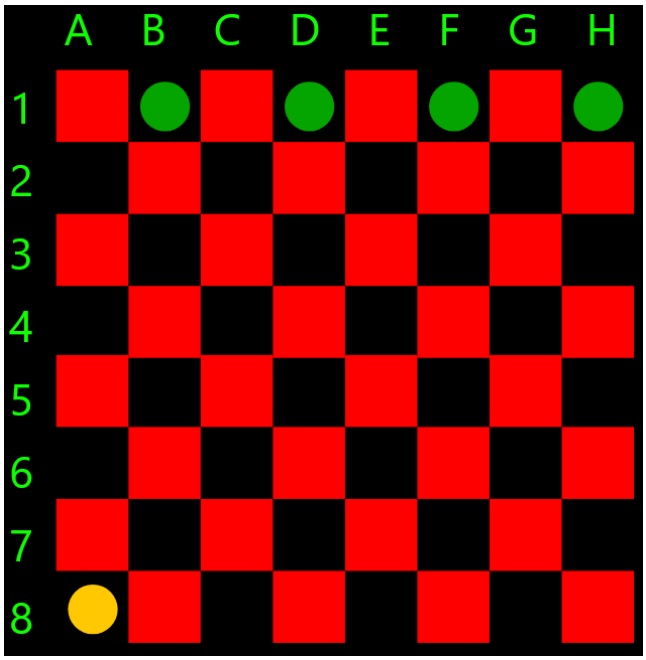
Fox and Hounds

Information on file naming, code location, and execution process is located under the Development Process, Tools heading.

Defining and Understanding the Problem

Statement of Intent

To design a command-line-based game in the C programming language, to emulate the *"Fox and Hounds"* board game. This game is played on a checker/chess board (8 x 8 checkered grid) with two players, one playing as the fox, the other playing as the hounds, as in the example board below.



The game begins with the fox moving diagonally in any direction, to another black square. The player playing as the hounds may then move any one piece diagonally to any black square. The hounds may only move in the forwards direction. The aim of the game as the fox is to escape to the other end of the board and the goal of the hounds is to stop the fox from doing so by trapping it.

Functionality Requirement

When the executable for the program is run, the user is asked whether they would like to play against an AI, or in a local two player game. The ai may then be set to a high or a low level of difficulty, written as “Rubbish” or “Overpowered”. *Player 1* then starts.

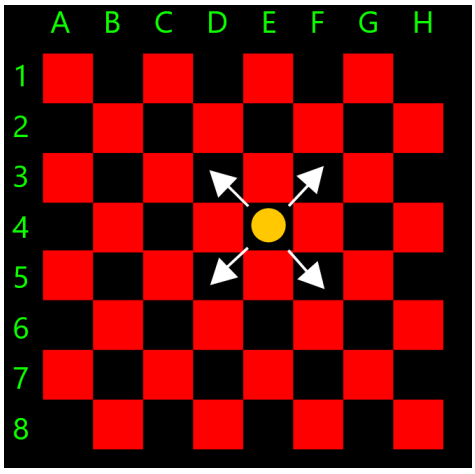
The game must include an interactive user interface which inquires when each player has their turn to move, they are asked which character (only applies to the hound), using a coordinate input from the user. If this character can move to multiple positions, the user is asked which, otherwise the character moves to the given position.

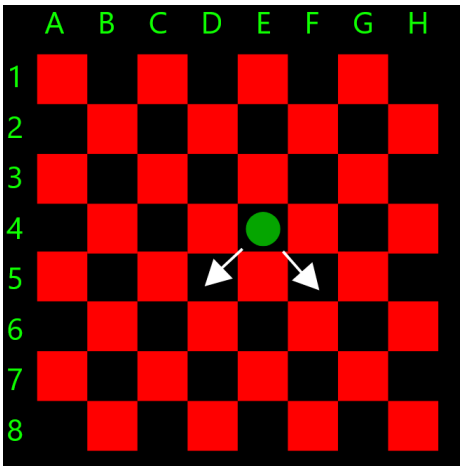
A game is won when the fox has no valid moves left (the hounds then win) or the fox over takes the hounds (the fox then wins). When any one player wins, the user is asked whether another game should be started, if so, the player names, as requested initially, are switched, and a new game begins, now a score is kept.

User Manual

Fox:

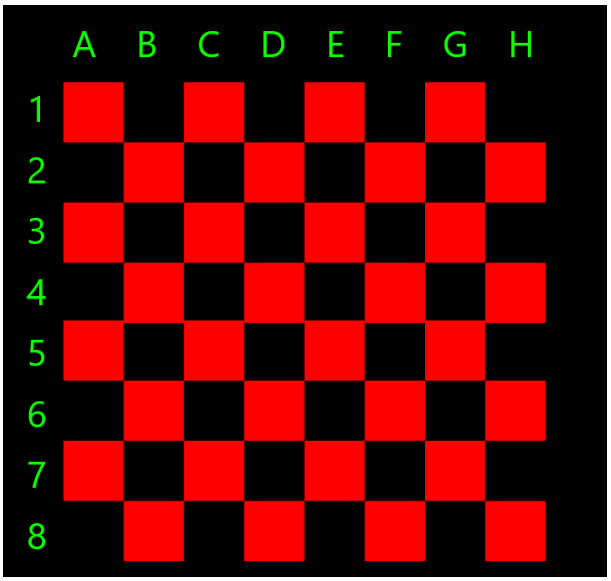
The fox's goal is to get past the hounds, to the other side of the board whilst avoiding being trapped by the hounds. There is only one fox piece that can move diagonally one square forward or back. Starts on A8. Starts as Player 1.





Hounds:

The hounds' goal is to trap the fox (move piece to prevent the fox from having any valid moves left). There are 4 hound pieces that can move 1 square diagonally forward. Only one piece can be moved each turn. Starts on B1, D1, F1 & H1. Starts as Player 2.

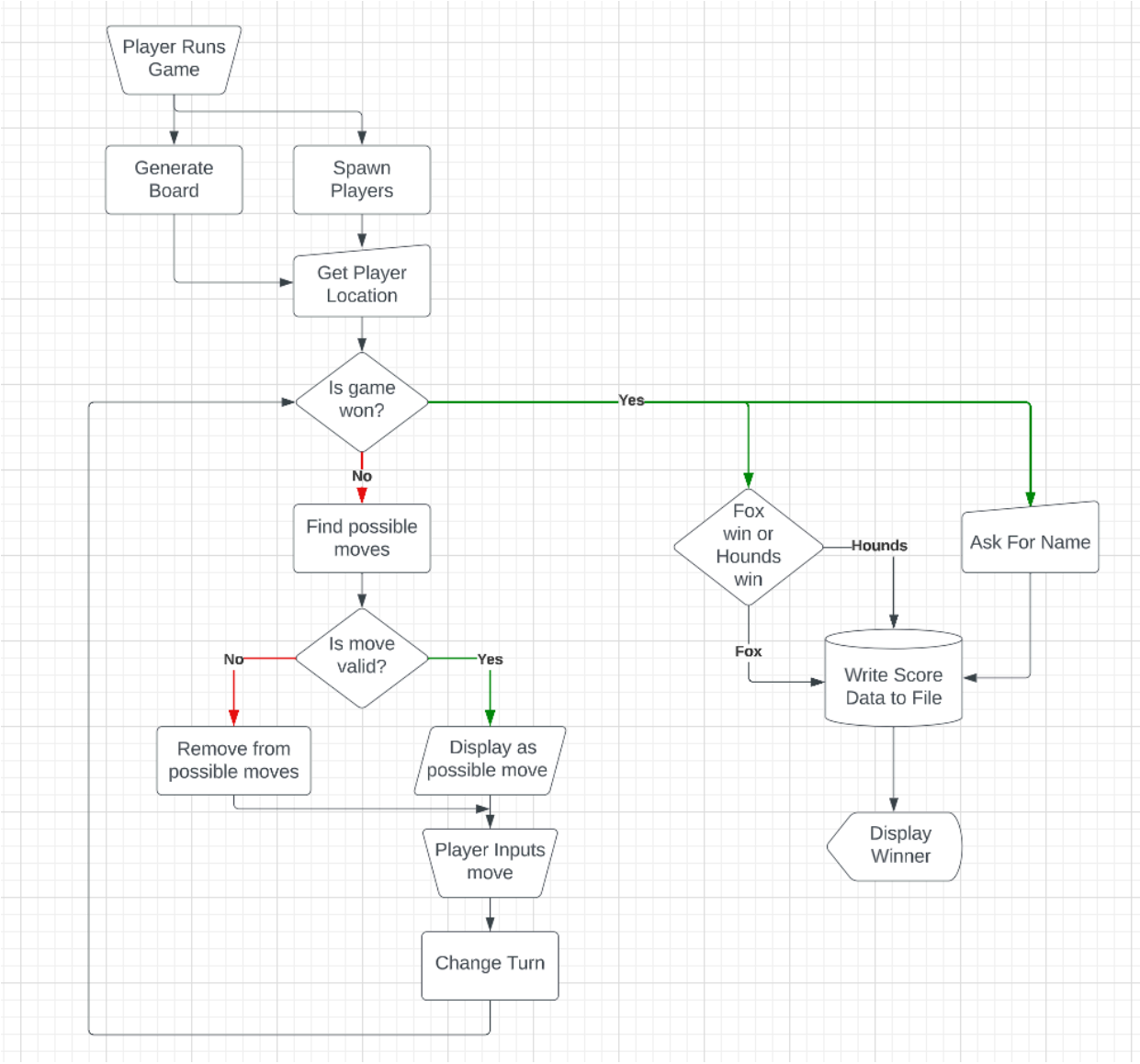


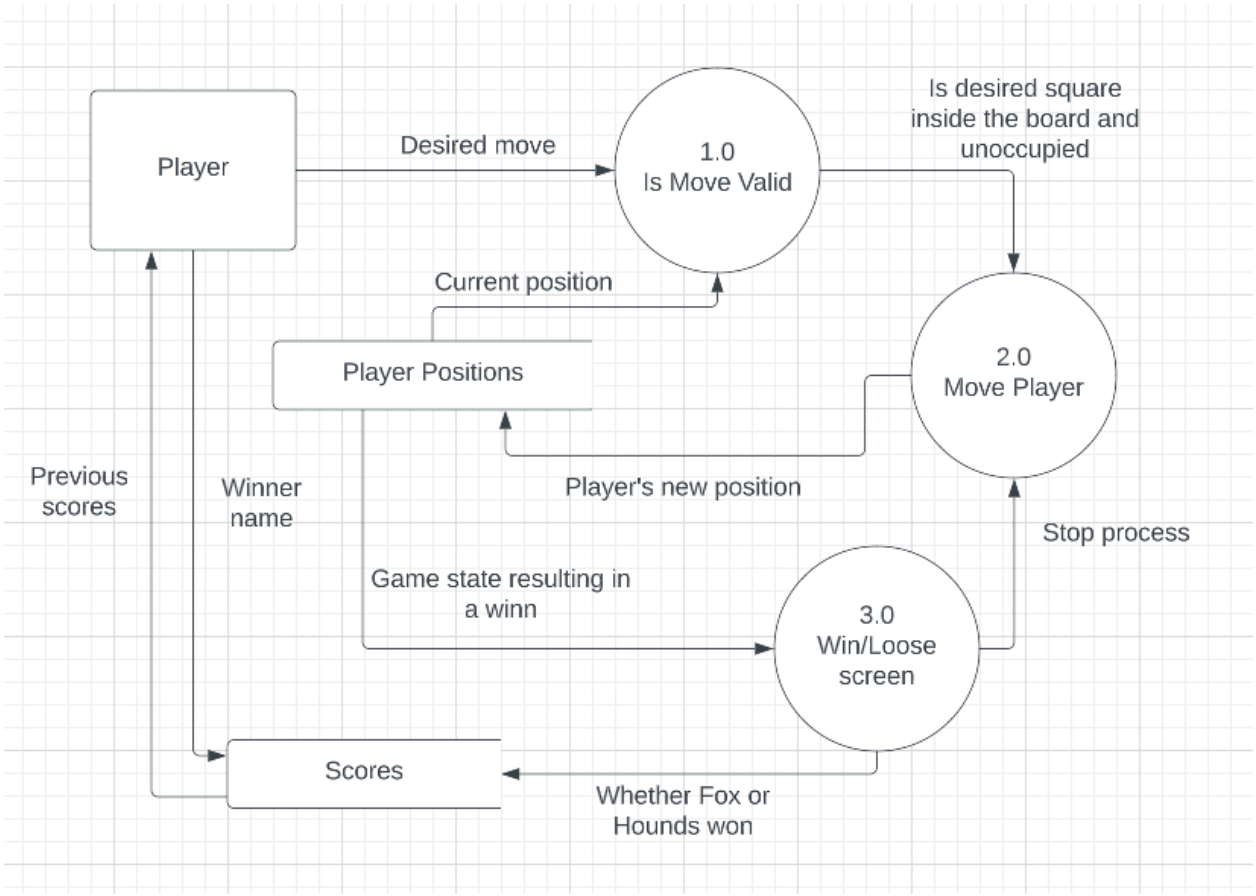
Moving:

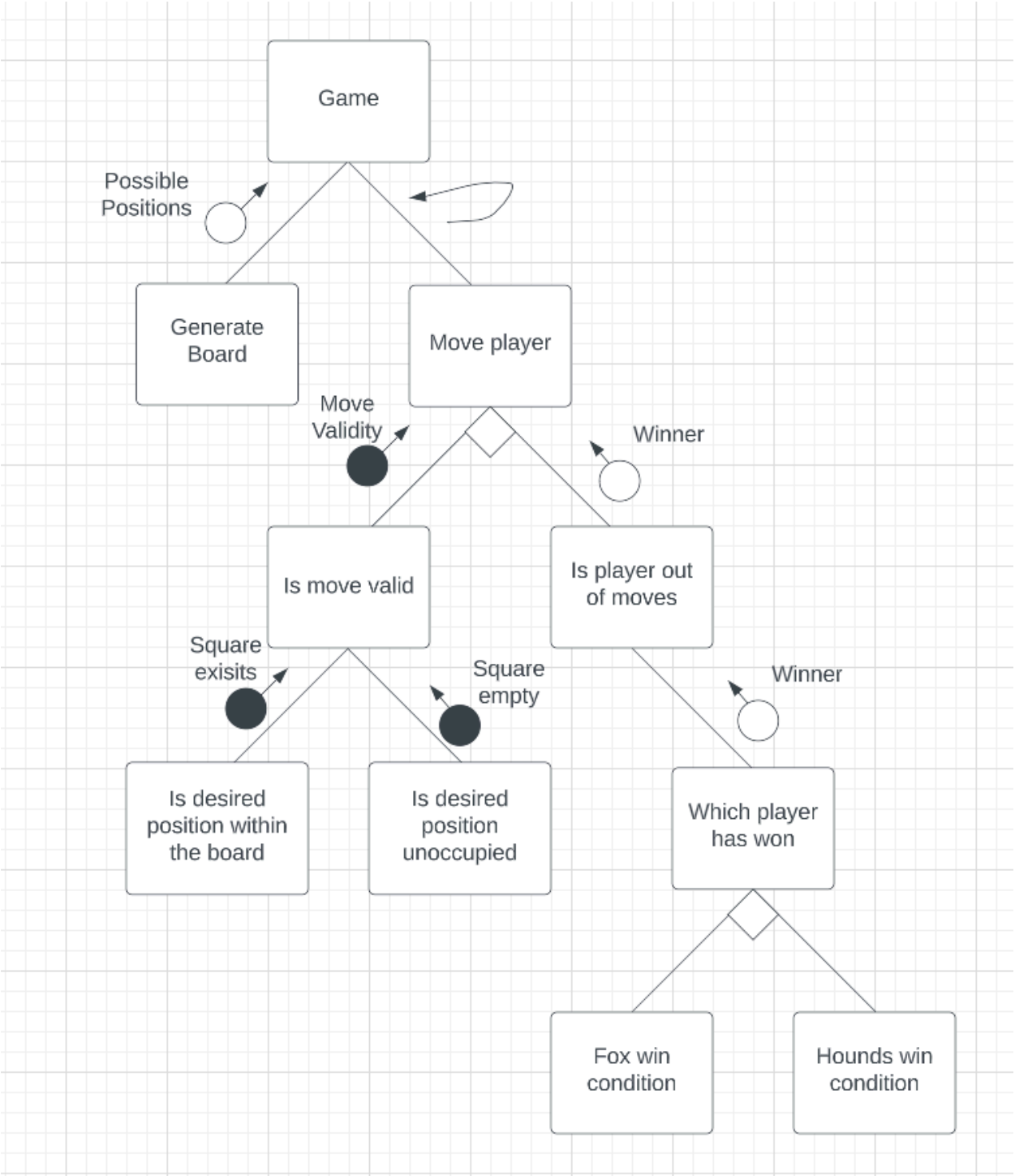
The board is a standard 8 x 8 grid with the black squares being the squares on which players can move. Each turn the possible moves are listed, the player must type in a valid move into the cmd and press enter in order to move to that square on the grid. The squares are denoted as a letter then number (e.g. A1). If there is only one possible move for a player, the player is notified and any input will perform this move.

Planning and Designing the Solution

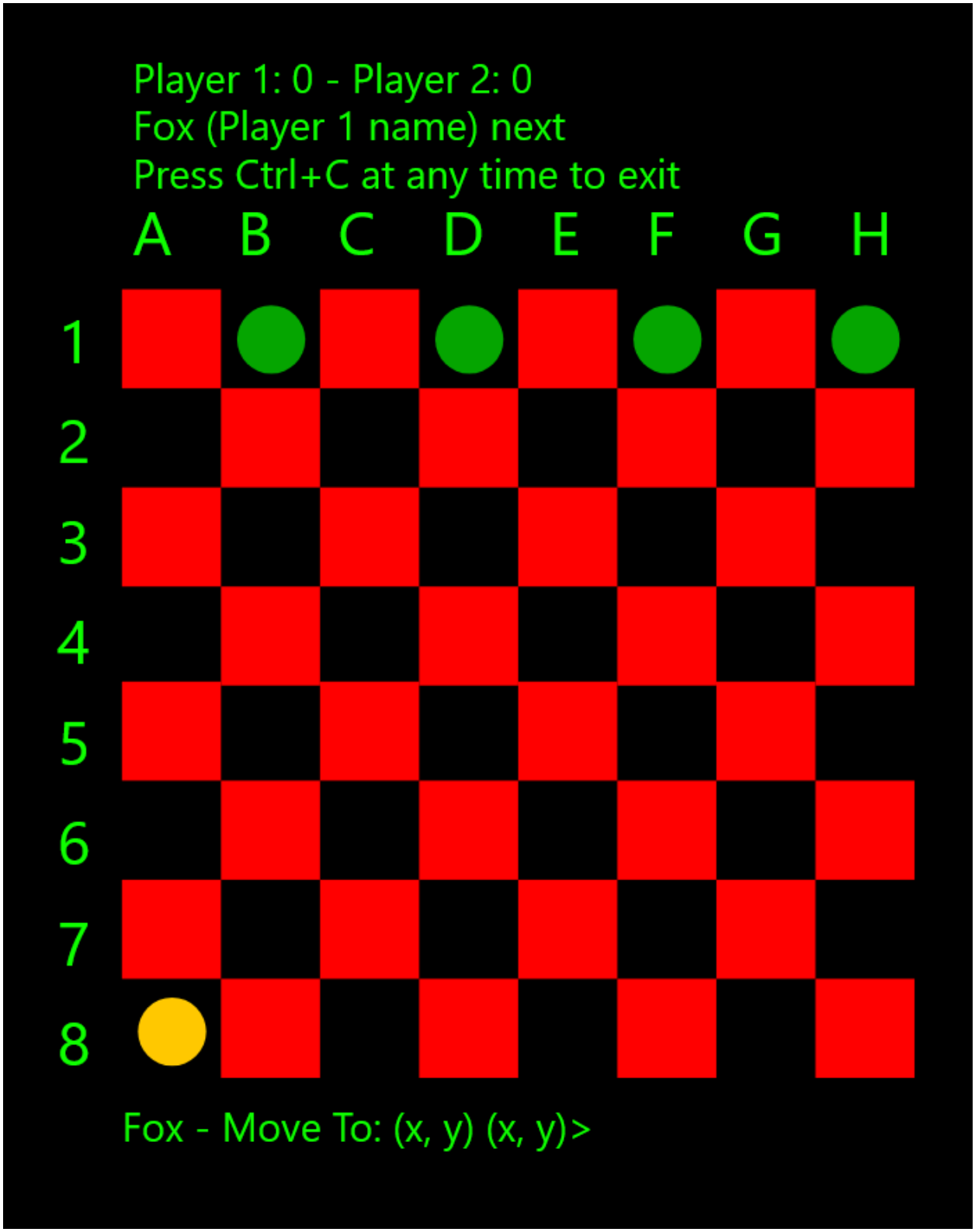
System Charts/Diagrams




Player
Desired move
1.0
Is Move Valid
Is desired square inside the board and unoccupied
Current position
Player Positions
2.0
Move Player
Previous scores
Winner name
Player's new position
Stop process
Game state resulting in a winn
3.0
Win/Loose screen
Scores
Whether Fox or Hounds won


Game
Possible Positions
Generate Board
Move player
Move Validity
Winner
Is move valid
Is player out of moves
Square exisits
Square empty
Winner
Is desired position within the board
Is desired position unoccupied
Which player has won
Fox win condition
Hounds win condition

Screen Design



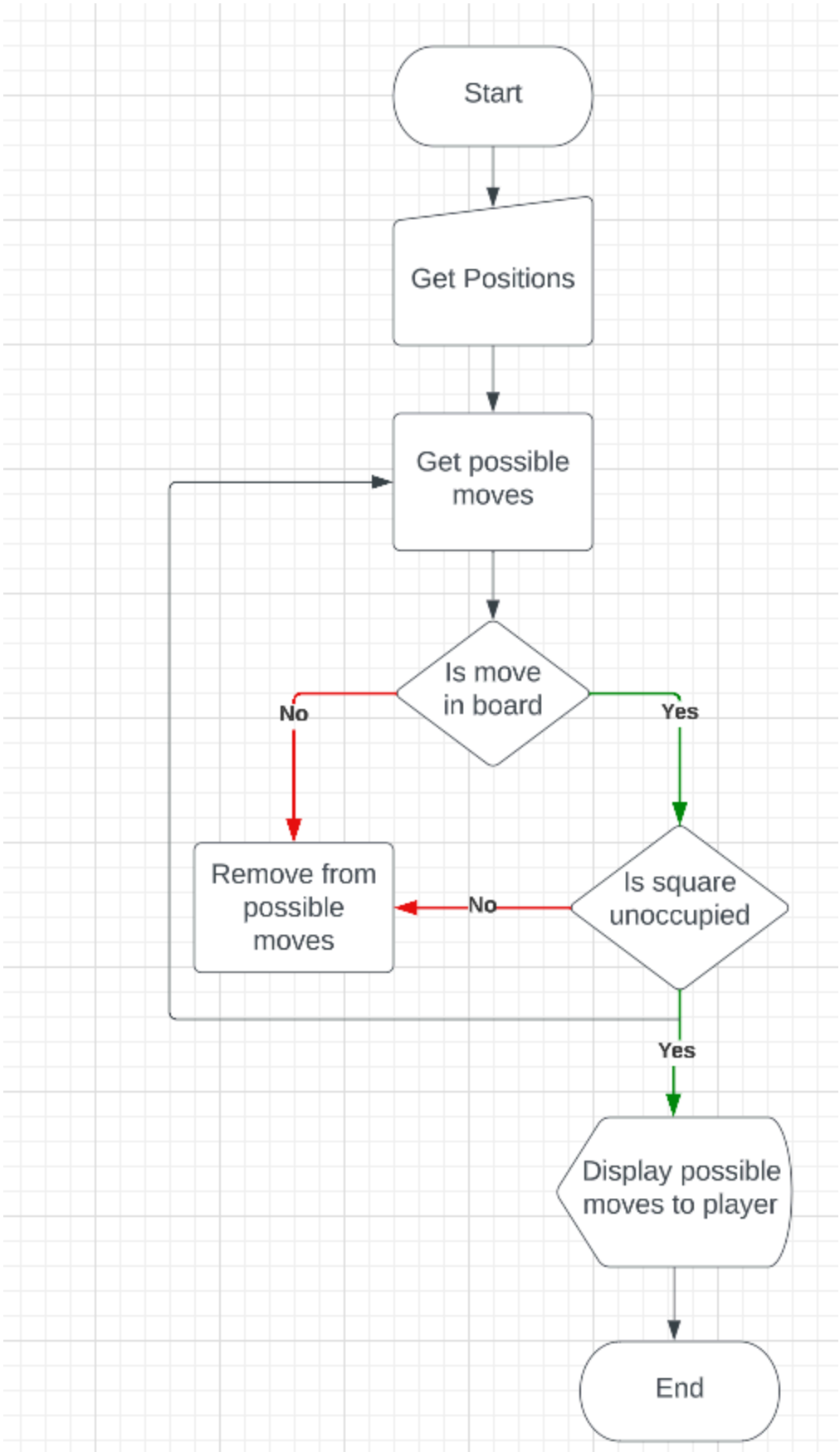

FOX AND HOUNDS
What should Player 1 be called (starts as fox)?
What should Player 2 be called (starts as hound)?

Module Charts/Diagrams

Move Hounds

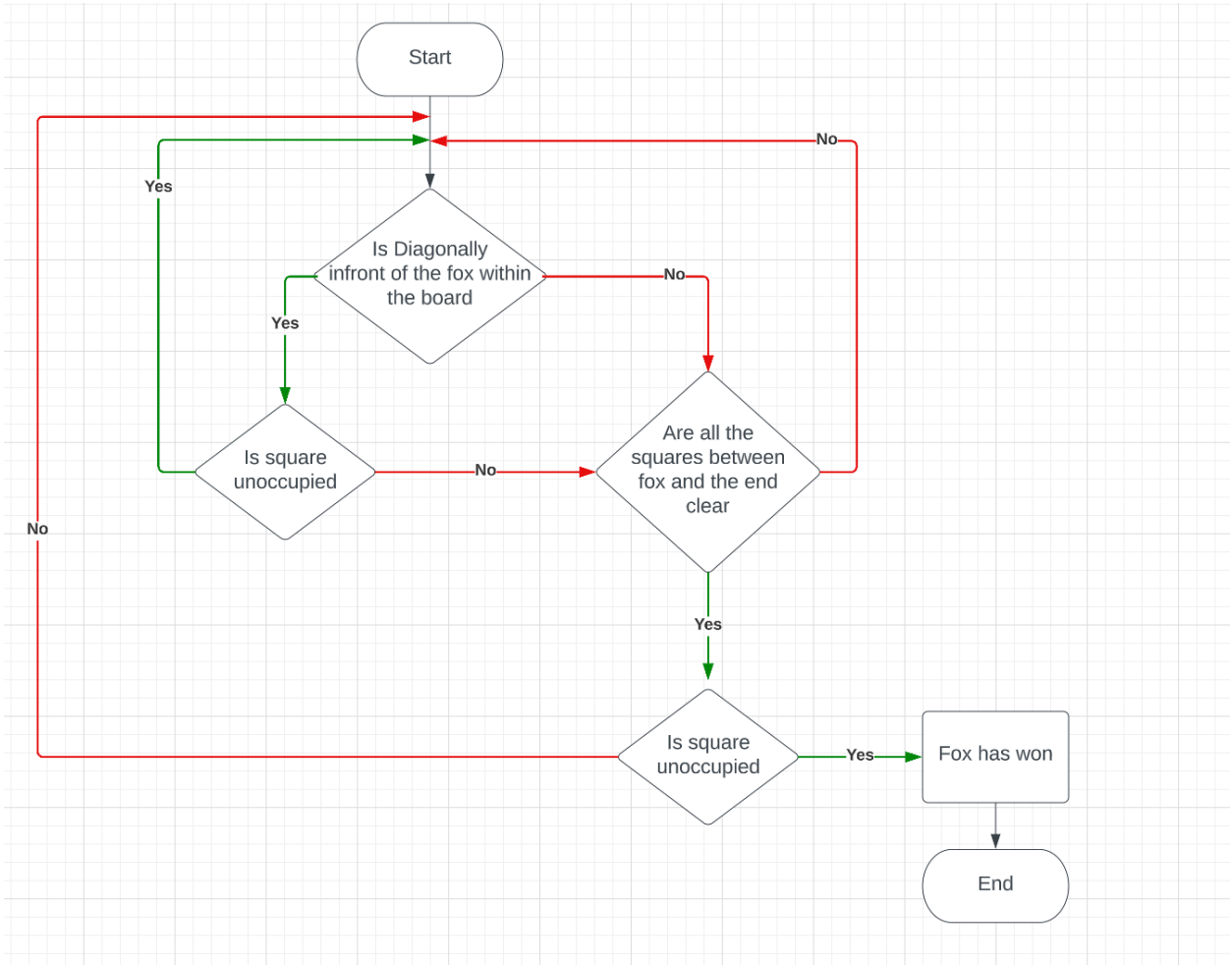
Input	**Process**	**Output**
HoundPositions, FoxPosition	Find which directions all 4 or the hounds can move	
	Filter for positions off the edge of the board	
	Filter for collisions with other characters	
	Ask player where to move	HoundsNextPosition

```
BEGIN Linear_Search(Array, Length, Value)
      found = NULL
      FOR i = 0 TO Length
            IF Array(i) == Value THEN
                  found = i
            END IF
      NEXT i

      RETURN found
END Linear_Search

BEGIN Future_x(CurrentX, i)
      RETURN (i MOD 2) * 2 - 1 + CurrentX
END Future_x

BEGIN Future_y(CurrentY, i)
      RETURN FLOOR(i / 2) * 2 - 1 + CurrentY
END Future_y

BEGIN Get_Hound_Movement(Hound_Positions_Array, Fox_Position)
      FOR j = 0 TO 3
            hound_positions = coordinate array (j)

            FOR i = 0 TO 3
                  next_x = Future_x(Fox_Position.X, i)
                  next_y = Future_y(Fox_Position.Y, i)

                  IF next_x BETWEEN (0, board_width) AND next_y BETWEEN (0, board_height) THEN
                        IF NOT Linear_Search(Hound_Positions_Array, 4, new Coordinate(next_x,
next_y))
                              fox_positions(i) = [next_x, next_y]
                              END IF
                  END IF
            NEXT i
      NEXT j

      movement_position_index = NULL

      WHILE movement_position_index == NULL
            PRINT "Move hound to?"
            GET selected_position
            movement_position_index = Linear_Search(hound_positions, j, selected_position)
      END WHILE

      RETURN fox_positions[movement_position_index]
END Get_Hound_Movement
```


Start
Get Positions
Get possible moves
Is move in board
No
Yes
Remove from possible moves
No
Is square unoccupied
Yes
Display possible moves to player
End

Check if the fox has won

Input	Process	Output
Fox position Hound position	Loop through positions diagonally away from the fox starting position,	
	Check if these contain hounds	
	Loop down toward the bottom of the board from these positions	
	Check if these contain hounds	Whether the fox has won

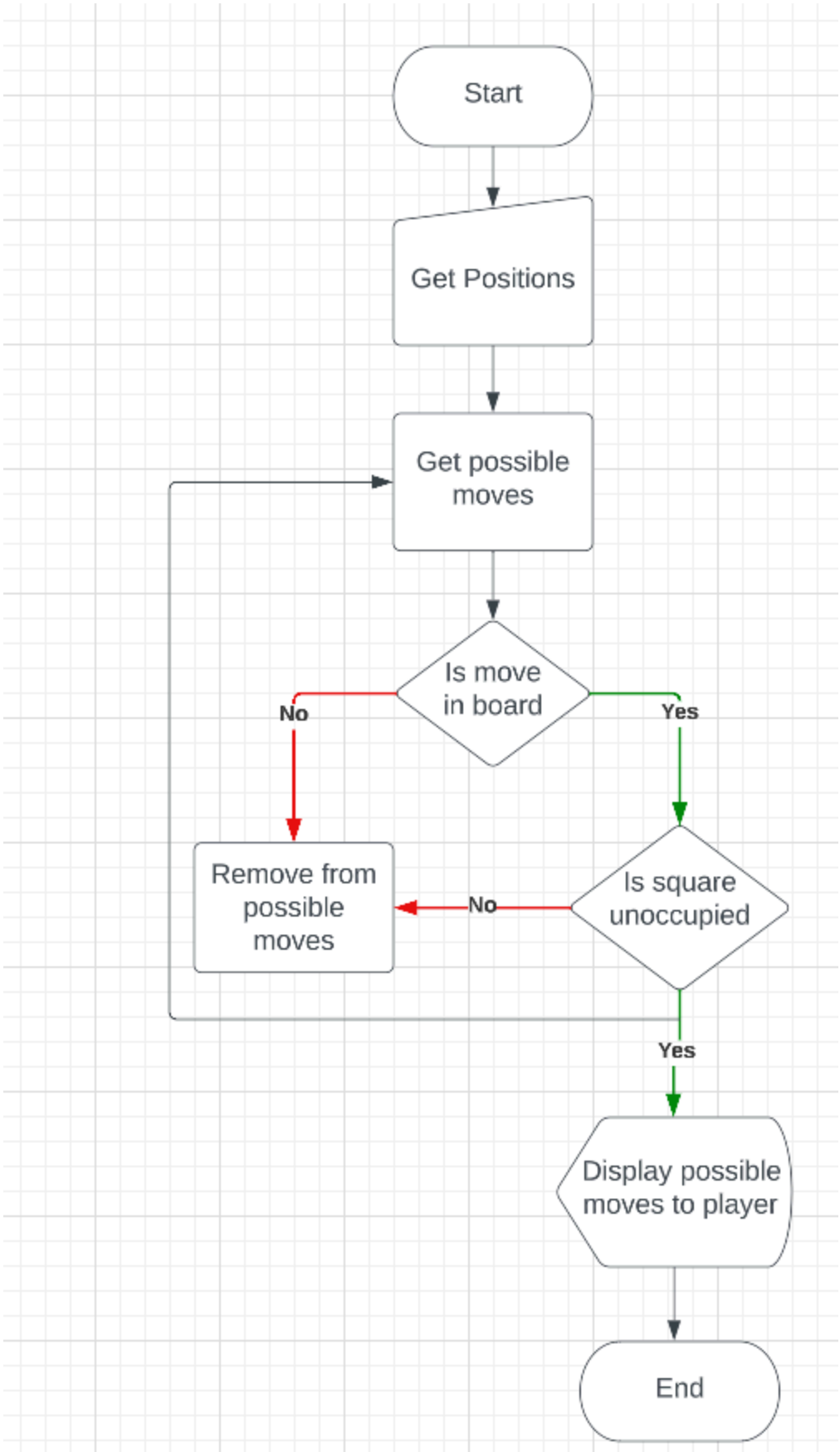


```
BEGIN Linear_Search(Array, Length, Value)
    found = NULL
    FOR i = 0 TO Length - 1
            IF Array(i) == Value THEN
                  found = i
            END IF
    NEXT i

    RETURN found
END Linear_Search

'REM fox begins at the top of board (y=7), while the hounds begin at bottom (y=0)

BEGIN Has_Fox_Won(Hound_Positions_Array, Fox_Position)
    FOR x = 0 TO Fox_Position.X
      FOR y = 0 TO Fox_Position.Y - Fox_Position.X + x
      'REM is escaped automatically if Fox_Position.Y - Fox_Position.X + x < 0

            IF Linear_Search(Hound_Postions, 4, [x, y]) IS NOT NULL
                  RETURN false
            END IF
      NEXT y
    NEXT x

    FOR x = Fox_Position.X + 1 TO 7
      FOR y = 0 TO Fox_Position.Y + Fox_Position.X - x
            IF Linear_Search(Hound_Postions, 4, [x, y]) IS NOT NULL
                  RETURN false
            END IF
      NEXT y
    NEXT x

    RETURN true
END Has_Fox_Won
```

Move fox

Input	Process	Output
HoundPositions, FoxPosition	Find which directions the fox can move	

	Filter for positions off the edge of the board	
	Filter for collisions with other characters	
	Ask player where to move	FoxNextPosition


Start
Get Positions
Get possible moves
Is move in board
No
Yes
Remove from possible moves
No
Is square unoccupied
Yes
Display possible moves to player
End

```
BEGIN Linear_Search(Array, Length, Value)
      found = NULL
      FOR i = 0 TO Length
            IF Array(i) == Value THEN
                  found = i
            END IF
      NEXT i

      RETURN found
END Linear_Search

BEGIN Future_x(CurrentX, i)
      RETURN (i MOD 2) * 2 - 1 + CurrentX
END Future_x

BEGIN Future_y(CurrentY, i)
      RETURN FLOOR(i / 2) * 2 - 1 + CurrentY
END Future_y

BEGIN Get_Fox_Movement(Hound_Positions_Array, Fox_Position)
      fox_positions = coordinate array (4)

      FOR i = 0 TO 3
            next_x = Future_x(Fox_Position.X, i)
            next_y = Future_y(Fox_Position.Y, i)

            IF next_x BETWEEN (0, board_width) AND next_y BETWEEN (0, board_height) THEN
                  IF NOT Linear_Search(Hound_Positions_Array, 4, new Coordinate(next_x, next_y))
                  fox_positions(i) = [next_x, next_y]
                  END IF
            END IF
      NEXT i

      movement_position_index = NULL

      WHILE movement_position_index == NULL
            PRINT "Move fox to?"
            GET selected_position
            movement_position_index = Linear_Search(fox_positions, 4, selected_position)
      END WHILE

      RETURN fox_positions[movement_position_index]
END Get_Fox_Movement
```

Testing and Evaluating Software Solutions

Development Process, Tools

How to Run

The code is precompiled and can be executed by running the a.exe file through cmd in the submitted zip folder. This launches a command prompt windows and opens the game.

General

The game was developed in the C programming language, with code for the files placed in the c/ folder in the code directory.

- _main.c - run the main game loop, get the player names
- constants.c - contains a simple linear search algorithm to find character location collisions.
- constants.h - contains definitions for substitutions related to play modes and coordinates
- rendering.c - responsible for the majority of the output of the code, including the board
- movement.c - controls the movement input from the characters

The only external libraries contained in the code are the stdio.h library for input and output commands, and stdlib.h library for standard operations and functions. With such a limited set of libraries the code could hypothetically run on Mac or Linux systems, assuming that the console executing the code supports it.

Compilation & Execution

On a system with gcc.exe in the path file, the compile_run.bat file can be run to compile and execute the code, this generates the a.exe file, which is also run automatically. This executable file contains the full infrastructure for the game, and can be run, away from the original directory, as is in a console.

Runtime

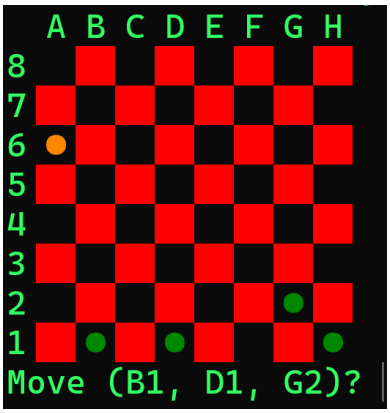
All command output is written to the console, in raw text. This is rendered into a pleasing, retro user interface using ANSI escape codes, which can change the background and foreground colour of a character, and clear the screen, as well as many other functions. Those two listed are the only functions used in this game.

Tests - Edge Cases & Input Validation

Player Names

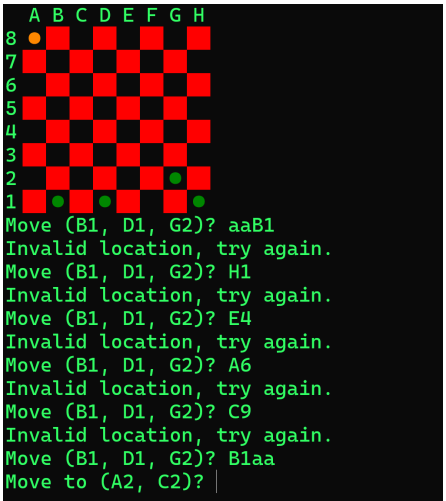
Movement

When a coordinate input is requested, two characters are read, the remaining characters are then discarded. A table is used to show whether some input data is read as expected. The screen when the test occurs is as follows:



INPUT	EXPECTED RESULT	RESULT	EVALUATION
aaB1	INVALID	INVALID	The last two characters are discarded, while the first two do not refer to a coordinate.
B1aa	INVALID	VALID - B1	A harmless false positive is achieved as the last 2 input characters are discarded and only the first 2 are read as B1, a valid input.
H1	INVALID	INVALID	Hound cannot move
E4	INVALID	INVALID	Blank space
A6	INVALID	INVALID	Fox cannot be moved on hound's move.
B1	VALID	VALID	A valid input value.
C9	INVALID	INVALID	C9 encodes to the same value as D1 (2D array overflow), as such if the range of the input was not validated it could have resulted in a false positive.

A screenshot of the tested data is included beside:



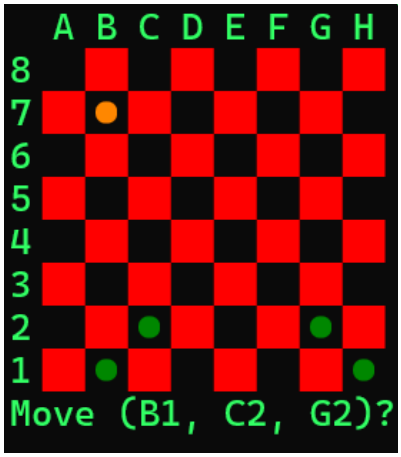
Tests - Game Events

A preset list of game events on which the game loop changes are listed below. A description of the expected result, and a Pass or Fail of each event/condition is included below.

Typical Operation

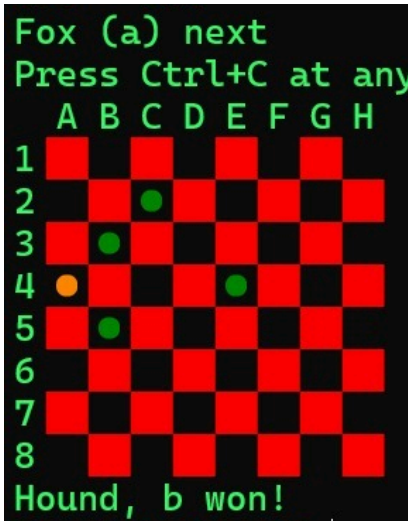
Shown beside is a typical (and the only possible) second move in the game. Once the fox has moved diagonally upward, s the first move of the game, the player for the hounds is asked which hound should be moved, here all hounds are listed as all have at least one legal move, (H1 can move to G2, F1 can move to G2 or E2, etc.). At the top of the board the player names ("b" for the Hounds, and "a" for the Fox) is listed with a score. The score is the sum of all game wins. Afterward, the player which moves next is listed. Finally the standard escape character is listed, which will force stop the game. - **PASS**

Character Collision & Wall Collision



Above, is shown a set of hounds in the given locations. The hound, furthest most to the right is trapped in a corner, unable to move. As expected, the user is not asked whether they would like to move this hound at H1. Diagonal upward, to the right, the hound would collide with the wall. Diagonal upward, to the left the hound would collide with another hound. As such, this hound is not present on the movable list, as expected. - **PASS**

Hound Win



When its the fox's turn to move, and it has no legal moves remaining, it is trapped, resulting in the win condition being reached for the hounds. Beside, the fox may not move to its left as it would collide with the wall, it may not move to its right as it would collide with two hounds (one for either direction). In the image, at the top, the fox having the next move is shown, at the bottom, the hound win message is shown, as required - **PASS**

Fox Win

A fox can win under two conditions, one, (the left most diagram), if the fox evades the hounds, and can escape to the end of the board without being interrupted. Here the fox moving toward row 1, has no hounds in B1 or D1, thus can move freely to this squares and reaches the win condition. In the right most diagram, the Hounds are unable to move in any direction, as such the fox has reached its win condition - **PASS**

Evaluation

Limitations

As stated in "*Development Process, Tools*", ANSI escape codes, and various special characters are used while printing to the console to generate the checkerboard, and to clear the screen between moves. Although these special characters are supported by almost all operating systems and command consoles produced in recent decades, similar support is lacking for the colouration used on the board. The ANSI codes used to change the colour of the board, however, lack the wider support that one would expect. Information from an open source, and microsoft suggest Windows 10 has had support since 2017, while many linux systems have had support even earlier, for 24bit colours, which should mean support is also available for 8-bit colour. The included screenshots are generated in Windows 11's *Terminal* program, and is untested in other command line emulator software.

The compiled C code, which is able to run independent of any other local files, is of 168KB uncompressed, it takes up very little space. The solution is shown by task manager to use "0.8MB" of RAM, which, although well above the expected ram usage level of <100B (calculated manually), it is well below negligable for modern systems, using roughly 0.01% of a standard 8GB of RAM. This is as a result of the use of a lightweight, system level, compiled programming language, which, while limiting debugging options and multi-system compatibility, has the advantage of generating light weight, and highly performant programs.

Criteria - AI

The *Statement of Intent* contains references to using an AI model for the game to include a single player option. Due to the existence of under 25 million board total states it was hypothetically possible to implement the “Overpowered” AI model, that will predict which move is optimal for any given board state from a stored, 20 MB file. Although front end support for an AI model was implemented, due to either compatibility errors or bugs in the model generator, complete support for the AI model was not implemented to completion. Thus, it lacks necessary functionality to run in a fresh game, and is inaccessible from the UI of the game in its latest version. The “Rubbish” AI model was also not implemented as its use is near meaningless as a result of its inefficacy as an opponent, against a remotely strategic player. No AI was implemented to completion in the finalized version of the game, and neither AI models are accessible by a user through the executable.

User Experience

The user interface contains colourised special characters to represent the game board, and command line input (stdin), for input relating to character movement and other options. Although it would've worked as a more effective user interface to implement clickable buttons in the command line, this would've been near impossible to develop with remotely acceptable multi-system support, as it would require calibration and various hacky solutions to algorithmic, and systematic issues. Similarly, implementing arrow key selection, although less difficult, would have required a more complex solution, and was out of the scope of the *Functionality Requirements* for this reason.