Tic-Tac-Toe Coursework Report

Maria Luque Anguita
40280156@napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET08122)

1 Introduction

The aim of this coursework is to demonstrate my understanding of both theory and practise in relation to the content of the Algorithms and Data Structures module. The task is to implement a text-based Tic-Tac-Toe game using the C programming language paying special attention to the algorithms and structures used for it. For it I programmed a game where you can choose to play alone (versus the computer) or with another person, as well as saving your game to a file and continue playing from that file later on. The current situation of the game is stored in a char list and moves and undone moves are stored in two different stacks.

2 Design

2.1 Playing grid

Lists are the most simple and most commonly used data structure. For that reason I chose to store the playing grid in a list of chars. A list is an ordered sequence of vertices where associated with each vertex is a data item, a previous vertex, and a next vertex, except the first one and last one who have null previous and next vertices [1].

This list stores the numbers that basically show the user what the available moves are. Each number represents a square in the Tic Tac Toe grid (Figure 1). When a player makes a move, if it is a valid move, the selected number changes from that number to the player's mark (X or O).

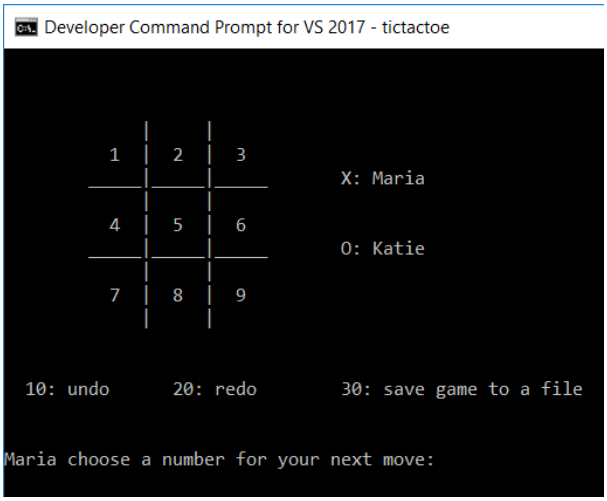

Figure 1: **Playing grid**

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

2.2 Moves

To check if the move played is valid I created a function which loops through the char list storing the grid to check if the number selected is still a number (if it is an X or an O it is an invalid movement).

```
1 int check_movement(int choice, char type)
2 {
3   for (int i = 1; i < 10; i++)
4   {
5     if ((choice + 48) == (int)numbers[i-1])
6     {
7       numbers[i-1] = type;
8       return 1; // valid
9     }
10  }
11  return 0; // invalid
12 }
```

In the function, choice (the number that the player entered) is of type int and the grid is stored as chars. To compare if an int and a char have the same ASCII value I add 48 to the choice, which gives the ASCII value of the number. For example, the ASCII value of number 1 is 049 and to get from 1 to 49 we need to add 48. Another way of doing it is to add the char 0 (1 + '0'), since 0 is 048 and it is the exact thing as just adding 48 to the int. If both the int entered and the char stored match, it means that the move is valid and the function returns 1. When the choice is not equal to the char stored it means that it has already been played and therefore the move is invalid (the function will return 0).

2.3 Undo and Redo

For the undo and redo functions I had to keep track of all the moves played. For this I chose stacks. A stack is a list with only the operations of inserting and deleting data items at one end (the "top") [2]. The first stack stores all of the original moves, and if the player chooses to undo a move, the number at the top of the stack goes into the second stack, and so on. If the player wants to redo the game, it was just a matter of doing the same thing but in the opposite way, from the second stack to the first "original-moves" stack. This way is easy, fast and efficient. Stacks have a LIFO structure (last in, first out) which allows this to be done, and are very handy for remembering state, doing something else that changes state and then reverting to the previous state, for instance CPU registers or for implementing undo/redo behaviour [3].

For the stacks to work I first declared the struct, and added the functions of initialising the stack, push and pop. Initialising a stack is done by setting the top pointer of the stack to -1. For the push and pop functions, the pseudocode is shown below:

if *top pointer == (max capacity of stack - 1)* **then**
  stack is full;
end
else
  top pointer++;
  place item into stack;
end

Algorithm 1: Push item to stack

if *top pointer == -1* **then**
  stack is empty and there is nothing to pop;
end
else
  access the value stored in the current location;
  alter the pointer to point to item below current item;
end

Algorithm 2: Pop item from stack

These two functions are the base of the stacks used to store, undo, redo all the moves. Every time a move is played, if it is valid, the number played is pushed into the first stack.

2.4 1v1 or against Computer

At the start of the game, the program asks the user if they want to play against the computer or against another player (Figure 2). If it is only one player, the program will then ask the player for their name, if it's two players, then it will ask for both names.

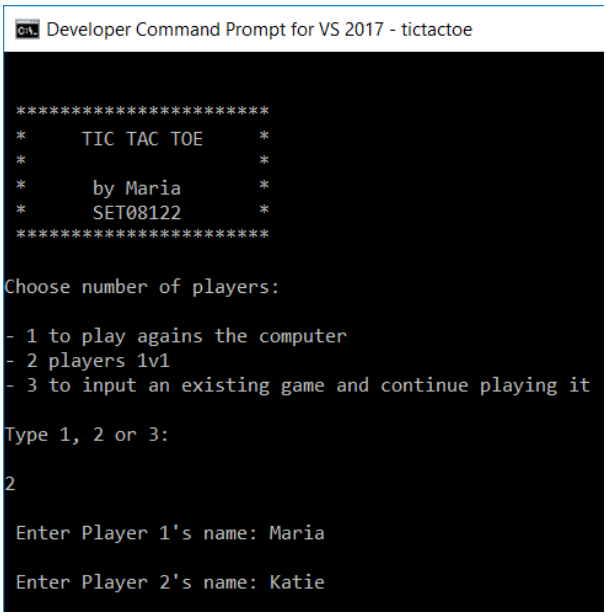

Figure 2: **Choosing number of players**

If the player chooses to play against the computer, the program goes through all the available spaces in the playing grid and chooses the next available one.

```
1 for (int a=1; a<10; a++)
2 {
3    if ((numbers[a-1] != 'X') && (numbers[a-1] != 'O'))
4    {
5       choice = a;
6       break;
7    }
8 }
```

This will make sure that the choice is always a valid move, since it doesn't choose a number that has an X or O, which means that that space has already been played.

2.5 Export games to file

The program has the option to save the current state of the game to a file manually (as seen in Figure 1, by choosing 30) and then reload it so the players can keep playing, as well as automatically saving all the games played when they end. The way I've done this is by saving the games in blocks of four lines in a text file. The first line contains the name of the first player, and line three contains the name of the second player. Line 2 has the first "original-moves" stack and line 4 the second "undone-moves" stack (this can sometimes be an empty line if there have been no undone moves). This allows the program to read the file and have the same players and moves as before, allowing the players to redo and undo all of the moves they did.

When a game is finished, the results are automatically appended to the games file ("game.txt"), keeping a history of all the games played. A finished game, with or without a winner, can also be loaded and movements can be undone so the output might be different. An example of the text file is shown below in Figure 3. Line 4 is empty because there are no undone moves, hence the second stack is empty.

Figure 3: **How games are saved**

To save the game state I used my function "void save_game()" that takes as parameters both player names and both stacks and saves them into the game.txt file. I used a for loop that goes through all the moves saved in the stacks and adds them to the document in a string format.

2.6 Importing games from file

As well as keeping a record of all the games played, finished or unfinished, the user has the option to open an already started game from the file and keep playing, undo movements or redo movements from that game. At the start, when the user is asked if they want to play alone, against someone else or play an existing game, if they choose the third option, all of the before played games will be displayed in the command line along with an ID number. The user then enters the

number of the game they want to continue playing and this game with the player names and current moves comes up. Figure 4 shows how the optional games to choose from are displayed.

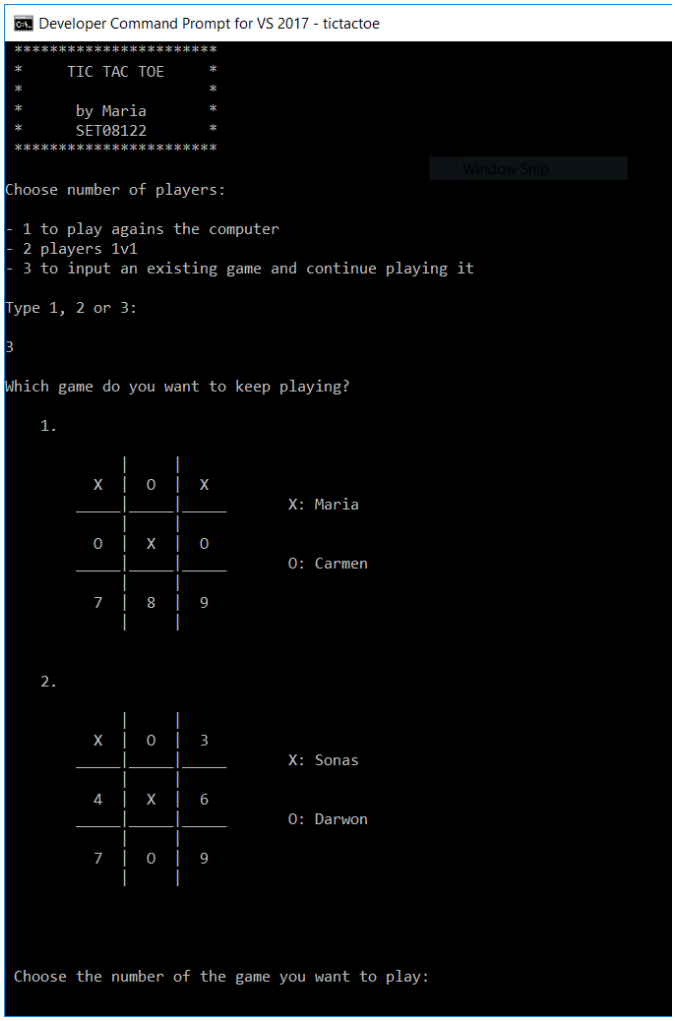

Figure 4: **Choosing an existing game**

When reading from the file, I use a do...while loop which initialises the stacks, playing grid and player number (1 or 2) every time a game block is read (if this didn't happen, all moves will just override each other and the last game will probably have most of the spaces played instead of the actual moves played), and displays the games and player names. I used fgets() since I am reading line by line, and the do...while loop stops when fgets() is NULL, which means all the lines of the file have been read.

The problem with reading the stacks is that since stacks are First In Last Out, the resulting string is the other way around, so I had to reverse the string and then use a for loop to go one by one and push them into the stacks. Of course, after the game has been read and displayed, the stacks had to be emptied again so the next game could use them, that is why I initialise them at the beginning of the loop.

Another problem that I encountered while using fgets() is that since it reads line by line into strings, they all get added a new line at the end, and for that I had to remove the last character of both names or ir would print the grid with a space in between and it would look weird.

To display the games in the command line I created a function (void print_games() - which displayes the grids shown in Figure 4) and this makes the program look more understandable, manageable and user friendly, the user can clearly see all of the games played and only has to choose the number of the game they want to continue playing.

2.7 Winner

To see who the winner is, I created a function (int winner()) that checks all possible winning combinations, first it checks if the vertical lines of the playing grid all have the same letter (X or O), then all the horizontal ones, and finally both diagonal ones. This function returns 3 possible outcomes: 1, 2 or 3, each representing a state of the game - there is a winner, the game has ended but there is no winner or they're still playing. I had to use a function (void check_end()) to determine that there is no winner because all the places have been played.

3 Enhancements

For this report we had more than enough time to implement many features. However, if I had to implement more I feel like it would be a good idea to be able to choose how big the users want the grid to be (3x3, 4x4, 5x5, 5x2, etc...) even though that would change the game to another one.

If the size of the grid is increased this means that there could be space for more players. And joining 3 squares in a bigger grid could make a player win.

The most important thing I would have improved is how the computer plays against the user. In this program it chooses the next available space and plays it, but if a Minimax algorithm was used it would have made it win a couple of times and would make it more difficult for the player to win, hence making the game more interesting.

The last thing I also thought about doing is a continuous game. This game will always have one space available and will allow the players to move one square at a time to finally always have a winner. This idea came from a game that I used to play as a kid online.

4 Critical evaluation

At the beginning I had to research the differences between stacks, queues and linked lists and see which one would be more effective for the undo and redo moves. Stacks were easy to use because it could be done as a linked list with a head pointer or as a dynamic array. The queue could have been normal or circular. The normal ones with the head a tail pointers would be easy to save the moves in order and it could have also been done as a circular buffer backed by an array.

The advantages of using a linked lists are that it is not necessary to choose a maximum list size in advance, as we do when we declare an array to hold a stack or a queue [1].

However, I used the activity in the workbook that measures the size of structs to see which one would occupy less space. A linked list's node occupies 8 bytes, while a stack struct with an array of 10 items occupies 44 bytes, which is half of what the linked list would occupy if there were 10 nodes. Linked lists would occupy less if there are less nodes but I figured that 9 is the maximum number of nodes I would need, therefore the stack was the struct to use.

To store the playing grid, I used a list of chars, since a char is 1 byte while ints are 4. This makes the program use less memory and therefore more efficient.

To critically analyse the program I read a couple of articles about PERT (program Evaluation and Review Technique). The main types of evaluation are quantitative and qualitative, but for this project, the most important one needed is qualitative. As Rebecca Gajda and Jennifer Jewiss said in their article [4], the main ways of finding qualitative data abour your program is by letting a third party use it. After asking a couple of people to give me feedback about my program, I added many of the functionalities they proposed and fixed any issues that they encountered. This allowed me to not have any bugs and see whether the program is making a difference, and if the participants have experienced success or problems.

5 Personal evaluation

This module and coursework helped me get into the hang of C again. It was a programming language that I never liked before but this coursework made me realise how easy C can be to perform certain tasks, and it is also very similar to the other languages we have learned in other modules, but since I learned C at the beginning and never liked it, I didn't realise how useful it was.

The lab workbooks have been a massive help for this. They are easy to follow, make us want to learn more by doing the tasks and came really useful for the coursework. It was never hard to follow each of them and since they teach us basic structures and algorithms used in every single program, they can be used anywhere. The idea of using stacks for this came mainly from the workbook activities and books recommended by the lecturers.

To better understand the structures I was going to use for my program I had to read about the different types explained in the books suggested in Moodle, which also provided me with a better understanding of C itself.

Before starting to program the game I had a look online and played around with a couple of examples [5] [6] [7] [8], from which I compared the way things were done and analyzed how useful those could be for my project.

With all the help provided (lecturers, lab workbooks, books and demonstrators) and the skills I have learned in this module about how to do research on my own and solve my own problems, I feel like now I can overcome any problem I have. This coursework opened my mind to the C programming language again and I am grateful for that and all I learned in this module.

Finally, I feel like I performed well in this coursework. I created an easy to follow Tic Tac Toe game which is effective and pretty.

References

[1] J. A. Storer, *Introduction to data structures and algorithms*. Birkhauser, 2013.

[2] K. Mehlhorn and P. Sanders, *Algorithms and data structures: the basic toolbox*. Springer, 2008.

[3] M. M. Raghuwanshi, *Algorithm and data structures*. Alpha Science International Ltd, 2016.

[4] "Practical assessment, research and evaluation (pare)," *Encyclopedia of Evaluation Encyclopedia of evaluation*.

[5] D. Singh, "Tictactoe game in c."

[6] J. Kripac, "An efficient undo/redo algorithm."

[7] GeeksForGeeks, "Difference between getc(), getchar(), getch() and getche()," Dec 2018.

[8] "Mini project in c tic tac toe game," Jul 2015.