Alex Simak
CMSC 421
Due: May 3, 2020

Chess Implementation

I first started this chess program on April 16, where I started with the initializations of my files. I followed the tutorial provided by John Boutsikas, the CMSC 421 teacher assistant, where he showed us how to initialize the main.c, Makefile, chess.h and chess_functions.c. After this tutorial I had the chess_open, chess_read, chess_write, and chess_release initialized. I have some global variables that I use through every function such as user_input, the chessboard, globalbuffer, the length of the global buffer, the colors of the players, the turn tracker and the game tracker to see if the game is still in a playable state. I started implementing the chess_read in order to send the buffer to the user. I have a global buffer to which I alter the contents for each write operation required.

The read checks for the global bufferLength to be true (anything other than 0) (to make sure read is not being called multiple times), and if it is, it sends the buffer to the user using copy_to_user. I then empty the user_input array to all ‘0’ for it to be ready for the next input by the user. I then return the length of the globalbufferLen so that the user can use that data to keep the game flowing.

I then worked on my write which basically holds the entire game play of this chess implementation. I started with getting the user input, my user input can hold a potential of 18 chars, but the max the user needs to send is 15 for the gameplay so I store a bit of an extra space if there is a typing issue or error by the user input. The first step to get the user input was the use the get_user operation where I acquired the input using a for loop. The write function has a parameter which sends in the length of the user input, so I used that to access the length of how many chars are sent. The user_input array now holds the users input for every operation whenever the user sends in data. The next operation was to store the necessary input in an array called menu_choice which consists of the user picking either 00, 01, 02, 03 or 04. The menu choice just takes the first two inputs of user inputs and stores that as a variable for easy access and comparison between strings. I then have a series of if statements which check for the menu choice.

The first one of these 5 that I started working with was 00 which begins a new game and resets the board to the original state. Whenever the 00 operation is sent in, the gameOn global variable is set to true, which then I set all of the board contents initially to "**" which is an empty space on the board. Then I started placing the pieces into the board using my 2D array chessboard. I started with setting the white pieces in the designated spaces, starting with the pawns on the [1][y] coordinate and then the rooks, knights, bishops, queen and king. Then I set up the same thing for the black pieces. Afterwards I dealt with the argument sent in with the 00 choice. The 00-menu choice only takes one argument, which is what color the user wants to play as. I check if the user inputs any other characters after and if so, I send the invalid format "INVFMT" command into the buffer, but if the user just sends the W or B command with the 00, such as this "00 W" it sets the game board up then applies the W color to my global variable called userColor. Depending on the user choice, I then apply the cpuColor global variable with the opposite color using a simple if and else statement. If all is good with the input, it responds with the "OK" command to the user to move onto the next step.

The second piece I worked on was the 01-menu choice. This returns the state of the board to the user so that the user can easily play the game. First, I make sure that the game has been initialized. Then if it has been started, I set each char of the chessboard into my globalBuffer array to which it is then sent to the chess_read function and sent to the user as described earlier in this documentation.

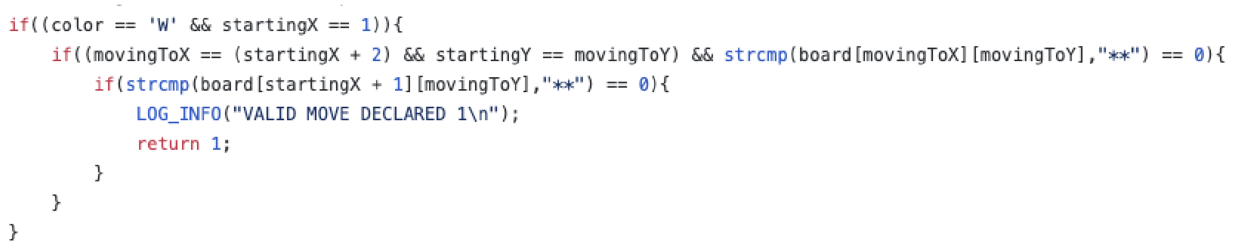
I then moved onto my 02 step which is a very large portion of this project. I started this with some input checking making sure that the user does not input junk or make sure that the format is correct. The user input for their turn should look like this – “02 WPe2-e4” which means user turn, white pawn moves from e2-e4. I check if there is no space in between the 2 and W, and if not, I send the invalid format command, or if the elements are missing from this basic move command it also sends the invalid format. I then start with the actual movement portion of the 02-menu choice. The first thing I check if whether the game is being played, and if so I move on and if not I send the “NOGAME” command through to my globalbuffer. Then I check if the user is supposed to be going, aka it is the players turn. I use the modulus operator for this, if it is an even number it is the players turn and if it is odd it is the cpu’s turn. After each turn I increase the playerTurn by 1 to keep track of the turns. If a wrong turn is tried, it sends the out of turn, “OOT”, command.

After those basic validations, I start looking at the user input and check if the user is trying to access the correct color. Making sure that if they are W, they are trying to move a WP for example and not a BP. If they are trying to move the wrong piece, it sends the invalid move command. Then I start converting the board letters to be the actual 2D board operations. For example the E2 board location equals the [1,4] on the 2D chessboard array. I implemented a function called convertLetterToNum() which takes in either a letter or a number from the input, (E2) and converts the E to a 4 and the 2 to a 1 to make the [1,4]. Then I look at the board at that specified location and say at chessboard[1][4] there is a WP, I then validate the attempted move. I make sure that the user is actually trying to move a WP, if they type 02 WNe2-e4 but in e2 there is a WP, it will send an invalid command. If they selected the correct piece to move, I move on to the next checking.

I do the same thing with the moving piece and grab the location of where they are trying to move to. For example, the e2-e4 will be [1,4] → [3,4]. After this, I make sure the user is not trying to move their piece to the same location, which does not count as a move in chess. I then started working on the validation of moves. I wrote a function that checks all the moves for each specific piece.

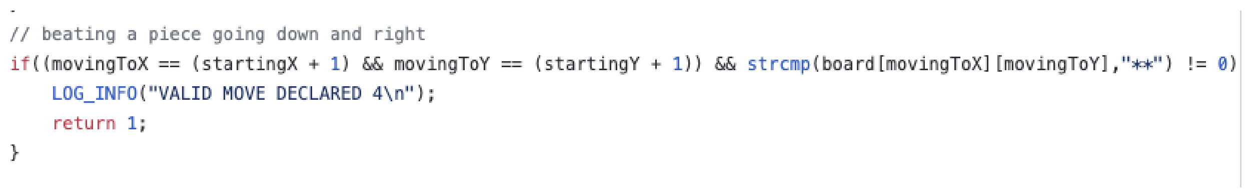
```
int validMove(char color, char piece, int movingToX,int movingToY, int startingX, int
startingY, char *board[8][8]){}
```

ValidMove: This function takes in the color of the specified player, the piece selected, the place they are moving and the place they come from, and the board. The 'board' parameter lies for temporary board association which has not been mentioned in this documentation yet. I check whether the move is to the same place, then I return 0 as not a valid move. Then I start checking for the pieces. I started simple with the **pawns**. I check if it is a pawn and if it is the correct color piece (CPU purposes), then it can only move forward one space, or two if it is coming from the starting location. If the pawn is white, it can only move forward two spaces when it is sitting on the [1][x] location and if it is a black pawn it can only move up two if it is sitting on the [6][x] location. So, I check for the movement of two spaces forward first, and it can only move forward two spaces forward if there is no other piece blocking the move.



```
if((color == 'W' && startingX == 1)){
    if((movingToX == (startingX + 2) && startingY == movingToY) && strcmp(board[movingToX][movingToY],"**") == 0){
        if(strcmp(board[startingX + 1][movingToY],"**") == 0){
            LOG_INFO("VALID MOVE DECLARED 1\n");
            return 1;
        }
    }
}
```

Pawns can only beat other pieces in a one diagonal forward move. So, if there is no other piece in front of it, it may move forward to spaces. I then check if the user specified to move 1 space forward, and again I check if it is empty using the strcmp function and if so, it is a valid move to which I return 1. Then I check for the beating aspect. Since I have the original locations of the board, and where I am trying to move, I can see if there is a piece and whether or not I can move there. For example, I am the WP trying to beat the BP that is sitting to the right diagonal from me, I check if the moving X location is my starting X location + 1, and my moving Y location is equal to the startingY + 1, then it is a valid move and I can beat the piece where I return 1. I use the strcmp operation to check if two strings are equal to one another. If they are equal, the strcmp function returns 0 and if not, it returns a nonzero number.



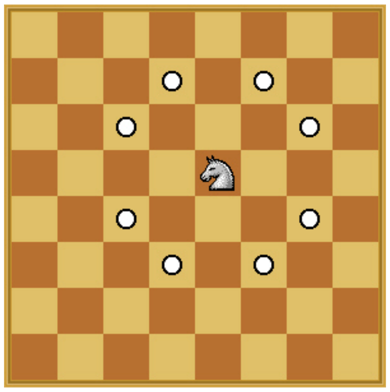
```
// beating a piece going down and right
if((movingToX == (startingX + 1) && movingToY == (startingY + 1)) && strcmp(board[movingToX][movingToY],"**") != 0)
    LOG_INFO("VALID MOVE DECLARED 4\n");
    return 1;
}
```

I have the white and black pawn operations separated because it depends on where the pawn lies to determine where the forward move is. The black piece starts on 6 so going forward requires me to subtract from the starting location, but the white piece I need to add to the starting 1 location.

After the pawns I started working on **rook**. Rook was a pretty easy one to figure out because it can only move vertically and horizontally but it can move forwards and backwards or left or right, so it does not matter which color is going. So, I start of the same as pawn and I check if the user is trying to move the R piece and it is their correct color (CPU purposes). For the rooks I have a simple way of checking, so if the starting X is equal to the moving X, that means the player is trying to move horizontally, and vice versa for the starting Y and moving Y. I then, in a for loop go from the starting location + 1, looking at the path of the rook, until the designated moving location -1, to see if those spaces are empty. If it finds that one of the pieces in between the starting move and designated end move, it returns it is not a valid move, but If it does not find a piece it returns that it is a valid move. This code segment shows the horizontal movement to the right.

```
if(startingX == movingToX){
    for(i = startingY + 1; i < movingToY; i++){
        if(strcmp(board[startingX][i],"**") != 0){
            return 0;
        }
    }
```

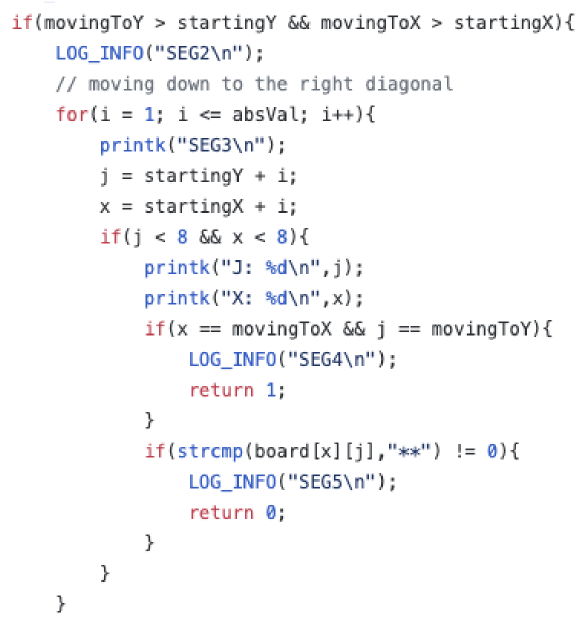
Following the rook, I worked on the **knight** which was also pretty easy because basically the knight can jump around. I just had a series of if statements checking if the user is trying to move in the correct locations. For example, if the knight is sitting in the [3,4] position, such as in the picture below, it can move to those white spaces for example [2,6]. I did this by just checking if the moving location X is equal to the starting location – 1 X and the moving Y location is equal to the starting Y + 2.



0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7

```
if(movingToX == (startingX - 1) && movingToY == (startingY + 2)){
    return 1;
}
```

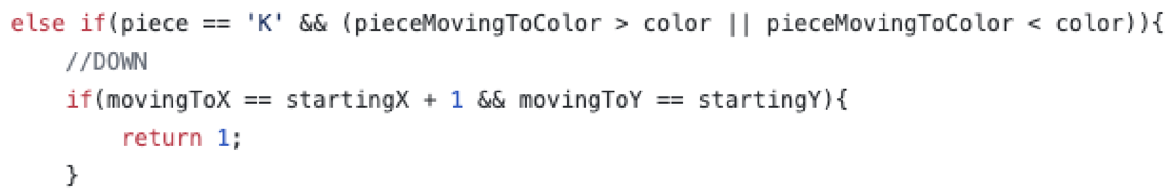
Then I moved onto **bishop** which was a little more complicated. This required a series of if statements as well, but I had special conditions to shorten the number of loops needed to be checked. For example if the bishop is in its starting location [0,2] and it wants to move to [3,5] I have a check looking if the movingX > startingX and if movingY > startingY this would mean the player is trying to move their bishop diagonally to the right and down the board. I then had a for loop running from 1 to the absvalue of the distance. I calculated the absolute value just by looking at the movingX – startingX then if it was negative, I multiplied it by a negative 1. So I incremented the board[i+1][j+1] elements until they either equaled the desired location, (which would return a valid move), until they one of the positions hit another player (which would return as 0, not a valid move) or if the I and j were out of bounds on the board, and in this case because we are incrementing both positions up, I am checking to make sure they are both lower than 8.



```
if(movingToY > startingY && movingToX > startingX){
    LOG_INFO("SEG2\n");
    // moving down to the right diagonal
    for(i = 1; i <= absVal; i++){
        printk("SEG3\n");
        j = startingY + i;
        x = startingX + i;
        if(j < 8 && x < 8){
            printk("J: %d\n",j);
            printk("X: %d\n",x);
            if(x == movingToX && j == movingToY){
                LOG_INFO("SEG4\n");
                return 1;
            }
            if(strcmp(board[x][j],"**") != 0){
                LOG_INFO("SEG5\n");
                return 0;
            }
        }
    }
```

After the bishop was finished, I worked on the **queen**’s movements. This was pretty easy because I was able to combine the elements of rook and bishop (aka just simply copy and paste) to make all the movements of the queen, so queen took less than 2 minutes.

Lastly, I added the **Kings** moves. These were also pretty simple because the king only has 8 possible moves to attempt and all the moves only require moving one space in a certain direction. I did this the same way I did pawns such as checking the moving coordinates with the starting coordinates. If one of the elements lined up with the if statements, it will be viewed as a valid move. For example, this picture below shows the kings ability to move down the board.



```
else if(piece == 'K' && (pieceMovingToColor > color || pieceMovingToColor < color)){
    //DOWN
    if(movingToX == startingX + 1 && movingToY == startingY){
        return 1;
    }
```

This concludes the validMove function and I moved back to the 02-menu choice where I left off. I am at the portion where I received the input from the user and I have where they want to go and where the original location is, so I pass those values through to the valid move function and if the move is valid I then check for the next components. If the user tries to beat another piece, they will use the x command, if the pawn is being upgraded to another piece, they will use the y, if the user is beating another player and being upgraded, then the user uses the x then y and lastly if the user just wants to go, they just write the input of the move.

I first started with the simple case of the player trying to make the move from one space to another using the default move format. I check if the space is empty where they want to move and then make sure that the user did not input an x in the position to beat any piece because the spot is empty. I also double check that it is not a pawn trying to move into the 7th or 0th position on the board without a y variable being inputted with it, and if so I send an invalid move command to the user, but if valid I swap the board locations and count it as a valid move and return OK to the user. The next step was to check if the user is trying to beat another piece to which they would need to input an x in the 10th spot of the user_input array. I make sure that there is a piece in the place they are trying to move, then I also check if it a pawn again, to make sure that the input has a y in it, if all is valid an OK is sent through to the user. Next I check if the pawn has entered the end rows of the board, without beating a piece. With a complicated if statement, I make sure that the user_input[10] is equal to y, (meaning they are ready to swap the pawn for another piece) and they are applying the change to a correct piece. For example, if I have a white pawn and I am entering the last row on the board, I need to swap to a W(piece) making sure it is the right color, then I make sure that they swap to a correct piece, such as a pawn, king, rook, etc. If so, I swap the position and apply the change to the pawn.

Lastly, I implemented the portion where the pawn is beating the opponent's piece and swapping the pawn for another piece. I had a similar composition to this portion with a complicated if statement to check where the x and y portions are of the user_input making sure that the 10th input was an X, the next two chars equaled the position on the board where the player is trying to move, which I used a strcmp for, and then I checked to make sure that the pawn is being swapped correctly similarly like I did in the simple implementation of the portion where the pawn swaps to another piece without beating the opponent. If all is valid, it then swaps and beats the player. Now if none of these are valid moves, then the illmove command is sent over to the user.

After making the move, I then check if the player put their piece in a possible check position and if so, it swaps the pieces back to the original board and sents the illmove command to the user. For this to be simple, I made a function called isCheck which just checks if the king is in check.

```
int isCheck(char color, char *board[8][8]){}
```

isCheck: In this function, I start off by checking which color is being imported in order to find the opposite color. I do this with an if statement. After I find the opposite color, I then loop through the designated board sent in with the parameter, and I look for the king position, then when the king is found, I save the coordinates in an x and y kingposition variable. After the loops finds the kings position, I have another loop that checks each of the possible moves of the players color pieces. I send the coordinates of the piece and then the coordinates of the king's position and if that piece can reach the king, it therefore counts as a simple check to the king. If a valid piece is found, it then returns 1 showing that a valid check has been found.

Going back to the user move and the 02-menu choice, if the user sets himself in check it counts as an invalid move and the user is set to go again. After I check if the user puts himself in check, I also check if the user is putting the computers king in check using the same function in check, but I use the userColor instead of the CPU because it looks for the opposite color. I check whether the CPU's king is in check and if it is, I check whether it results in a check mate. For this I made an isMate function.

```
int isMate(char color){}
```

isMate: In this function, I first start to check whether the king can possibly move out of a check position. I do this by in a loop – until the move is valid --, I start off by checking if the king is a check position, and if so I try to move him in all the variations that can be moved. If he cannot move (I try to have him move 10 times to make sure that the king is tried to move multiple times) then I try to have another piece block the check from occurring. I have the while loop select a random piece to try to move. I do this by at first, I set a random variable equal from 0 to the number of 6 (number of pieces), then from that I find the number of pieces of that type are on the board, for example there would be 5 pawns and it will then find a random number from 1 – 5 and select that pawn to move. I keep track of that specific pawns coordinates and then I have those as the original location of the move, then the placement of the move is also found by finding 2 random positions from 0 – 7 and see if that is a valid move. If not the loop repeats until a valid move is found that blocks the check. I let the loop run for a while, but if no moves are found then it counts it as a checkmate.

If a checkmate is found, then I send the message MATE through, but if only check then it sends CHECK through to the user. If it is not a check or mate and it is a valid move, the OK is sent to the user through the global buffer.

Now moving onto the 03-menu choice. In this the cpu makes its valid move, if possible. I start off with making sure no arguments are sent in with the 03 command, checking to make sure all the user_input after the first two commands is empty, and if invalid it sends the invalid format command. Then I move on with checking whether the game is occurring and whether it is the correct turn, the same way I check in the 02-menu portion described earlier in this document. Then I start with the random movement of the CPU. The movement is done very similarly as in the isMate function. I first start off with a loop – until a valid move -- which finds a random piece from 0-6 then see if it can be moved and if it is a valid position it is trying to move to. If it is valid, it makes the move and continues to the operation where it makes the move, if it is an invalid move, the loop is run again until a valid move is made. If the move is to an empty space, and not a pawn and in the 7th or 0th index, then it makes the move and sends OK. The next thing it looks for is if it a pawn and in the 7th or 0th position of the board. It then randomly swaps the designated pawn by picking a random number 0-6 and assigns it a new piece for that board index. It also checks for check and checkmate like in the 02-menu choice where the user makes their move. If there is a check, or mate, it will return CHECK and MATE respectively and if none of the above, it will return OK to the user.

Lastly, I worked on the 04-menu option. This option is where the user resigns the game and counts the CPU as the winner. Here I just did the basic operations where it checks to make sure the game is playing, if it is the users turn, and if there are any other arguments after the 04. All of these commands I already described in this document. Throughout the read and write operations I implemented mutex locking and unlocking making sure that only one process can access the read and write operations at one time.