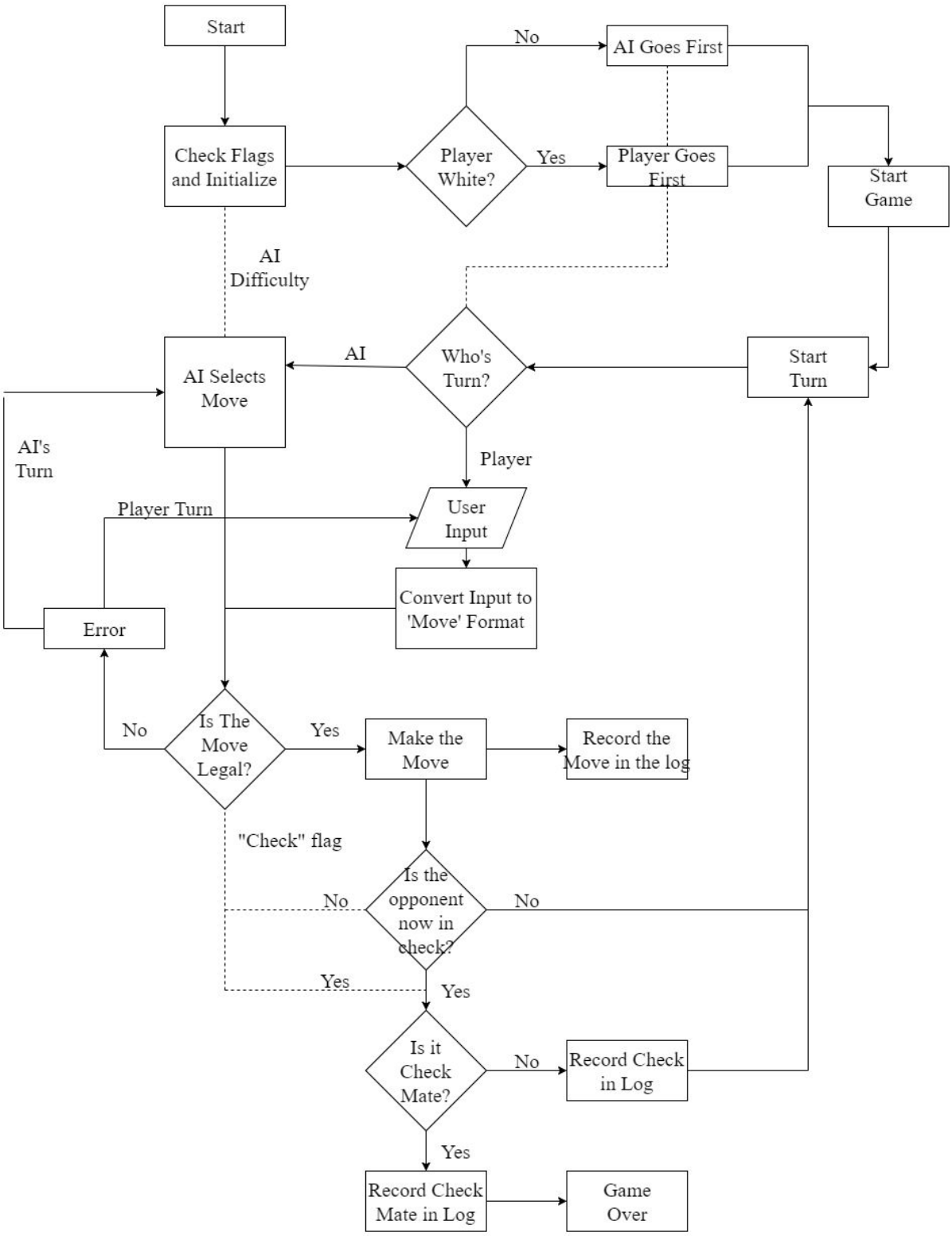
## 1.3 Module Interfaces




### 1.3.1 API of major module functions

- Interaction of APIs covered in 1.2

## 01.4 Overall Program Control Flow



# 2 Installation

## 2.1 System Requirements and Compatibility

- OS: Linux access on either: Windows 7 or higher or macOS
- Processor: 2.0 GHz Dual Core
- Memory: 2GB RAM
- Network: Broadband Internet Connection
- Storage: 500 MB
- Terminal Shell

## 2.2 Setup and Configuration

- Open the command prompt (Terminal, MobaXterm, PuTTY)
- Login to SSH server
- Extract directory from the repository by using CVS
- Run command “make all” generated by the Makefile
- Execute the program “./chess”

## 2.3 Building, Compilation, and Installation

- Section 2.2 covered building, compiling, and installation.

# 3. Documentation of  Packages, Modules, and Interfaces

## 3.1 Data Structures

- **Board:**

  ***Piece [8][8];**

  The board will be represented by an 8x8 array of pointers that will either point to different Piece structures or contain null pointers, representing the location of the pieces on the chess board as well as the location of empty squares on the chess board respectively. This will allow us to intuitively and easily keep track of the locations of our pieces based upon the indices of the array.

- **Piece:**

| Piece |
| --- |
| **char PieceType** |
| **char Color** |
| **char Value** |
| **char Moved** |
| **char Xpos** |
| **char Ypos** |

The piece will be a structure we define that will be used for keeping track of the location, type, and relevant data of each piece on the board. The Type, Color, and Moved values will help us in calculating the legal moves for any given piece, the Value will be important in our implementation of the AI, and the Xpos and Ypos will correspond to the indices in the ‘Board’ array at which the pointer to the piece is stored.

- **Move:**

| Move |
| --- |
| **char initX** |
| **char initY** |
| ***Piece Mover** |
| **char finX** |
| **char finY** |
| ***Piece Capture** |

The move will be a structure we define that will be a standardized method of representing potential moves in the chess game, which will allow us to compute legal moves and the legality of desired moves more easily. It will take the standard “A1 to B2” and represent it as:

| Move |
| --- |
| **char initX = 0** |
| **char initY = 0** |
| ***Piece Mover** |
| **char finX = 1** |
| **char finY = 1** |
| ***Piece Capture** |

Where the X and Y values correspond to indices in the ‘Board’ array, or squares on our chess board. The pointers will be the value of the pointers stored in the array at that location, which will point us to the piece at the beginning of the move (the one that the user desires to move) and the piece at the end of the move (be it the piece the user intends to capture, or the empty space to which they are moving the initial piece). This will allow us not only to standardize the format that we use to handle moves, but will also aid us in checking the legality of a move, as we will be able to access the properties of the start and end squares to check the basic requirements for a legal move. Furthermore, this format will be used in the generation of lists of legal moves, which will allow us to compare the user input to the list of all legal moves to check its legality, as well as generate lists for our AI to

choose from when making its move. In short, the move structure will be a standardized structure that we will use in the computation of our chess moves

- **MoveList:**

| MoveList |
| --- |
| ***Piece Mover** |
| **char length** |
| ***Entry First** |
| ***Entry Last** |

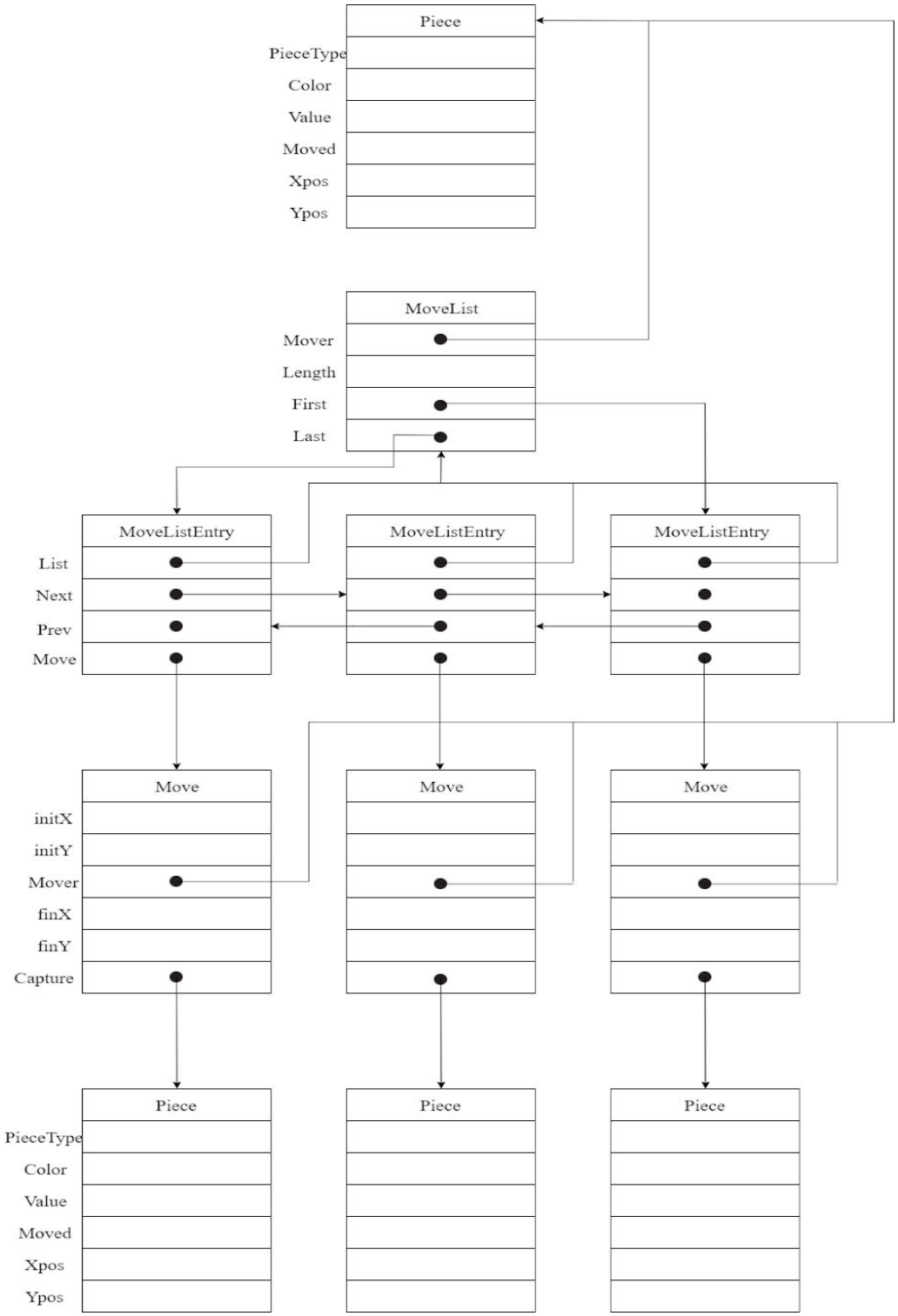
The MoveList will be the header for the double-linked list structure that we will be using to generate the list of all possible legal moves for a given piece. The list header is relatively standard for a double-linked list outside of the inclusion of the pointer to the piece that it is representing all of the possible moves for. This is more for our own utility further on in our algorithm implementation, as it will give us access to all of the relevant data for the piece should we need it. Outside of this we have a char that stores the length of the list for us as well as pointers to the first and last entries on the list.

- **MoveListEntry:**

| MoveListEntry |
| --- |
| ***MoveList List** |
| ***Entry Next** |
| ***Entry Prev** |
| ***Move Move** |

The move list entry will serve as a standard entry in the double-linked list that will store all legal moves for a given piece. Each entry will point back to the header structure of the list, it will also point to the next and previous entries in the list and finally point to the move structure that is a member of the list.

- **LegalMoveList Double-Linked List Structure:**



- **AliveList:**

| AliveList |
| --- |
| **char Length** |
| **char Color** |
| ***Entry First** |
| ***Entry Last** |

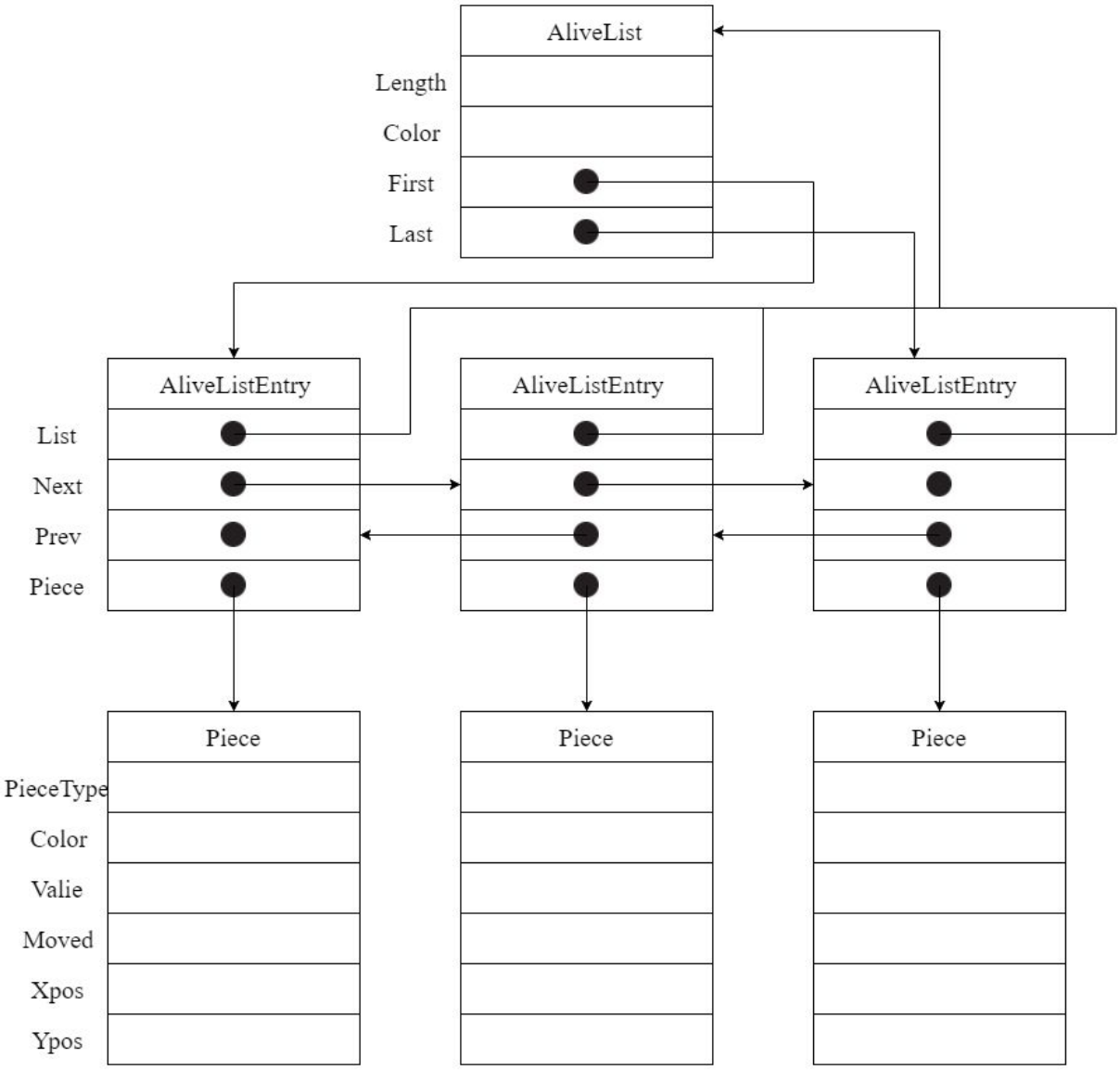
The AliveList will be a double-linked list structure used by the AI to keep track of it's alive pieces, and this list will be used in calculating its next move. The AliveList structure will be the header for the double linked list that keeps track of the Alive pieces, containing the length of the list, the color of the pieces it contains, and pointers to the first and last entries.

- **AliveListEntry**

| AliveListEntry |
| --- |
| ***List List** |
| ***Entry Next** |
| ***Entry Prev** |
| ***Piece Piece** |

The AliveListEntry will be the entry structure in our double-linked list that will allow for traversal of the list. It is implemented in standard double-linked list format, with a pointer to the list, the next entry, the previous entry, and the piece that it represents

- **AliveList Double-Linked List Structure:**




## 3.2 Functions and Parameters

Under ChessLayout.h:

- board InitBoard(void);
    - return board;
    - This function will create the board and initialize the pieces. It will be of type of void and return the board, an array.
- piece* CreatePiece(char PieceType, char color, char value, char moved, char xPos, char yPos);
    - return *piece;

- This function will create a piece type of our choosing. This will also be helpful for the special move “promotion”.  It will be of type void and will return a pointer to a piece.

Under Move.h:

- MoveList* CreateMoveList(*piece currentPiece);
  - return *MoveList
  - This function will take a piece and return its possible legal moves.
- void DeleteMoveList(*list possibleMoves);
  - return void;
  - This function will delete a piece’s move list.
- move* CreateMove(intY, intY, *piece Mover, finX, finY, *piece capture);
  - return *move;
  - This function will essentially serve as the function to make a piece move, and will return the position the piece moved to.
- void AppendMoveEntry(*MoveList possibleMoves, *Move legalMove);
  - return void;
  - This function will add a new possible legal move to a piece’s move list.

```
#ifndef MOVE_H
#define MOVE_H

typedef struct moveList moves;

struct moveList{

};

/* creates a set of moves */
int CreateMoveList(void);

/* deletes a set of moves */
int DeleteMoveList(void);

/*append a set of moves */
int appendMoveList(void);

/*delete a set of moves */
int deleteMoveList(void);
#endif
```

Under FileIO.h

- move StringToMove(char* string, board* board)
  - return move;

    - This function will take the users input, which is a string, and convert it to the move data type.
- char* MoveToString(move currentMove)
    - return string in form of char;
    - This function will take the AI's move, and convert it to a string.
- char* LogMove(string)
    - Return string;
    - This function takes a string in order to log the move(print the move onto the users screen).
- char* LogCheck(color)
    - return string;
    - This function takes a piece's color in order to log who has put who in check(print check onto the users screen).
- void LogMate(color)
    - return void;
    - This function will take a piece's color in order to log who has put who in checkmate(print checkmate onto the users screen).

Under Legal Move:

- int LegalMoves(move, board, check)
    - return (char legal)
    - This function will take in the pieces move, the current status of the board, and the current status of check in order to determine the possible legal moves for a piece.
- move GenLegalList(piece, board, check)
    - return *moveList;
    - This function will take in a piece, current status of board, and the current status of check in order to generate all the legal moves, and will be updated with every legal move as the game goes on.

Under Locomotion:

- void MovePiece(move, board)
    - return void;
    - This function will take the move parameter and the current status of the board in order to move the piece to another position on the board.

```
#ifndef LOCOMOTION_H
#define LOCOMOTION_H

#include "Move.h"

/* move piece function declaration */
void MovePiece(move, board);

#endif
```

Under EndGame:

- char CheckCheck(Board, turn)
    - return (char check);

Under Makefile:

- Responsible for generating .o files
- gcc will be ran through here
- File generator or clean up

## 3.3 Input and Output Formats

### 3.3.1 User Input Syntax and Format

**Initial Game Options**

Pending the implementation of a full GUI, the initial options for the game will be implemented via command line flags, which will allow the user to select which side they play as, player v player or player v AI mode, and the AI difficulty.

**Making a Move**

Pending the implementation of a full GUI, the user's moves will be implemented via the command line in the format "A2 B4" for example, where the initial square is given first and the final square is given second, the program will parse the move and throw and error if the input is incorrectly formatted or the proposed move is illegal

### 3.3.2 Log File Recordings

**Log File**

After every move, a C-string stream consisting of the moves (in the format of [‘moveNumber’ ‘start’ ‘end’ - ‘color’ ‘time’] will be output onto a log.txt file. If any special move is processed, or a check, or a checkmate occurs, we use special notation. The log file will create a new line per move and number the lines.

- Example entries include:
    - “1 A2 A4 - white pawn - 2s” to indicate a while pawn has moved two spaces, and the move lasted 30 seconds
    - “567 Black King E8 in check by white bishop C6 - 7s”
    - “859 Black King F8 in checkmate by white Queen D6 - 5s -Game Over! ”

# 4 Development Plan and Execution Timeline

## 4.1 Task Partitioning

| TASK | Bianca | Cesar | Joshua | Michael | Reza | Tun | Timeline |
|---|---|---|---|---|---|---|---|
| **AI** | | | x | | x | | Week 4 |
| **Chess Layout** | | x | | | | | Week 3 |
| **Endgame** | | | x | | x | x | Week 3 |
| **GUI** | x | | | x | | | Week 3 |
| **File I/O** | x | | | x | | | Week 3 |
| **Legal** | | | x | | x | x | Week 3 |
| **Locomotion** | x | x | | x | | | Week 3 |
| **Main** | x | x | | x | | | Week 3 |
| **Makefile** | x | x | x | x | x | x | Week 3 |
| **Move** | | | x | | x | x | Week 3 |

## 4.2 Team Member Responsibilities

We partitioned the tasks as shown in 4.1 whereas each team member is collaborating with another team member to accomplish each assigned task. In this way, we get the tasks done as efficient as possible. Timeline will start as early as week 2 of dev and will continue to be adjusted accordingly until week 5. Furthermore, each team member is expected to document their update on the servers via CVS, as well as work on the deliverables and changing them appropriately with respect to the time permitted and accomplished tasks.

# 5 Copyright

## 6 References


“Let’s Play Chess” Summary of the Moves in Chess. USCF

Terminology and definitions in C. Doemer, Rainer. Lecture. UCI. 2017-2018

## 7 Index