Coursework Report - Tic-Tac-Toe C Application

Christopher Johnson
40275286@live.napier.ac.uk
Edinburgh Napier University - Data Structures and Algorithms (SET08122)

1 Introduction

The aim of this coursework was to create a Tic-Tac-Toe style game using the programming language C. This was to demonstrate an understanding of different data structures and algorithms. The game I produced is a command prompt based working Tic-Tac-Toe game with some extra features. Users can play the game against another person or against the computer, with the user being notified when the game has finished with the result of the game. Users can undo moves if a mistake is made and they can choose different levels of difficulty when playing against the computer.

2 Design

Visually, the application I have created is very simple due to the limitations of the windows command prompt. My user interface contains 2 main pages, the main menu (Appendix A) and the in game view (Appendix B). There's also an end game and difficulty selection view but these are only seen briefly by the user. I made sure that it is clear to the user how to interact with a page using instructions telling the user what key does what at the very top of the page. These interfaces are built using a series of print statements and populated by data from the board array. I chose to make the layout of the board squares to be the same as a keyboards numpad so that it is more natural to play by mirroring the boards squares on the keyboard.

To create the basic Tic-Tac-Toe game I used an array, a data structure with a sequence of memory allocations which each store an element of data, to store the content of the board. I could have used an array of size 9 and stored the values of 1 to 9 as each board square number but I instead opted to use an array of size 10 and store a random character as the first element (array[0]) and 1 to 9 as the rest. Although a larger array will make the program larger and (on a larger scale) potentially run slower, this had negligible performance effects. This made it so each element had the same location as the board square number stored inside it. This made referencing elements inside the board array much less confusing whilst working on the code because "board[1] = 1" as shown below.

```
1    char board[10] = {'-', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
```

The Tic-Tac-Toe game works by taking user inputs one at a time. Assuming a player makes a move and does not quit or undo, the program checks that the board space is free. If the space is taken the user is prompted to make another selection. At the end of every move the program checks that for a win. This is done by checking if any of the possible win combinations are present on the board with a series of if statements. If there's a victory a message appears telling the user who won (Appendix C). If every board space is taken and there is no winner then a draw is announced.

When playing against the computer the user is given the choice of two difficulties (Appendix D). The easy difficulty means the computer will select the next available space for a move. This would have been better randomized as once the user figures this out it is impossible to lose. When the user is playing with the hard difficulty the program will check if there either user is about to win with two out three in a row. The computer will move to complete this with the third in a row which will either block the user from winning or make the computer win. If the computer does not find one of these it will make a move in the next available square.

In order to allow a user to undo a move, another data structure I used was a stack. Stacks are last-in-first-out data structures. When a stack is initialized an item can be added using a push but in order to read data from the stack the last item to be pushed is popped and can be read. I used this to keep track of player moves. Every time a player makes a move the board square number they selected is added to a stack. If the player wants to under a move the top item is popped and the number read is the board square that is reset back to it's number and not an 'X' or 'O'.

```
1    // Stack that will be used to keep track of players' turns
2 struct stack
3 {
4     int array[9];
5     int top;
6 };
```

3 Enhancements

If you know what you are doing it is possible to never lose Tic-Tac-Toe. If I had more time I would have liked to create an unbeatable difficulty for the player vs computer mode. Although this does not add much in terms of user enjoyment it would have been a challenging and interesting feature to implement.

For both difficulties when playing against the computer, it is possible to learn how the computer plays and predict it's moves such that you can win every time. I did try to implement an element of randomness but struggled to get it working in time. If I had more time I would have made the easy difficulty randomly select moves and the hard difficulty

would have made a random move if a better move isn't found first.

One of the additional requirements for the coursework was to record the history of play and allow earlier games to be automatically replayed. I was not able to get this working in time. If I had more time I would have saved the items of a game's moves stack as a .csv file at the end of a game. This could then be read by the program and the moves replayed back to the user.

4 Critical Evaluation

I think overall the program I have made and it's features works well. I am most happy with the player versus player game as it includes the undo feature and I have not been able to break it.

I was able to break the game whilst playing against the computer, if a user made an invalid move the computer would still take it's turn. I fixed this very last minute using a "goto" statement. This is usually bad practice as it can make code difficult to follow. If I had found the bug sooner I would have been able to spend the time properly fixing it.

I think that some of my code could have been altered to be more efficient, for example when the game checks for a win it does so by checking if any of the possible win combinations are present on the board. I was unable to work out a faster way of checking this but it felt more complicated than necessary. Luckily with a program this small it does not effect the user's experience.

The user interface, though basic, works well and is consistent making the game feel well structured. I have made sure instructions are clear to the user and the game board is centered with plenty of padding. It would have been nice to implement some basic animations to the UI, such as a brief "Thinking..." with an animated ellipsis before the computer makes a move. Currently when the computer takes a turn it is almost instant and it is not made clear to the user that it is now their turn again.

5 Personal Evaluation

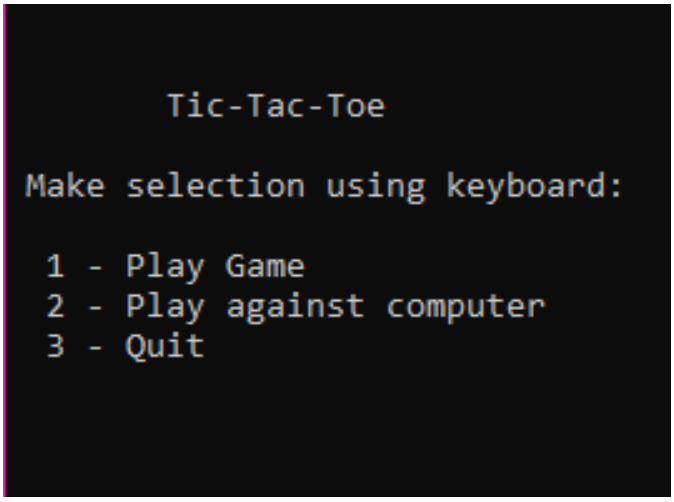
From this coursework I have learned how using different data structures can be easier to use for solving different problems. I was not very familiar with C beginning this module but I now feel more confident in my knowledge and feel I could easily go into a project using the same programming language.

Whenever I ran into problems I used print statements to debug. I asked friends to help test the game, asking them to try and break it in order to find bugs I may not have though of checking for. This worked well and was good for catching lots of small mistakes.

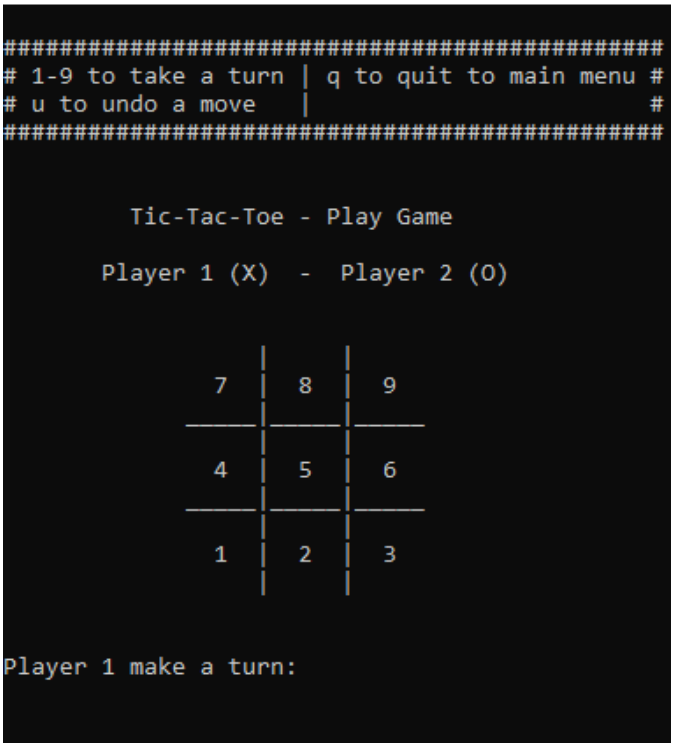
This was a fun coursework to attempt and overall I think I performed well. My program is working as intended, with no incomplete features and no way to crash it that I have found. Now that I am more familiar with some of the data structures I've used I feel I will be able to use them to solve problems in the future.

6 Appendices

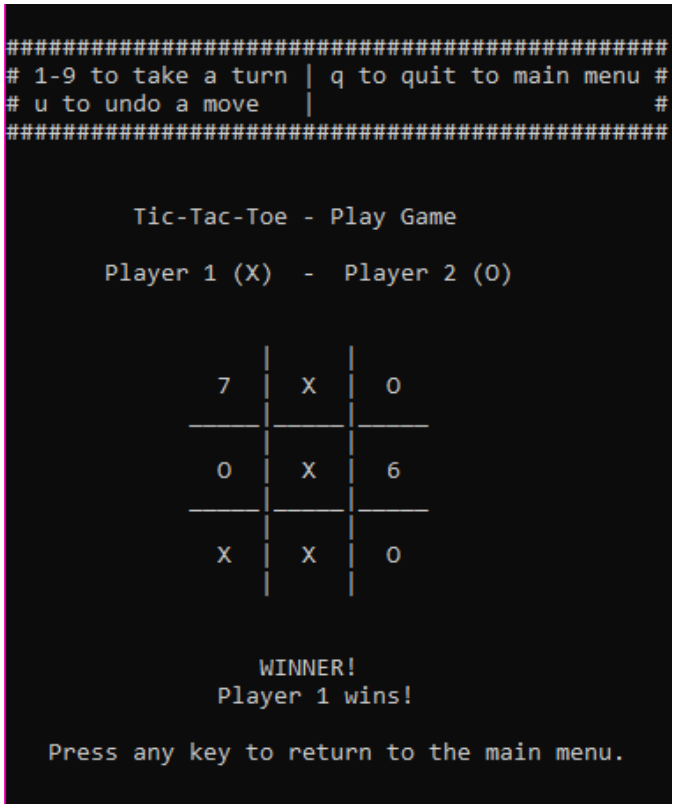
Appendix A - Main menu



Appendix B - In game view



Appendix C - Victory view



Appendix D - Difficulty selection view