ZotChess

Software Architecture Specification Version 2.0.0

University of California, Irvine - EECS 22L SOFTWARE ENGINEERING PROJECT IN C LANGUAGE - Winter 2019

TEAM 5 - 4OUR

HE, JINGTIAN
RAMOS, ROY
TANG, JOHNATHAN
TRINH, BRYAN
ZHU, KEVIN
ZOU, ZIHAO

January 18, 2019

TABLE OF CONTENTS

GLOSSARY

2D Array:

Also called **2-dimensional array**. Similar to a 1D array, a 2D array is a collection of data, however it is organized as a matrix with rows and columns.

Define:

When something is defined, it means that it is created and exists here, and to make memory for it. It is similar to declarations and initialization, but differs from the former by having memory allocated and the latter by not having an initial value.

Enum:

Also called an **Enumeration**. Enum is a user defined data type in C, mainly used to assign names to integral constants - the names make a program easy to read and maintain.

Graphic User Interface:

Also called **GUI**. It is a form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators, instead of pure text-based user interfaces.

Hard-Coding:

Hard-Coding is the practice of fixing the data/parameters in a program such that it would run only that certain way, as opposed to obtaining data from other external sources at run-time.

Include:

A request to use a particular header file, written as "#include" in C. This ties in with header files as the file contains C declarations and macro definitions to be shared between multiple source files.

Initialization:

Initialization is the assignment of an initial value for a data object or variable. The manner in which initialization is performed depends on programming language, as well as type, storage class, etc., of an object to be initialized.

Macros:

A macro in computer science is a rule or pattern that specifies how a certain input sequence should be mapped to a replacement output sequence according to a defined procedure. This fragment of a code is given a name, and whenever the name is used, it is replaced by all of the contents of the macro. The 2 types of macros are object-like macros, which resemble data objects, and function-like macros, which resembles function calls.

Output:

An output is a collection of data that is generated by a computer, including data produced at a software level, such as a generated calculation, or a physical level, such as a printed document.

Pointer:

A pointer is a programming language object that stores the memory address of another value located in computer memory. A pointer references a location in memory, and obtaining the value stored at that location is known as dereferencing the pointer.

String:

A string is a sequence of characters as a lateral constant or in a form of a variable (that is able to be changed after creation)

User-Input:

Also simplified to just **Input**. User-Input is the collection of information or data sent to the computer for processing through the use of an input device. It's counterpart is **Output**.

1.0 Software Architecture Overview

1.1 Main Data Types and Structures

Enum

Enums are fundamentally integers. Enums are used to locate the files such as the A column in the board position A1, A4. Enums also increase code efficiency when compared to using Strings. This can be seen when white pawn is defined as an Enum with the value 1. This also makes the code easier to read. Instead of only comparing values of strings or using '==' to certain arbitrary integer, logic can be immediately check with arguments such as '== whiteP'.

2D Array

2D Array is the backbone of the program. As previously discussed in Enums, by creating pieces with Enums first, 2D Array of integers can be created, such as pieces. The value of 0 would be located to an empty square on the board. This data structure is chosen because a chess board is inflexibly 8x8 and would store the same amount of information. There is really no dynamic allocation necessary.

Structures

Only one struct is used for the program. It is used to create the 'Coordinate' struct. The reason behind this was because the grid enum positions can be embedded afterward(recall the locations in chess being represented as A1, C1, etc). By creating this struct, storing the coordinate values (x,y) becomes easy. The only difficulty is remember that when accessing a 2D array that it would become [y][x].

1.2 Major Software Components

2D Array 'Data Structure'

- An 8x8 integer 2D array that contains everything on the chessboard

Coordinate 'typedef Struct'

- Helps more effectively locate coordinates on the chess board

Piecetype 'typedef Enum'

- Assigns pieces integer values for better communication
- Also allows coder to type a specific name representing a white or black piece instead of remembering int values on the array

Classic Artificial Intelligence

- The core structure of the bots that human players play against
- Respond the human players' moves with certain moves that are created by the algorithm
- Potentially has different difficulty level due to the different algorithms that are used to calculate what the move should be

GUI 'Display'

- Accepts user input and communicates the input to the rest of the program
- GUI takes in the board so that it can update the locations of the pieces

The program itself contains all of the above and follows game logic refer to **Section 1.4** detailing the game logic and program flow. It will communicate with the GUI receiving and sending values to help update the display. Finally after the game is over, it will print out a text file with the game.

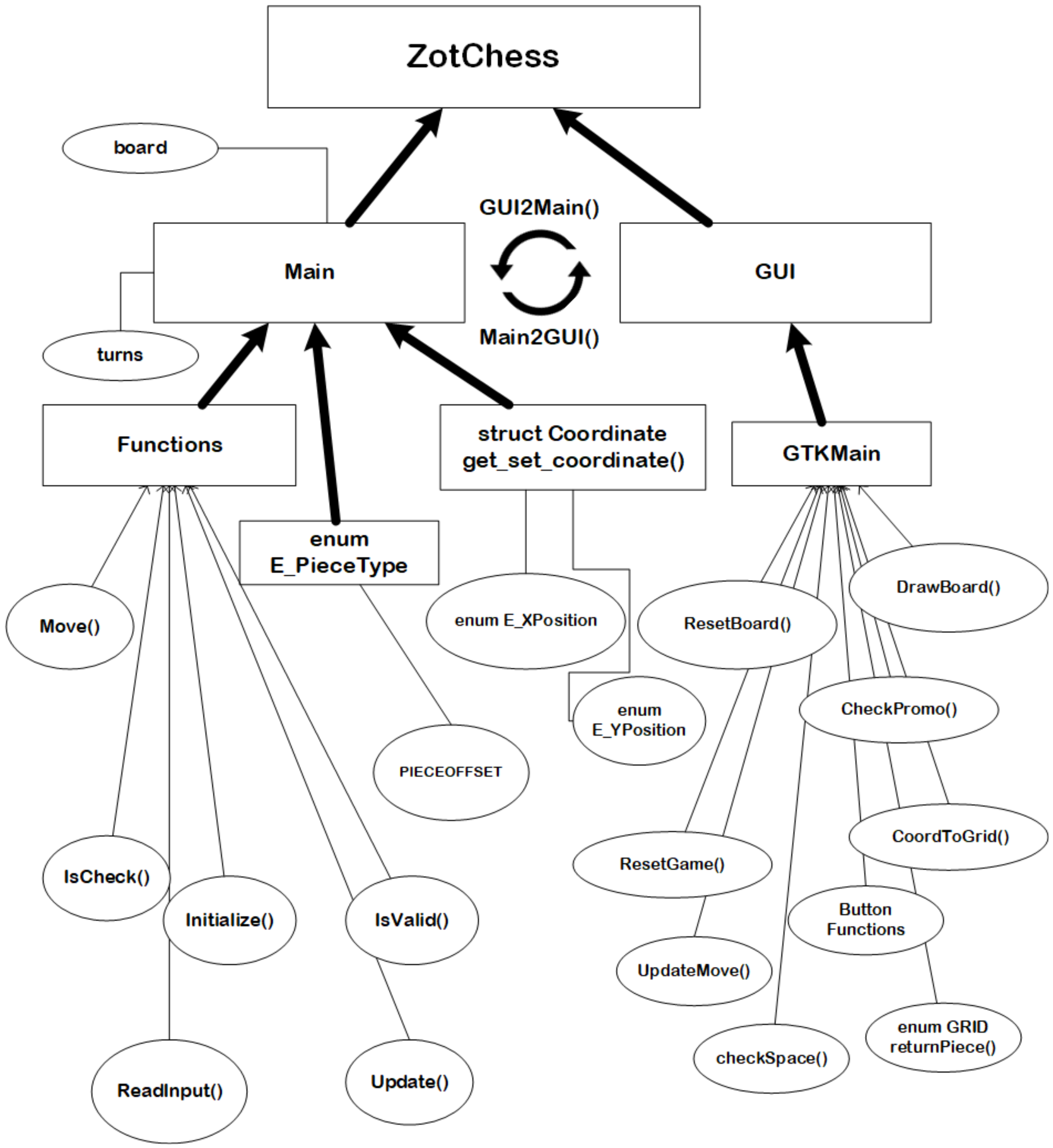


Figure 1:

Module Hierarchy of ZotChess

1.3 Module Interfaces

API of Major Module Functions

Initialize ()

- Input: board[][], game variable with the type of game the user makes, turns
- Output:
- Description: Takes the game board and game type as input. Depending on the type of game the user selects, the function will call the corresponding game type (Player vs Player or Player vs Computer) It can also loads a game with the selected pieces.

PrintBoard ()

- Input: board[][]
- Output:
- Description: When called, function prints out the 8 by 8 game board with the proper formatting. This hardcoded and is a placeholder for the GUI.

ReadInput ()

- Input: turn, board[][], currentmove
- Output: pointer with to currentmove
- Description: Reads the user input and returns a pointer with initial position, final position, and if necessary promotion.

Move()

- Input: pointer array to the currentmove, board[][]
- Output:
- Description: Move the piece the initial position targets to the desired final position. Function also updates the placement of pieces of game board. The move is modified also depending on some global flags which include castling and En Passant.

IsValid ()

- Input: turn, board[][], current move array pointer
- Output: an integer acting as a bitfield (0 = invalid, 1 or greater is valid)
- Description: Checks if the two positions the player is trying to move is valid. Also checks for En Passant and castling. This function hard codes the pieces movement.

IsValidCheck()

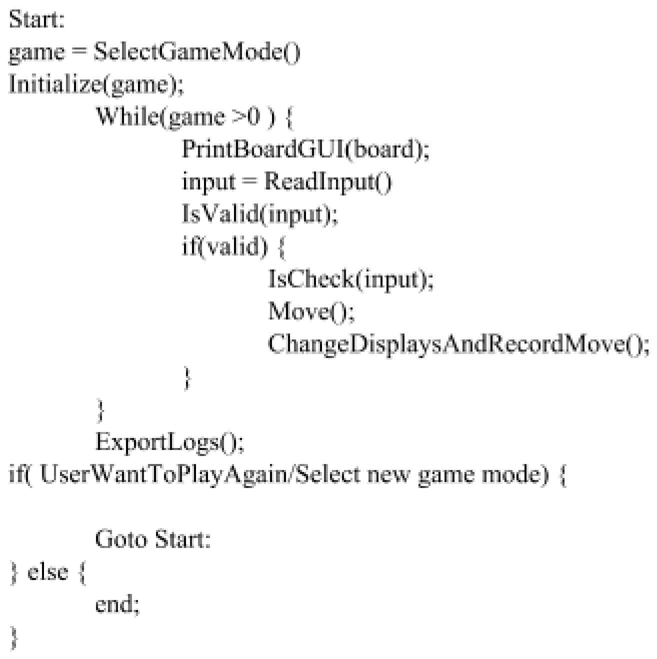
- Input: Turn, Board[][], Current Move Pointer
- Output: an integer that signals if player would still be in check (0 = valid, 1 = invalid)
- Description: When player's king is in check, function checks if the players next move would still result in a checked king, therefore invalid.

IsCheck/Checkmate()

- Input: board, current move pointer

- Output: an integer with 0 signaling not in check and 1 signaling check
- Description: Evaluate the game board and determines if there is a “check” or “checkmate” is in effect.

1.4 Overall Program Control Flow



```
Start:
game = SelectGameMode()
Initialize(game);
        While(game >0 ) {
                PrintBoardGUI(board);
                input = ReadInput()
                IsValid(input);
                if(valid) {
                        IsCheck(input);
                        Move();
                        ChangeDisplaysAndRecordMove();
                }
        }
        ExportLogs();
if( UserWantToPlayAgain/Select new game mode) {

        Goto Start:
} else {
        end;
}
```

Figure 2:

Pseudocode of overall program control flow

The ZotChess program welcomes its users with a title screen that asks for a game mode input. ZotChess offers two game modes: player versus player or player versus AI. After selecting the game mode, the game will then be initialized. ZotChess will then enter the main while loop. The while loop checks for whether there’s a game that is in progress; this is determined by a flag that is turned on if there’s a game. In the while loop, the first step is to print the GUI board. The GUI board will then ask for inputs and read the inputs. Such inputs are ran through an IsValid function that checks if the input is valid. If the IsValid returns false, the program returns to the top of the while loop. If the input is valid, the input goes through a Check function that checks for checkmates. Once the IsCheck function is finished, the Move function leads the GUI to update, a process that is detailed in the ChangeDisplaysAndRecordMove module. Once the program reaches the end of the if statement, the program would go back to the top of the while loop. When the game concludes, the movement logs are to be exported. The program will then ask the user if he/she would like to play another game using an if statement. If the user chooses to play a new game, the program will be prompted to start from the top of the flowchart. If the user enters no, the program will end.

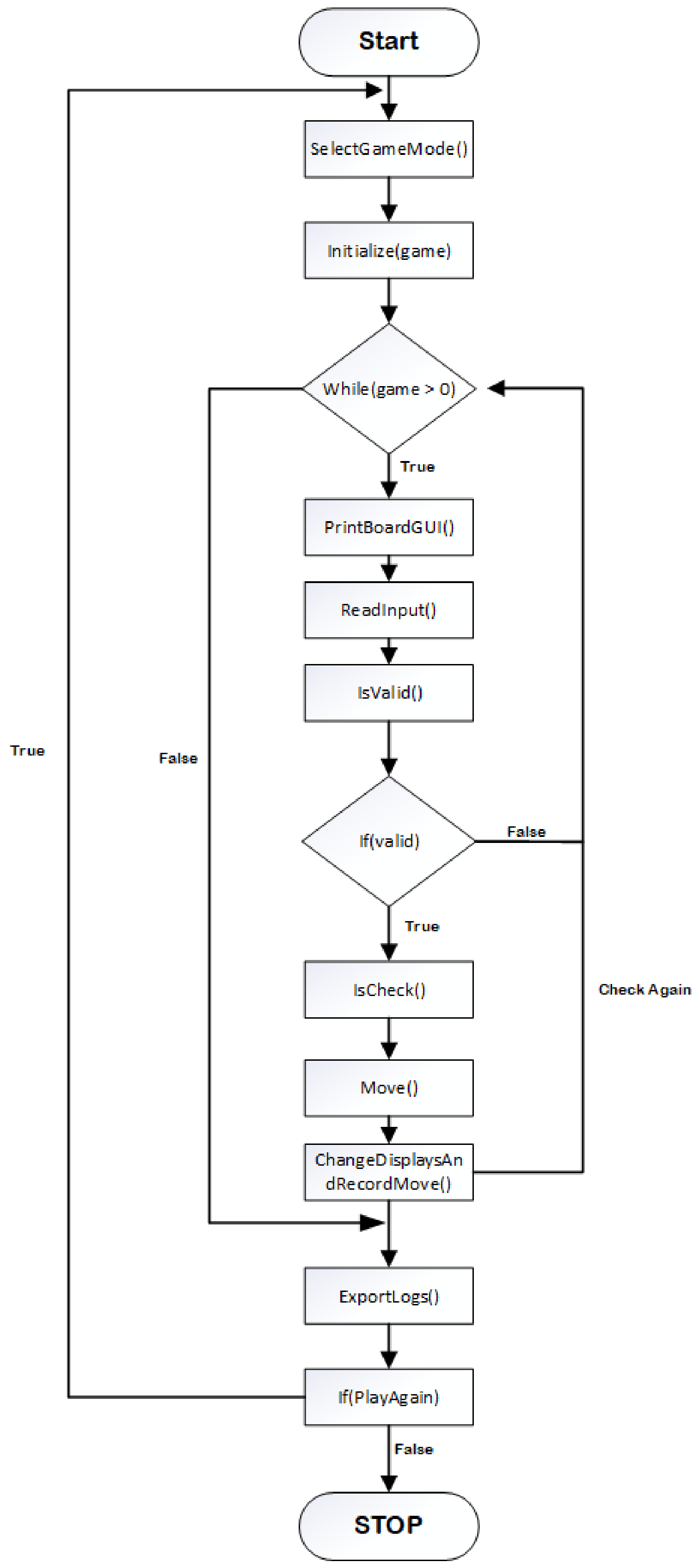


Figure 3:

Flow Chart of overall program control flow

2.0 Installation

2.1 System Requirements, Compatibility

ZotChess requires little processing power, yet succeeds to present a classic game that is meant to provide a fun-filled experience for its users. It does not only take 4 GB of Random Access Memory to run ZotChess, but it also only requires 4 GB of Hard Drive space. Either an Intel Core i3- 3210 3.2 GHz or an AMD A8-7699 APU 3.1 GHz is required alongside a Nvidia GeForce 700 Series or a AMD Radeon Rx 200 Series Graphics Processing Unit(GPU) in order for ZotChess to be launched. The machine must run at least a 32-bit GNU/Linux Operating System. However, a 64-bit version of the Linux Operating System is recommended. Nevertheless, ZotChess is built and designed to run flawlessly with such provided system specifications.

In order to achieve mind-boggling performance, it is recommended that the machine must run using an Intel Core i7-8700 3.2 GHz Central Processing Unit. Another recommendation is for the machine to have 16GB of DDR4-2666 RAM with the company of a Nvidia GeForce RTX 2080 GPU. Lastly, ZotChess' optimal performance is deemed best when saved in a Solid State Drive(SSD). In this case, A 480GB M.2 SSD would allow for the fastest boot up times.

2.2 Setup and Configuration

The installation process of ZotChess was designed to be as feasible as possible for its users. The first step is to download the tar.gz file. Once the downloading process has finished, untar the tar.gz file using an untar application. After following the provided steps, the user is able to view the contents of the tar.gz file and is ready for the compilation process.

2.3 Building, Compilation, Installation

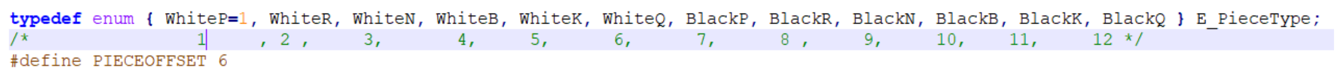
ZotChess' tar.gz file comes with everything that the user needs for the set up process. The developers of ZotChess has made it possible for the user to spend as little time as possible installing the program. The Makefile in the untarred folder allows for the compiling process to commence. By simply typing "make all" and pressing enter, the compiler through of the Linux system is prompted to compile the modules necessary. Once the compilation is successful, the installation process is done. Running the executable file would then start up ZotChess.

If the user chooses to uninstall ZotChess, the user must first close the application window. If necessary, the folder where the contents of the tar.gz file were unpacked is to be deleted. Otherwise, deleting the tar.gz file would finalize the uninstallation of ZotChess.

3.0 Documentation of Packages, Modules, Interfaces

3.1 Detailed Description of Data Structures

The primary data structure is a 2D int array. It holds integers that correspond to the E_PieceType. E_PieceType is an enum that makes it easier to address pieces. An enum is also an integer. This makes accessing the array easy, and flexible when compared to a the coordinates of a piece struct for example.



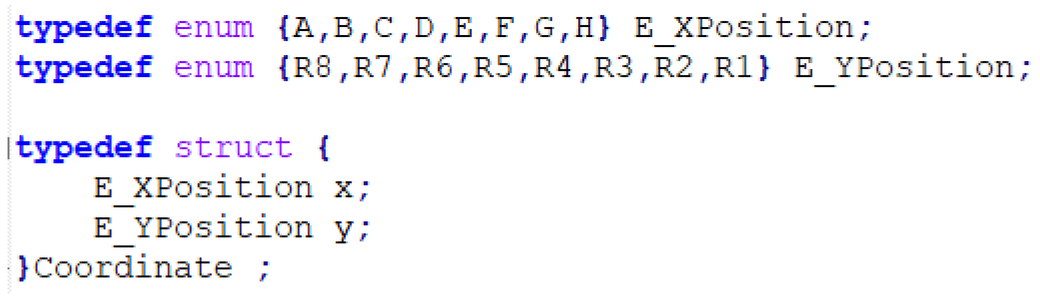
```
typedef enum { WhiteP=1, WhiteR, WhiteN, WhiteB, WhiteK, WhiteQ, BlackP, BlackR, BlackN, BlackB, BlackK, BlackQ } E_PieceType;
/*                 1      , 2 ,      3,        4,      5,       6,       7,       8 ,      9,      10,     11,      12 */
#define PIECEOFFSET 6
```

Figure 4:
Snippet of typedef enum for chess pieces. Note that the value of a WHITE piece plus 6 is equal to the value of the counterpart BLACK piece, and vice versa.

These are the values that used for E_PieceType. The macro, PIECEOFFSET, has a value of 6. It is the conversion between white and black. A white pawn has a value of 1, but a black pawn has a value of 7.

The 'Coordinate' class is the **coordinate system of the 2D array**. This class was written to be as portable as possible. Regardless of which side of the board was initialized, there needs to be coordinate conversion between the actual chessboard on the display and the actual array since arrays start at index 0 instead of 1.

A coordinate logically speaking has an x and a y coordinate which are both integers, which can help map 1 to 1 the space on the matrix. In order to make the conversion simpler, the coordinate class has two members 'x' and 'y', which are enums of 'E_XPosition' and 'E_YPosition' respectively. Because enums can be represented as integers, if it is easier to access the chessboard. Another and the better way to access the chessboard, especially if you play chess, is using the enum classes. The designers chose to pick the board in facing direction of white. The location A8 on the chessboard actually would be [0][0] on the array.



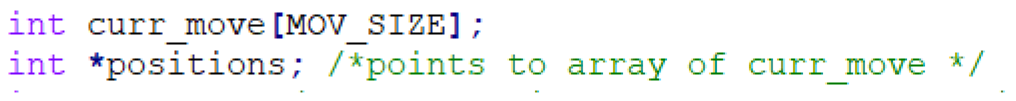
```
typedef enum {A,B,C,D,E,F,G,H} E_XPosition;
typedef enum {R8,R7,R6,R5,R4,R3,R2,R1} E_YPosition;

typedef struct {
    E_XPosition x;
    E_YPosition y;
}Coordinate ;
```

Figure 5:
Snippet of typedef struct Coordinate

To make this a lot easier, in the x-axis is a mimic of 'E_XPosition'. Note the 'A file' as it called in chess would represent the column 0 in the array and it is mapped as such. The '8th Rank' in chess would be the

0th row in the matrix and thus R8 = 0 as can be seen in the enum. Thus the coder wanted to represent the position e4 on the board. It can be as easy as [R4][E]. (Again the x and y values are switched in arrays be careful).



```
int curr_move[MOV_SIZE];
int *positions; /*points to array of curr_move */
```

Figure 6:
Snippet of curr_move to hold values of user input

Arrays were used to hold the x and y values of the user's input. This was a bandaid to prevent the use of global variables. Thus, the majority of the functions must know what the move is in order to function, and the pointer is essential for the majority of the functions. MOV_SIZE is the size of the move. As maybe if other vital information is needed, then it can be passed into the array. MOV_SIZE was through a define statement and can be changed or repurposed if necessary.

The log files were created using a linked list. A struct 'log' dynamically allocated using malloc(). These logs are combined together into one linked list that has a First, Last, and Length in order to create the log files. Each log has information regarding the piece that was moved, if that move was valid(note we used bits for IsValid read more about IsValid below and in section 1.3), and if the king was in check or checkmate. The logs can be printed by looping through the linked list starting with First and then traversing all the way to last. Along with the standard of just adding and deleting a log, an additional function was created to help speed up the initialization process of a log. This function, called create arguments, takes the aforementioned arguments and creates a new log and assigns all the values. As for the logfile list, it can append logs using the 'AppendLogFile' function. There is also code for DeleteLastLog in the logfile, but it is currently unused.

3.2 Detailed Description of Functions and Parameters

- **void** Initialize (**int** board[8][8], **int*** game, **int*** turn)
    - First looks at the game integer to help select a game mode. If the integer is less than 4, then a standard setup of a chess game will happen
    - A board with invalid values will be passed in. These invalid values will be changed to the values associated piece values associated with the chessboard.
    - The initial set up position is hard coded in for simplicity.
    - If the game mode is 4, it means that it will load a game. This part of the code parses a text file and then reads a series of numbers which represent the pieces due to how the enum class was defined. This gets translated over and put into the board. Additionally, the number of turns that has occured to get to this position can be used to change the turn variable
- **void** PrintBoard (**int** board[8][8])
    - When called, function prints out the 8 by 8 game board using 2-D arrays with updated pieces of the game and their coordinates.

  - This function uses a **for loop** to cycle through the board and then print out text according to its location on the chessboard. Empty squares (on the board with value 0) will just print spaces while everything else will print out their respective piece.
- **int*** ReadInput (**int** turn, **int** board[8][8], **int** curr_move[MOV_SIZE])
  - Reads the user input and returns a pointer with initial position, final position, and if necessary promotion.
  - The read input uses a scanf to read user input. This string comes as two chess coordinates concatenated by a space. It uses the string tokenizer (stok) to separate the input by spaces. It then converts the inputs so that the 2d board array can read it.
  - It then returns this pointer with the positions of the user's move documented as {initial x, initial y, final x, final y, promotion}. Promotion will be -1 if there is no promotion occuring. When promotion does happen, the pawn must reach the last rank, the GUI will pop up with a screen allowing the player to select which piece they want. The ReadInput() does not handle that.

- **void** Move(**int*** positions, **int** board[8][8])
  - Move the piece the initial position targets to the desired final position. Function also updates the placement of pieces of game board.
  - It uses the position of the initial and grabs the value of that piece. It stores the value in a temporary int. It then removes the initial position and places the piece on the final position in the board array.
- **int** isValid (**int** turn, **int** board[8][8], **int*** positions)
  - Check if user's move is legal based on the initial and final coordinates that the user inputs.
  - First determines which player is in turn, therefore the player cannot move the opposing player's piece. Since enum is used to define our pieces, White cannot access any pieces above 6 while Black cannot access pieces below 7. After, check which piece the user is targetting. For example, if the coordinate points to a pawn, the corresponding value would be either 1 or 7. Finally, proceed through a series of conditional statements to check if the relationship between [initial.x][initial.y] and [final.x][final.y] is valid based on the rules of the piece. If it is valid, function returns boolean value **1,** allowing the main function to increment to the next turn. However, if the move is not valid, the function returns **0,** which prompts the user that the move is invalid and to input another set of coordinates.

- **int** IsValidCheck (**int** turn, **int** board[8][8], **int** *positions)
  - Checks if users next move would still result in a checked king and therefore output an invalid flag.
  - To avoid contaminating the actual game-board, the function creates a temporary 8 by 8 2D array before transferring content from the game-board to temporary board. Next, function replicates player's move (tboard[final.y][final.x] = piece) on temporary board before passing the temporary board to the IsCheck function to determine if that move is invalid. If move is invalid, function returns 0 (false), else 1 (true).
- **int** CheckCheckmate (**int** board[8][8], **int*** positions)

  - Evaluate the game board and determines if there is a "check" or "checkmate" is in effect and returns 1 if Check and 0 if not in check
  - This function takes the final position of the piece where it moves and then calls the IsValid() and sees if it aligns with the king's position
  - Additionally, discovered checks exist. It needs to check all the bishops, rooks, queen, and rooks to see if there was a discovered attack. They also need to call the isValid() function, using a for loop to loop through the board.
  - Finally, if there is check, then it will check for checkmate.
  - It will use a **for loop** to loop through the possibilities and will break upon seeing a valid move
  - If a capture is possible then it needs to check for isCheck again to make sure that that piece is not protected.
  - If it runs out of all possible squares for the king and there is no valid move, then it should return 2 indicating checkmate, 1 if there is check and 0 if there is no check.
- **Void** PrintLogs (**LogFile** *lf, **int** turn)
  - Takes the pointer to the logs and the number of turns in the game and appends , "w", a text file called 'Logs.txt'
  - It will loop through the log file, correctly interpreting the piece (using the enum), then grab the final position from the array and see if it needs to add a 'x' for capture '+', 'o-o' or 'o-o-o' for castling '#' for checkmate, etc.
  - It does this by creating a string and concatenating the different elements together,finally using memcpy() to produce the desired result.
  - PrintLogs takes in turns because of the complexity of loading a game. If a game starts at move 5, the log file will show that the game started on the 5th move as opposed to turn 1.
- **void** SaveLogs(**LogFile** *lf, **int** board[8][8], **int** turn,**int** check**,int** valid,**int*** positions)
  - Save Logs takes in the logfile, the board, number of turns, check, if the move is valid, and the positions and does not return anything
  - It simply goes to the positions and finds the piece. This information is needed in log.
  - It takes the LogFile and appends a log with all the information above.

3.3 Detailed Description of Input and Output Formats

The final version of the product obtains user input from clicking on the GUI. The user input will send will be sent to the GUI files. These files will process the input. Here the interactions will be defined as **GUI to GUI** or **GUI to Board**.

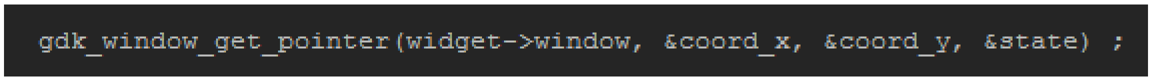
GUI to Board is defined as the interactions when the user clicks on the board part of the GUI. GUI to GUI are all the other interactions that should be handled in the GUI including but not limited to clicking a button, starting a new game, and scrolling through the moves recorded.

When the user makes two inputs on the board, these coordinates will be converted into coordinates on the chessboard. A coordinate (321, 1093) maybe converted to e2 because that is the position on the chessboard. The two inputs represent the two positions that the user will be inputting. The program will then concatenate the two inputs together into a string such as 'e2 e4' with a space in between. In case of promotion, the input would be 'e2 e4 q'. This input will then be handled by the board.

The board handles the inputs and will receive the input from above. The code is written in C and uses a string tokenizer (stok in the string class) to parse the elements. These things will be converted into readable coordinates for the 2D array. The ReadInput function then returns a pointer with the first coordinate and second coordinate.

After checking if the move is valid and if there is check or not from the move, the reading log file function will take these inputs in as well as the array from reading the input. It will find the piece and then store the move in standard chess notation. The initial position is dropped because that is not standard chess notation. A brief explanation through an example is the pawn moving from e2 to e4. The program will store e4 and not e2 or E4. At the end of the game the log will automatically be saved as a text file called ‘Logs.txt’. The moves will be logged with the move number followed by White and then Black’s move. In case if the game was imported, it will start at the specified turn. If for whatever reason, its black’s move, white’s move will be logged as ‘...’ (standard chess notation).

gdk_window_get_pointer(widget->window, &coord_x, &coord_y, &state) ;

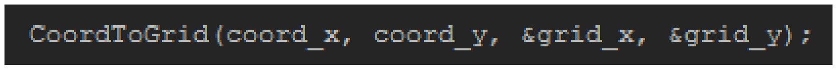


```
gdk_window_get_pointer(widget->window, &coord_x, &coord_y, &state) ;
```

Figure 7:
Snippet of function to get pointer’s current position

Where *widget* is a GtkWidget struct with a member called window, *coord_x* and *coord_y* are ints and *state* is GdkModifierType. This function takes in the pointer’s current position and modifier state through the GDK window and stores the x and y values into *coord_x* and *coord_y*, respectively. It also stores the modifier mask into *state*. This function is critical in the implementation of the GUI since it allows the selection of a certain chess piece and move it.

CoordToGrid(coord_x, coord_y, &grid_x, &grid_y);

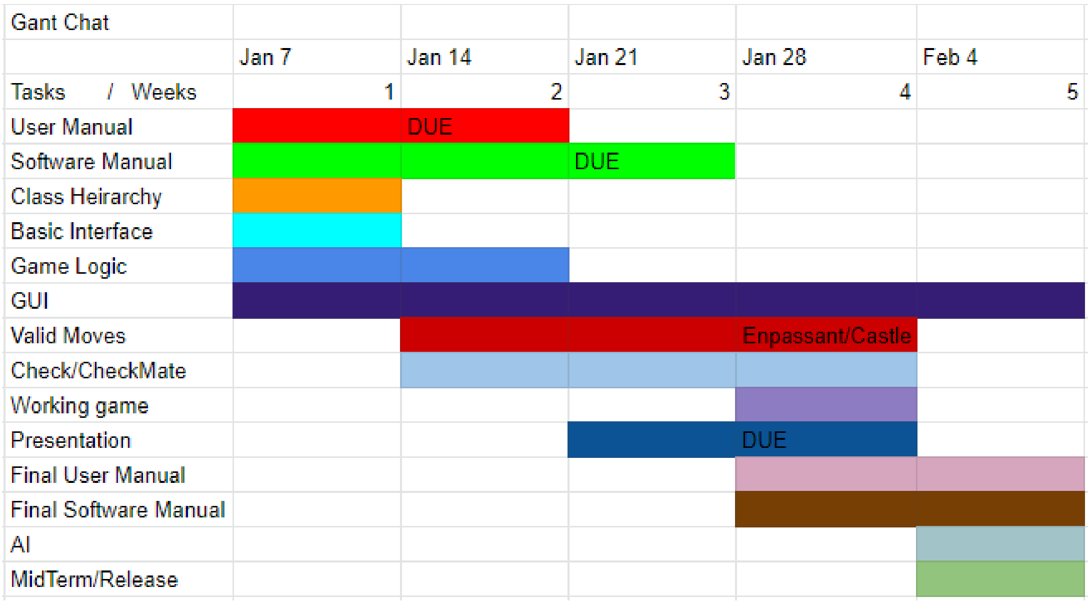


```
CoordToGrid(coord_x, coord_y, &grid_x, &grid_y);
```

Figure 8:
Snippet of function to map position to grid

This function takes in the previously set values (coord_x and coord_y) and converts the values using an algorithm to make the pair into grid values that are used by the chess board. Thus, the output for the most important function of the GUI is used as input for translation of the values into information that can be easily accessed to map the chess board and its pieces.

4.0 Development Plan and Timeline



Gant Chat					
	Jan 7	Jan 14	Jan 21	Jan 28	Feb 4
Tasks / Weeks	1	2	3	4	5
User Manual	🟥	🟥 DUE			
Software Manual	🟩	🟩	🟩 DUE		
Class Heirarchy	🟧				
Basic Interface	🟦				
Game Logic	🟦	🟦			
GUI	🟪	🟪	🟪	🟪	🟪
Valid Moves		🟥	🟥	🟥 Enpassant/Castle	
Check/CheckMate		🟦	🟦	🟦	
Working game				🟪	
Presentation			🟦	🟦 DUE	
Final User Manual				🟪	🟪
Final Software Manual				🟫	🟫
AI					🟦
MidTerm/Release					🟩

Figure 9:
Gantt chart, dividing up the
days to work on certain aspects

4.1 Partitioning of Tasks

Graphical User Interface: Bryan, Roy
Valid Moves: Johnathan
Implementation of Checks: Jingtian, Zihao (main)
Lead AI Coder: Jingtian
Data Structures, Input, Printing Logs: Kevin

Each of the six developers are contributors to the User Manual, Software Manual and Artificial Intelligence aspect of the project.

4.2 Team Member Responsibilities

Manager: Kevin Zhu
Recorder: Johnathan Tang
Presenter: Bryan Trinh - *Documentation Formatter*
Reflector: Jingtian He - *Submission*
Reflector: Roy Ramos - *GitHub Manager*
Reflector: Zihao Zou

5.0 Copyright

University of California, Irvine
The Henry Samueli School of Engineering
EECS 22L - SOFTWARE ENGINEERING PROJECT IN C LANGUAGE
E4130 Engineering Gateway, Irvine, CA 92697
+1 (949) 824-5333

6.0 References

1. Walia, Dipesh. “Data Structures in C Programming Language.” *Conio.h | Programming Simplified*, Creative Commons Attribution , www.programmingsimplified.com/c/data-structures.

2. Chaudhary, Shashank. “Artificial Intelligence 101: How to Get Started.” *HackerEarth Blog*, 25 Sept. 2018, www.hackerearth.com/blog/artificial-intelligence/artificial-intelligence-101-how-to-get-started/.

3. Gtk. “The GTK+ Project.” *About GTK+*, The GTK+ Team, www.gtk.org/.

4. “Getting Started with GTK+.” *GNOME Human Interface Guidelines*, The GNOME Project, developer.gnome.org/gtk3/stable/gtk-getting-started.html.

5. Gale, Tony, and Ian Main. “GTK+ 2.0 Tutorial .” *GNOME Human Interface Guidelines*, developer.gnome.org/gtk-tutorial/stable/.

7.0 Index