Tic-Tac-Toe Game: A Console-based version using C Programming

Project Report of Group 2 by:

S M Uday Haider
North South University
Student ID: 2512062642
Department of Electrical & Computer Engineering

Kanij Fatema Lubna
North South University
Student ID: 2512430642
Department of Electrical & Computer Engineering

Asifur Rahman Apon
North South University
Student ID: 2512131642
Department of Electrical & Computer Engineering

Safin Ahmed
North South University
Student ID: 2511967642
Department of Electrical & Computer Engineering

Abstract

The goal of this project is to build a console based Tic-Tac-Toe game in C Program where two players can compete against each other. The key features of this project are: interchanging the moves, displaying the outcomes and declaring the result. The target is to create a user-friendly program. It is an ideal project to understand C programming in more diverse applications. This report outlines the efforts made, problems faced, and improvements expected in the future.

1. Introduction

Tic- Tac- Toe is a widely known game that was made around 1300 B.C. in ancient Egypt. It is played between two players in a 3x3 grid using the 'X' and 'O' symbol. There can be three cases of conclusion: win, loss and draw. Different fundamentals of C programming are used for making the game logic and establishing the conclusions. They are: arrays, loops, functions and conditionals. On the console, the players will be asked to input a number from 1 to 9 one after another. Each number is assigned for each block of the game and as an output, 'X' will be shown in the first player's preferred block and 'O' will be shown in the second player's preferred block.

2. Objective

- To deploy a fully effective 3×3 Tic-Tac-Toe game.
- For practicing and improving skills in the use of arrays and user inputs in C.
- For controlling game logic, win/draw conditions, and input verification.
- To maintain code clean and organized using functions.

3. Implementation Progress

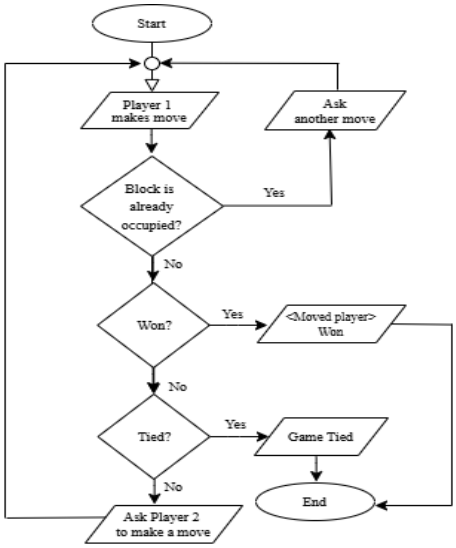
The following aspects are planned to executed:

- *Interchanging the moves:* The game board is represented by a 3x3 grid which is initialized with numbers (1-9). The board updates automatically after each valid move.
- *Displaying the outcome:* Players are asked to input their moves and the program will check if the position is already occupied or not.
- *Declaring the result:* The result will be declared as per the horizontal, vertical and diagonal blocks. The first player who will mark their signs accordingly wins the game.

4. Requirements

- *C Compiler:* GCC, Turbo C, Dev-C++.
- *Operating System:* Windows/Linux/MacOS.
- *IDE*: Code::Blocks, Visual Studio Code etc.
- A basic CPU with a small amount of RAM.

5. Flowchart



Start:
The game begins and shows a welcome message.

Create Game Board:
A 3x3 grid is prepared using numbers 1–9 to indicate available moves.

Show Current Board
The current state of the board is shown to the players.

Ask Current Player for a Move
The current player is asked to choose a cell number (1–9) to place their mark.

Check If Move is Valid
The program checks if the number is between 1 and 9 and the selected cell is not already taken. If the move is **invalid**, show an error and ask again (go back to Step 4).

Place the Mark
The selected cell is updated with the current player's symbol (either 'X' or 'O').

Check for a Win
The program checks if the current player has won by completing a row, column, or diagonal. If **yes**, display the board and show a win message → Go to **End**.

Check for a Draw
The program checks if all cells are filled and no one has won. If **yes**, display the board and show a draw message → Go to **End**.

Change Turn
The other player is set to play next. Go back to **Step 3** to continue the game.

End
The game ends with either a win or a draw message.

6. Methodology

The project was completed by employing a modular approach:

Board Setup:
A 3×3 matrix was created using the numbers 1–9 to occupy cell positions.

Input from Players:
A loop prompts each player for input. The input is considered to ensure it falls within the range and the brick is not already occupied.

Logic in Games:
After every valid input, the computer confirms the winning combinations (rows, columns, diagonals). If there is no winner when all cells are occupied , then the result is a draw.

Input Validation:
Unavailable or duplicated inputs are tackled politely to handle unfairness and program crashes.

7. Code Structure

The full completed code is given below:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define SIZE 3

char board[SIZE][SIZE];

void clearScreen()
{
     system("clear"); // use "cls" for
Windows
}
```

```
void printColor(const char *text,
const char *colorCode)
{
      printf("%s%s\033[0m", colorCode,
text);
}
```

Board Initialization:

```
void initializeBoard()
{
    int num = 1;
    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++)
            board[i][j] = num++ + '0';
}
```

Displaying Board:

```
void displayBoard()
{
     printf("\n   \033[1;37m    Tic Tac
Toe Arena 🎮\033[0m\n\n");
    for (int i = 0; i < SIZE; i++)
    {
        printf("     ");
        for (int j = 0; j < SIZE; j++)
        {
            char mark = board[i][j];
            if (mark == 'X')
                    printColor(" X ",
"\033[1;31m");
            else if (mark == 'O')
                    printColor(" O ",
"\033[1;34m");
            else
            {
                   char buf[4] = {' ',
mark, ' ', '\0'};
                      printColor(buf,
"\033[1;33m");
            }

            if (j < SIZE - 1)
                printf("\033[1;37m│");
        }
        printf("\n");
        if (i < SIZE - 1)
                                printf("
──┼───┼──\n");
    }
    printf("\n");
}
```

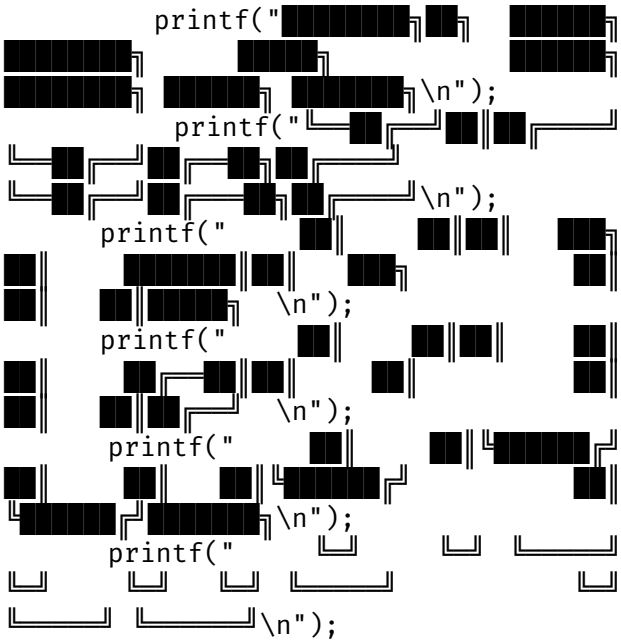
Winning Condition:

```
int checkWin()
{
    for (int i = 0; i < SIZE; i++)
    {
         if (board[i][0] == board[i][1]
&& board[i][1] == board[i][2])
            return 1;
         if (board[0][i] == board[1][i]
&& board[1][i] == board[2][i])
            return 1;
    }
     if (board[0][0] == board[1][1] &&
board[1][1] == board[2][2])
        return 1;
     if (board[0][2] == board[1][1] &&
board[1][1] == board[2][0])
        return 1;
    return 0;
}

int isDraw()
{
    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++)
             if (board[i][j] != 'X' &&
board[i][j] != 'O')
                return 0;
    return 1;
}
```

Main Game Loop:

```
void intro()
{
    clearScreen();
    srand(time(0));
```

```
    printf("\033[1;35m");
          printf("

\n");
    printf("\033[0m\n");

    const char *quotes[] = {
        "Think fast 🧠",
        "No mercy today 🔥",
         "Can you outsmart your rival? 👀",
         "Let’s play a classic… with style 😎",
         "This is not just a game. It’s WAR. ⚔️"};
    int q = rand() % 5;
    printf("💬 %s\n", quotes[q]);

    printf("⏳ Loading");
    for (int i = 0; i < 3; i++)
    {
        fflush(stdout);
        sleep(1);
        printf(".");
    }
    printf("\n\n");
}

void dropAnimation(int row, int col, char mark)
{
    char temp;
    for (int r = 0; r <= row; r++)
    {
        temp = board[r][col];
        board[r][col] = mark;
        clearScreen();
            printf("🎯 Dropping your move...\n");
        displayBoard();
        usleep(150000);
        board[r][col] = temp;
    }
    board[row][col] = mark;
}

void fireworkFrame(const char *frame[], int lines)
{
    for (int i = 0; i < lines; i++)
    {

printf("\033[1;36m%s\033[0m\n", frame[i]);
    }
}

void victoryFireworks()
{
    const char *frames[4][7] = {
        {"     *     ",
         "    ***    ",
         "   *****   ",
         "  *******  ",
         "   *****   ",
         "    ***    ",
         "     *     "},
        {"    \\|/    ",
         "  -- * --  ",
         "    /|\\    ",
         "     |     ",
         "   * | *   ",
         "    / \\    ",
         " 🎆 WOW!   "},
        {" \\   |   / ",
         "   \\ | /   ",
         "---  *  ---",
         "   / | \\   ",
         " /   |   \\ ",
         "             ",
         " 🎇 BOOM!  "},
        {"   *   *   ",
         "  * * * *  ",
         " *   *   * ",
         "*    *    *",
```

```
        " *   *   * ",
        "  *     *  ",
        "✨ KABOOM! "}};

    for (int i = 0; i < 8; i++)
    {
        clearScreen();
          fireworkFrame(frames[i % 4],
7);
        usleep(300000);
    }

    printf("\n");
}

void playGame()
{
    int player = 1, choice, row, col;
    char mark;

    initializeBoard();

    while (1)
    {
        clearScreen();
        printf("🎯 Current Board:\n");
        displayBoard();

         mark = (player == 1) ? 'X' :
'O';
          printf("👤 \033[1;36mPlayer
%d\033[0m (%c), enter a number (1-9):
", player, mark);
        scanf("%d", &choice);

        if (choice < 1 || choice > 9)
        {
                printColor("❌ Invalid
move! Try again...\n", "\033[1;31m");
            sleep(1);
            continue;
        }

        row = (choice - 1) / SIZE;
        col = (choice - 1) % SIZE;

         if (board[row][col] == 'X' ||
board[row][col] == 'O')
        {
            printColor("⚠️ Spot taken!
Try again.\n", "\033[1;33m");
            sleep(1);
            continue;
        }

        dropAnimation(row, col, mark);

        if (checkWin())
        {
            clearScreen();
            displayBoard();
                                printf("🏆
\033[1;32mPlayer    %d    WINS!\033[0m
🎉🎉🎉\n", player);
            victoryFireworks();
                                printf("🏆
\033[1;32mPlayer  %d  WINS!  GG!\033[0m
🎉🎉🎉\n", player);
                                printf("🔥
\033[1;33mLegend                status
unlocked.\033[0m\n");
            break;
        }

        if (isDraw())
        {
            clearScreen();
            displayBoard();
             printColor("🤝 It's a tie!
Respect. 🤜🤛\n", "\033[1;35m");
            break;
        }

          player = (player == 1) ? 2 :
1;
    }
}
```

Main Function:

```
int main()
{
    intro();
    playGame();
    return 0;
}
```

8. Issues and Solutions

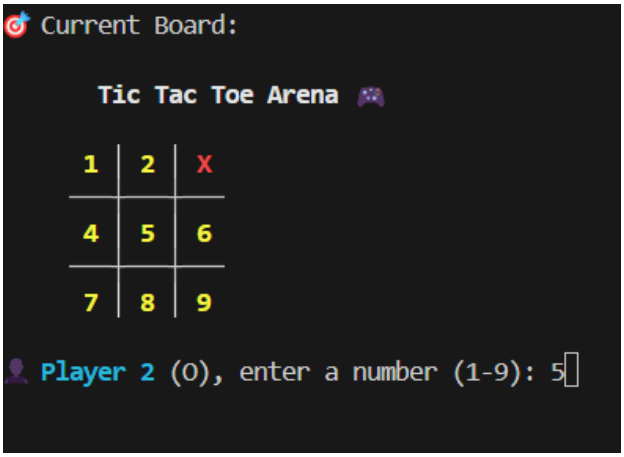
Issues: This project validates inputs to allow only numbers 1 to 9 and prevent infinite loops. Turns switch only after a valid move, ensuring players don't lose turns due to invalid inputs. For a smooth game, input errors have to be handled accurately. The game should only terminate when a player wins or all blocks are filled.

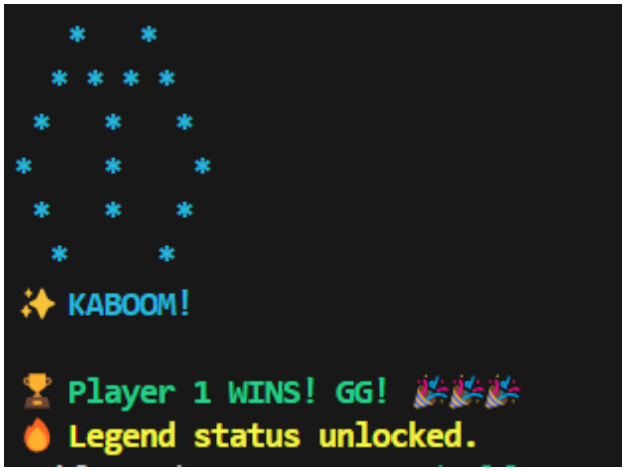
Solutions: This program evaluates the input if it is a number between 1-9 and asks the players again if it is not valid. Turns will switch only after a valid move. Therefore, opting an occupied block or inputting a wrong input won't change turns. Error management should prevent infinite loops, allowing the game to run smoothly and terminate properly when someone wins or cells are filled.

9. Testing and Result

The game was examined through various conditions:

- Win by row, column, and diagonal
- All draw situations Invalid inputs (e.g., 0, 10, letters)
- Repeated cell selection
- The game accurately responded to each test case, dealing unexpected situations elegantly.





10. Possibilities

- *Development Approach:* The game is designed with a step-by-step approach, starting with board representation, where the playing grid is set up and displayed. The game flow ensures smooth turn-based interaction between two players. Proper input handling is essential to validate moves and prevent errors. Finally, conclusion checking helps determine the winner or if the game results in a draw.

- *Future Enhancement:* There are several ways to improve the game. Implementing AI would allow single-player mode against a computer opponent. Adding a graphical interface would make the game more user-friendly. Another potential upgrade is enabling network-based multiplayer, allowing players to compete online.

11. Educational Impact

Working on the project gave students a deeper understanding and appreciation of programming logic and structure. Having the ability to construct a fully functional and complete game allowed them to hone valuable skills such as problem-solving, algorithmic design, and software design.

12. Modularity and Reusability

- During the duration of the project, several valuable lessons came to light: >Validate user input to prevent bugs and unwanted behavior at all times. >Modular code makes future changes and debugging much easier.
- Preplanning the game progression guarantees maintaining a clean design.
- Regular testing during development guarantees finding issues early.

13. User Experience and Interface

The game was designed with modularity by splitting the logic into smaller, manageable functions. This makes the code readable, easier to debug, and also easier to enhance. It also allows for portions of the code—such as input processing or presentation logic—to be reused in future projects.

14. Lessons Learned

Special attention was paid to creating an easy-to-use interface. The board layout is simple to read and uncluttered, and prompts guide the player effectively through the game. Such careful design allows for a smooth and pleasant experience for the player.

15. Conclusion

The C-based console Tic-Tac-Toe game allows two players to play interactively. It is fun, engaging, and competitive. The game sets up the board, manages inputs, and checks for win conditions. It prevents invalid moves and malpractices.This project covers loops, conditionals, arrays, and user input in C programming.It emphasizes error handling and user experience.

16. References

1. C: The Complete Reference - H. Schildt.
2. Problem Solving and Program Design in C - J. Hanly and E. Koffman.
3. C: How to Program - Deitel & Deitel.