Noughts & Crosses Coursework Report

Ross Langan
40276526@napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET08122)

Abstract

This document covers the planning, design and implementation of Noughts & Crosses- a program based on the popular game and written in C language. I have used carefully selected data structures and algorithms to produce a functioning game with a range of required and extra features and will record my choices, thought processes and evaluations in this report. The making of this game was both challenging and fun and required thinking outside the box throughout development. I have furthered my knowledge in many areas of my study and feel more confident coding in the C language.

Keywords – Algorithms, Structures, Board, Game, Text-based

1 Introduction

The objective of this coursework is to produce a playable text-based game of noughts and crossesr primarily focusing on the use of appropriate data structures and algorithms. The basic requirements of the game are to implement appropriate structures and algorithms to represent the game board, players, pieces(the noughts and crosses) and the positions of the pieces. Further requirements include recording the history of play and automatic playback of the recorded games, the ability to undo and redo moves, and play against an automated player.

The game of noughts and crosses I have produced supports playing against both a basic computer and another real-life player. Moves are entered one-by-one on the keyboard and when submitted appear on a 3x3 board on screen. Users have the ability to undo and redo moves using numeric codes also entered on the keyboard. On completion of a game through win, loss or draw the user will be notified of the result of the game and is given the option to play again or quit. The option is also given in the main menu to replay the last game which the user can watch the game being played- each move plotted in 1 second intervals.

In designing the game I faced several challenges and had to make many decisions surrounding different data types and structures, functionality and design aspects. In order to implement some features I have used code which could be improved with further time and knowledge including use of the highly discouraged 'goto' function. The final version of the program I have produced is fairly long, and with improvement could be condensed into a shorter, cleaner program. Further improvements and some analysis of my code will be investigated in the following sections.

2 Design

2.1 Basic Structures

In meeting the minimum requirements I have used my prior knowledge and self directed learning to represent the game board, players, pieces and positions with appropriate data structures. In this section I will begin with the structures used in the final version of my program and discuss the problems I faced throughout.

When initially faced with this coursework, the first step I identified was to print the 3x3 board to the screen. I used physical sketches and notepad++ to create a number of designs for character-based boards using pipes(||), plus symbols(+) and dashes(-) and settled on a design with outside borders and numbers in the center of each square. To print the board to the screen I used the 'printf' function- one line of the board per printf statement. In order to show each number in the board in a way which it can be replaced with a nought or cross I used the '%' format specifier and refer to the matching values in an array of 9 values- the numbers one to nine, one for each space of the board. I store the above instructions to build the board inside a void function. Also known as a non-value returning function I chose to use this structure as it does not return a value, therefore never affecting the game, and can be called at different parts of the game, for example straight after a move is plotted, to construct and update the board on the users screen. This means that I do not have to repeat the instructions to build the board more than once in the whole program.

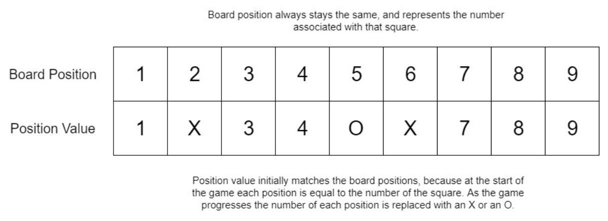
To represent the board pieces I used the char data type to store the values X and O. I have stored the character X in a char named 'X' and the character O in a char named 'O'. This is a simple way of storing each of the pieces which I found useful as each value of the board can simply be replaced by either of the chars at the appropriate times and eradicates the need to set it to a character in apostrophes each time.

In storing the players of the game, I decided for simplicity not to directly store the number of the active player(alternating between 1 and 2), but rather store the turn and use it to determine which player is active. At the beginning of each game the turn value is set to 1. The value increases by 1 every time a player submits a position, reaching the point a player wins or turn 10 when the board is full. The turn value can change up or down when the undo or redo keys are used. In order to determine which player should be active, I use the modulo operator to check whether the current turn value is divisible by 2. If the turn is not divisible by 2 then the number is odd and it is player 1's turn. If the turn is divisible by 2, it is even and therefore it is the turn of player 2. Using this technique allows the program to accurately keep track of the player's turn even after move undos and redos.

In order to store the positions of the X and O pieces on the board I created an array of 9 chars as referred to above in the creation of the board. The array, 'square', is initialized at the start of the game with the numbers 1-9- one for each number on the board. As each move of the game is made, the number matching the users choice will be replaced with an X or O based on the turn. Using an array of chars was the best choice for storing these values because any element can be accessed in the array in any order and they can be easily replaced at any point in the game.

2.2 Stacks

Using the above structures I was able to construct a basic working version of the game. Each turn the user could select a number and the corresponding square on the board would change to the piece, however this version had no way record the order the moves were plotted in and users could plot their piece on a square that was already taken. To add validation of the number chosen and progress the program my first course of action was to implement an associative data structure (as seen in figure 1) to store the board position and the board piece as key value pairs. Using this structure I was able to produce a working game with users unable to choose taken squares, however after problems with implementing the undo functionality using this technique, I reconsidered my use of the associative data structure. At this point it felt like a complicated way of achieving what I required.


Board position always stays the same, and represents the number associated with that square.

Board Position	1	2	3	4	5	6	7	8	9
Position Value	1	X	3	4	O	X	7	8	9

Position value initially matches the board positions, because at the start of the game each position is equal to the number of the square. As the game progresses the number of each position is replaced with an X or an O.

Figure 1: **Associative Data Structure**

After consideration of all possible data types, the stack stood out as the perfect choice for this game as each move can be pushed into the stack in a first-in-last-out(FILO) order. This is perfect for the undo function as the previous moves can be popped off one-by-one until the first move is popped and the stack is empty. I modified my working program to replace the associative structure with a stack (of size 9) and included code to replace the numbers on the board(in the square array) with an X or O if they are pushed into the stack. To implement the stack to the best standard possible I followed the Lab 03 located on Moodle closely. [1]

With the stack implemented successfully and the game running as before, I used the pop function I had set up to handle the user undoing a move. Every time the user enters the number '44' it triggers an undo and the stack pops the most recent value added. This process can be repeated as far as the start of the game.

In order to include the redo function I thought carefully about the similarities between the undo and redo functions and concluded that they are similar functions that perform opposite tasks from each other. Following this train of thought I considered the use of a second stack, holding the moves that have been popped from the first stack. As seen in figure 2, when the undo function is performed the number of the board position is popped from the game stack, removing it from the board, and pushed into the redo stack.

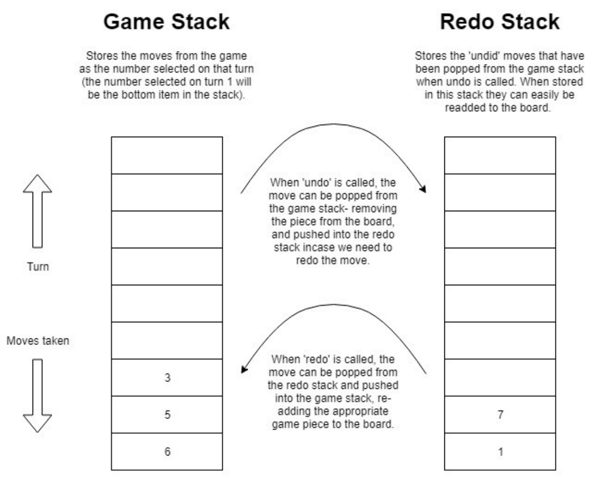

Game Stack
Stores the moves from the game as the number selected on that turn (the number selected on turn 1 will be the bottom item in the stack).
Redo Stack
Stores the 'undid' moves that have been popped from the game stack when undo is called. When stored in this stack they can easily be readded to the board.
Turn
Moves taken
When 'undo' is called, the move can be popped from the game stack- removing the piece from the board, and pushed into the redo stack incase we need to redo the move.
When 'redo' is called, the move can be popped from the redo stack and pushed into the game stack, re-adding the appropriate game piece to the board.
3
5
6
7
1

Figure 2: **Game Stack and Redo Stack**- moves can be transferred between the stacks if the user undos or redos moves.

The redo stack when not empty contains the series of moves that have been undid from the current game. This means that at any point in the game the player can undo all the moves in a backwards order from the current move to the start of the game. When popped from the game stack they will be pushed straight into the redo stack in an order where they can be re-added to the board in the same order they were removed.

The use of two stacks working in parallel was definitely the right choice for implementing the undo and redo functions and for storing the positions of each game piece. By using the turn variable to keep track of which player's turn it is and using stacks to store the moves the game will not produce any errors related to the undo and redo features.

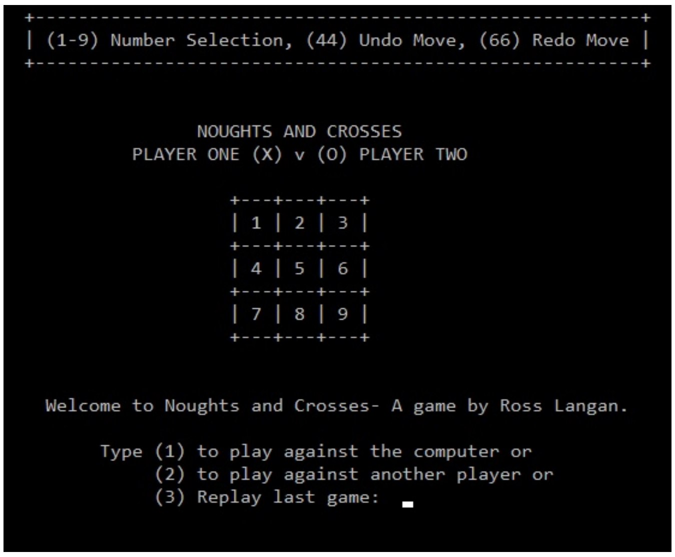
For the majority of the development I had included a queue in my program to store the completed game for replaying. This structure worked, however it would not take into account the undid and redid moves. If these functions were called the queue would contain the moves up to that point and all new moves after that. After studying the code associated with the queue I realised that I could quite easily implement a third stack to store the completed game in the correct combinations of moves.

When a game is completed by any means the game stack contains the complete list of moves. Any undid or redid moves will have been corrected in this stack and therefore the game stack is exactly how the game was played. In order to replay a game I pop every value from the game stack straight into the replay stack, which will be empty for the first game of the session. When the user selects the 'replay game' option the board is cleared and the moves are popped out the stack one by one onto the board. This will continue while the stack is not empty. When the stack is empty the game is complete and the winner is determined based on the current turn. Unfortunately this feature currently only works on the first game of the session however with improvement I would hope that it could play the last game played and empty the stack when a new game is started.

3 Enhancements

3.1 Visual Changes

In terms of the visual aspects of the game I do not believe there are many enhancements that would improve the quality or the playability of the game due to the simplistic nature of the noughts and crosses, however I have identified two improvements that would add the final touches and high quality finish that the game interface requires. The main visual problem I encountered, and one that proved bothersome throughout the entire project was the centering of the screen elements- the key, the game board and the text below. Because of the frequent changes to the game including the addition of undo and redo features, start menu and multiple blocks of text it was extremely hard to keep each element centered in the middle of the screen. Using a trial and error approach and a combination of tabs and spaces in the print statements I had to center the key, game board and multiple blocks of text several times during the project.



```
+------------------------------------------------------+
| (1-9) Number Selection, (44) Undo Move, (66) Redo Move |
+------------------------------------------------------+


                 NOUGHTS AND CROSSES
           PLAYER ONE (X) v (O) PLAYER TWO

                    +---+---+---+
                    | 1 | 2 | 3 |
                    +---+---+---+
                    | 4 | 5 | 6 |
                    +---+---+---+
                    | 7 | 8 | 9 |
                    +---+---+---+


  Welcome to Noughts and Crosses- A game by Ross Langan.

       Type (1) to play against the computer or
            (2) to play against another player or
            (3) Replay last game:
```

Figure 3: **Launch screen-** Centering is not perfect

The second visual element I would change is adding a a running score for each player in the case of multiple games being played. For example if a user plays against the computer 3 times, I would like to add the number of wins for each player at the top of the screen like a score board. This adds to the competitive aspect of the game as users can determine who is winning over multiple plays. In researching other versions of noughts and crosses I found that the majority of versions contained some way of retaining the score over multiple games.

Figure 4: **Scoreboard as seen on Google's version of tic-tac-toe**

3.2 Operational Changes

A major operational issue I identified early on in the development is the infinite loop that is triggered when a player enters a letter rather than a number at any point in the game. I tried to implement some error handling code to validate the entered character was an integer and prompt the player for a different response however after many unsuccessful attempts I chose to focus on other issues in the game. Researching online for solutions, I found several articles referring to this issue including an article on c-faq.com [2] detailing the reason for the common error, however I still could not produce a working fix. Removing the possibility of triggering an infinite loop is one of the major changes I would make to my game to improve the playability. I would additionally like to change some of the inputs to accept text. Ideally the player would type 'u' to undo a move and 'r' to redo a move.

3.3 Code Changes

The code I have written for this game is 700 lines long and contains switch cases, fairly large loops and a few functions such as the reset function(resets the contents of the arrays). In reviewing the code for enhancements there is a few areas in which I could have improved the flow and quality of the code.

During development I had difficulties restarting loops or reaching parts of code when conditions were not met. I researched ways to return to previous bits of code or other loops and discovered the 'goto' statement on tutorialspoint [3]. Although the goto function was exactly what I needed the use of it is not encouraged. The quote below is taken from the webpage.

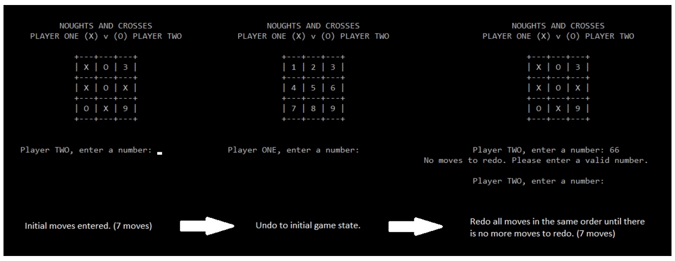
"Use of goto statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them."

In my program I use goto statements in 2 different situations- one where the user tries to redo a move and there are no moves in the stack and the other when the user wants to play again and the program must return to the start of the main loop. Ideally I would like to remove the use of these statements in my program as they are seen as bad practice. Additionally because any program that uses them can be rewritten my program could be modified to work without them.

4 Critical Evaluation

When the game is launched the first screen the user will see is the main menu page with 3 options (as seen in figure 3). I think this page as a means of navigation works well. It provides a good first impression of the game because it is presented in a simple format with 3 clear options and the game board and key are visible. The navigation in other parts of the game follows the same format and uses the same keys, further adding to the consistency of the game.

Both the undo and redo features work well and exactly as I had planned them to. By using stacks to represent the moves to undo I am able to pop the moves from the stack all the way to the initial game state as requested in the requirement specifications. Similarly I am able to pop moves from the redo stack to replot the moves on the board to the furthest move that was reached. This can be performed as many times in one game as required and will always work as expected. To ensure the features were robust and definitely worked as intended I performed tests on the features both throughout implementation and after my game was complete. In the image below you can see the three stages of the testing I performed- plotting several moves, undoing all moves to the initial game state and then redoing all moves again until there is no more to redo.



```
NOUGHTS AND CROSSES
PLAYER ONE (X) v (O) PLAYER TWO

+---+---+---+
| X | O | 3 |
+---+---+---+
| X | O | X |
+---+---+---+
| O | X | 9 |
+---+---+---+

Player TWO, enter a number:
```

Initial moves entered. (7 moves)

```
NOUGHTS AND CROSSES
PLAYER ONE (X) v (O) PLAYER TWO

+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 7 | 8 | 9 |
+---+---+---+

Player ONE, enter a number:
```

Undo to initial game state.

```
NOUGHTS AND CROSSES
PLAYER ONE (X) v (O) PLAYER TWO

+---+---+---+
| X | O | 3 |
+---+---+---+
| X | O | X |
+---+---+---+
| O | X | 9 |
+---+---+---+

Player TWO, enter a number: 66
No moves to redo. Please enter a valid number.

Player TWO, enter a number:
```

Redo all moves in the same order until there is no more moves to redo. (7 moves)

Figure 5: **Testing Stages**- (l) Set of moves entered, (c) Moves undid to initial state, (r) Moves redid to furthest point

A feature which I thought was essential is the ability to play against an automated player. I determined this was an essential feature because quite often you may not be with a second person to play the game against. This feature works well on the basis that the user does not try to undo or redo moves as I have not made these features work in this game mode.

In testing the versus computer mode I found an issue with the random function from C Library. Each time I tested the game mode the computer would always produce the same sequence of random numbers. The computer would choose 4 first, then 9, then 8, and so on in the same order each time. Investigating further I found that if I chose 4 first, the computer would just move on to the next number and plot 9. I concluded that the computer was not generating truly random numbers each time but rather the same randomly generated sequence of numbers each time the game started.

I was able to find the solution quickly by searching on stack overflow [4] for the issue. The answer was simply that I had not seeded the rand() function. When the function goes unseeded the default seed is set to 1 and it produces the same number sequence each time. In order to guarantee a unique sequence of moves each time I have seeded the function with the current time, meaning that there should be no chance of the same seed in a play session.

Referring to the above, I think that the computer game mode works well as it generates completely random moves, it is possible for both players to win the game and the game will be recorded successfully in full because of the lack of undo and redo moves. Furthermore as expected the computer complies with the rules of the game unable to choose spaces that have already been chosen.

The replay game function is perhaps where my game falls below the high standards I have tried to maintain throughout. Due to time constraints this feature is not complete. At the present moment it can record the first game that is played in the session. After this game the program will keep trying to fill the stack with moves from the current game. Realistically this should not happen after a full game has been recorded.

Overall I am slightly disappointed at how this feature has turned out. Although it partially meets the requirements in the sense that it can record a game and play it back at a future time it does not meet the same standards as the rest of my game. In the future I would look to improve the quality of this feature by adding in support for recording more than one game and automatic playback rather than step-through.

5 Personal Evaluation

On the release of this coursework I decided due to upcoming assignments for other subjects it would be a good idea to start work as early as possible to allow the maximum time for development. Overall this was a great idea as I have had a full month to develop a working game with features well above the minimum requirements. My eagerness to perform well however came at a cost as I was so confident with my initial idea on how to build the game(associative structure) that I did not plan enough to foresee the problems that the chosen structure would cause. Dealing with choosing the wrong data structure for an important aspect of my game proved to be a critical challenge I faced and overcame, ultimately costing me 3-4 days of development time.

While restructuring and writing my code I learned about the importance of good practice when writing code. The text-editor I was using- Notepad++, does not provide auto-indenting for C as far as I am aware, so moving around code and adding in loop statements particularly would make the program very very untidy. To assist me in producing quality code I installed the NppAutoIndent plugin. This both allowed me to maintain high coding practices and strengthened my knowledge that it is much easier to work with code that displays high standards. In particular I found it much easier to follow the flow of code when loops are displayed in the correct format.

Another technique I used to help me follow the flow of code was commenting each function, loop and complex pieces of code with a short description of the purpose. After using this technique for several days however, I found that in reading through the code or trying to quickly navigate to other sections of the program that the comments were distracting and made the program hard to read. It is for this reason that I have learned that personally I would rather have two copies of the code active at the same time- one commented and one uncommented. This allowed me to refer to commented sections of code on one program and adapt the code on the other uncommented file.

During the course of the assignment I faced multiple challenges which changed the way my code works, the requirements I fulfilled and the knowledge I took from this project. The first challenge I had to overcome was getting used to coding in C again, having used C# as my main language for a reasonable amount of time and not using C for a year or so. This challenge was fairly easy to overcome having completed all the class tutorials in full and having prior knowledge of the language. Generally speaking I enjoyed the change to C and the development associated with it and I have furthered my knowledge of the language.

One of the biggest tasks I was presented with was adding in the ability to play against a computer. My main loop was already reasonable large for a loop and this solely provided play against another user. In order to give the user a choice of which game mode to play and implement the computer mode I had to massively increase the size of the loop and eventually settled for using a switch statement to provide better navigation in the loop. This challenge took me several days to complete due to the complicated nature of the loops within loops and the validation involved in the data entry. Constructing loops with many many conditions was definitely the most challenging aspect of the assignment.

In order to overcome the challenges I faced throughout I used several techniques to find a solution to the problem. The first step I took was always to take a step back and assess the situation, thinking of different ways to approach the problem and if I could determine a way to fix it. Occasionally this technique was effective as I found taking a step back and clearing my head can help me see problems in a different way. On failure of this technique I would search stackoverflow and other sources I trust for solutions. This usually works for both easy problems such as errors displayed on the command line due to missing brackets and undefined variables, and more obscure issues such as the random number not being entirely random.

Further techniques I found useful was to create a copy of the program and experiment with changing code in the copied version as to not ruin a working version of the program. As I have learned recently this is similar to creating branches on github as the changes you make will not affect the master copy of the program. I occasionally also left a problem till later and worked on a separate part of the program so I could return to the initial problem with a fresh mind.

I feel that my performance of this coursework has been of a fairly high standard and would hope that my marks reflect this. I decided from reading the specifications at the beginning of the project that I would try and aim for an 80%+ submission and I have retained this aim throughout. In adding some extra features and strongly focusing on the visual aspects of the game I have produced a game which I think is very playable. Overall I have stayed relaxed throughout the development of the coursework and although I am very happy with my performance with upgrades to my code I would be even more pleased.

References

[1] c faq, "Data structures 2: Contiguous structures," https://www.dropbox.com/s/bxmt5eqgnqruk3k/lab03_data.structu

[2] c faq, "Scanf jam," http://c-faq/stdio/scanfjam.html.

[3] Tutorialspoint, "C goto statement," http://www.tutorialspoint.com/cprogramming/c_goto_statement.h

[4] H. Yassir, "Why do i always get the same sequence of random numbers with rand()?," https://stackoverflow.com/questions/1108780/why-do-i-always-get-the-same-sequence-of-random-numbers-with-rand.