TIC TAC TOE GAME IN C

Computer Course Project (CCP)

NED UNIVERSITY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT: BCIT

CLASS: FSCS-D

COURSE INSTRUCTOR: Muhammad Abdullah

GROUP MEMBERS:

Maryum Mairaj (CT-170)

Abdul Raheem Sheikh (CT-192)

Taha Ahmed Mallick (CT-183)

Table of Contents:

Abstract

The **Tic Tac Toe Game** project is a console-based two-player board game implemented in the **C programming language**. It focuses on the application of fundamental programming concepts such as arrays, conditional statements, looping constructs, user input handling, and modular programming using functions.

The purpose of this project is to simulate a classic game while reinforcing coding logic and problem-solving skills. The program displays a 3×3 grid where two players alternately mark spaces with **'X'** and **'O'**, aiming to form a straight line of three identical marks horizontally, vertically, or diagonally.

This report discusses the **development process**, including **problem analysis, algorithm design, implementation**, and **testing**. It also includes a **flowchart**, **code breakdown**, and **detailed discussion of challenges faced during development** and how they were resolved.

Introduction

Tic Tac Toe, also known as *Noughts and Crosses*, is one of the simplest and most recognized logical games. It has been widely used to teach **strategic reasoning** and **programming fundamentals**.

In this CCP project, we have implemented Tic Tac Toe using the **C language**, emphasizing clear logic, structured code, and a friendly console interface. The program allows two human players to compete, handles invalid inputs, announces the winner or a draw, and even offers a replay option.

The project serves as a foundation for understanding **algorithmic thinking, input validation, modularity, and flow control** — essential principles for any aspiring programmer.

Objectives

The main objectives of this project include:

1. To design a **functional, interactive Tic Tac Toe game** using core C programming concepts.
2. To apply the principles of **arrays, loops, conditional statements, and modular design**.
3. To ensure robust **input validation** to avoid unexpected program crashes.
4. To develop an **intuitive and color-coded** console interface using ANSI escape sequences.
5. To encourage teamwork, logical reasoning, and debugging practice.
6. To provide replay capability without restarting the program.

Problem Statement

The task was to design and implement a **two-player Tic Tac Toe game** that could run in a terminal window and follow standard game rules.

- The board should be displayed clearly after every move.
- Two players should take alternate turns entering their desired positions.
- The program must validate inputs to prevent overwriting of existing cells.
- The game should detect **win**, **draw**, or **ongoing** states.
- Players should be prompted whether they want to play again after a match concludes.
- The interface should be neat, interactive, and easy to understand.

Program Design and Logic

1. Data Structures Used

Character Array (board[9])
The game grid is represented by a single-dimensional array of 9 elements corresponding to positions 1–9.
Example:

- `Index: 0 1 2 3 4 5 6 7 8`
- `Value: 1 2 3 4 5 6 7 8 9`

When a player enters position 5, for example, element `board[4]` becomes `'X'` or `'O'`.

Winning Combination Array (win_lines[8][3])
This stores all 8 possible winning conditions in the form of index combinations:

- `{0,1,2}, {3,4,5}, {6,7,8},   // rows`
- `{0,3,6}, {1,4,7}, {2,5,8},   // columns`
- `{0,4,8}, {2,4,6}             // diagonals`

2. Program Flow and Game Loop

- The game starts by initializing variables and displaying the empty board.
- A **while loop** runs indefinitely until the player decides to exit. Inside this loop:
  - The current board is displayed.
  - The current player (Player 1 or 2) is prompted for a position input.
  - Input validation checks for non-numeric, out-of-range, or already filled positions.
  - The chosen box is updated with `'X'` or `'O'`.
  - The `check_win()` function determines the result of the game:
    - Returns **1** if a player wins.
    - Returns **-1** if the board is full (draw).
    - Returns **0** if the game continues.
  - After a win/draw, the program offers the user a choice to replay or exit.

3. Function Breakdown

- **`print_board(int status)`**
  Displays the grid with color-coded marks. The `status` argument determines whether to highlight the winning line.
  Uses `system("cls")` for a clean screen refresh.
- **`check_win()`**
  Iterates through all 8 winning combinations to determine if any match is found.
  Also counts filled boxes to detect draws.

4. Validation Logic

The program prevents:

- Characters or strings as inputs.
- Numbers outside the 1–9 range.
- Selecting already occupied positions.

This ensures smooth gameplay and eliminates runtime errors.

5. Console Enhancements

- Color-coded text using **ANSI escape sequences**:
  - Player 1: **Red (X)**
  - Player 2: **Green (O)**
  - Winning line: **Blue**
- Clean display using box-drawing characters (│, ─, ┼) for grid alignment.

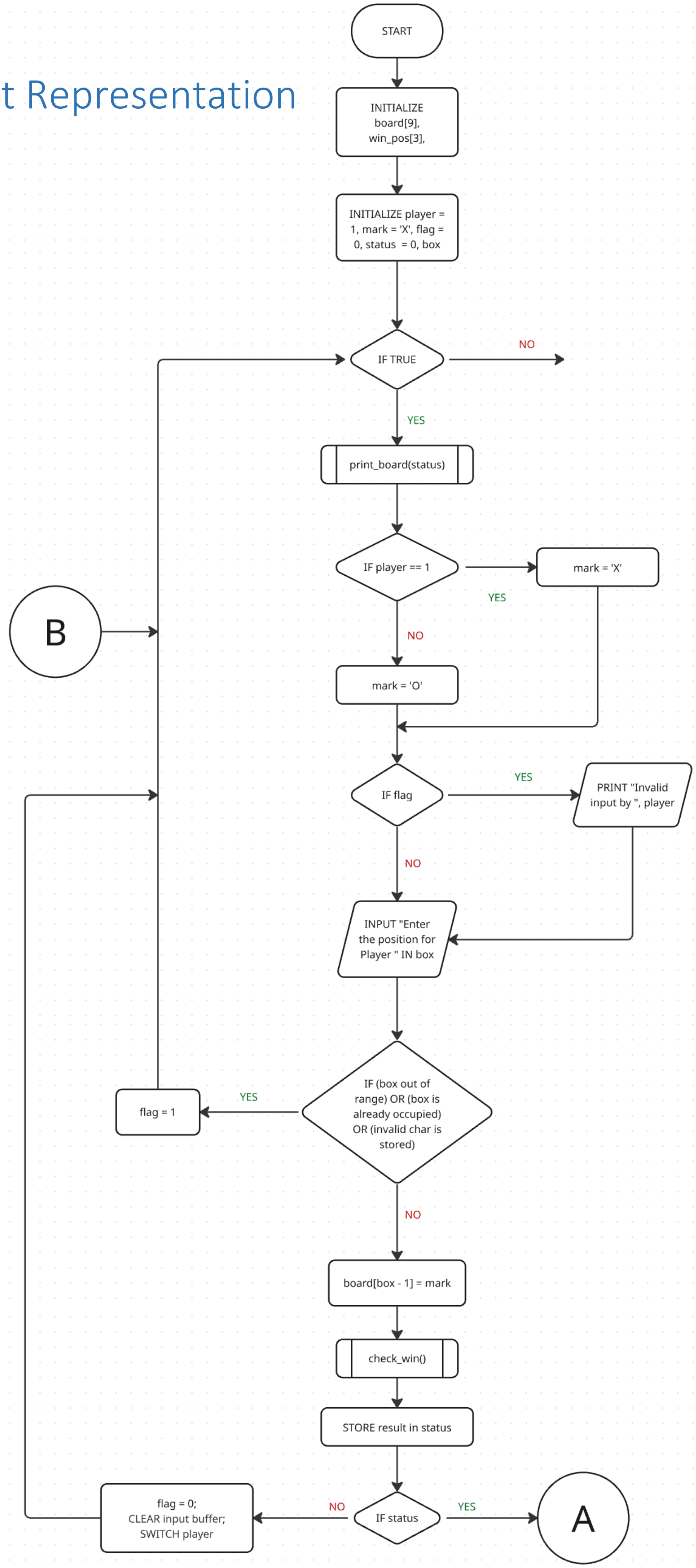
6. Replay System

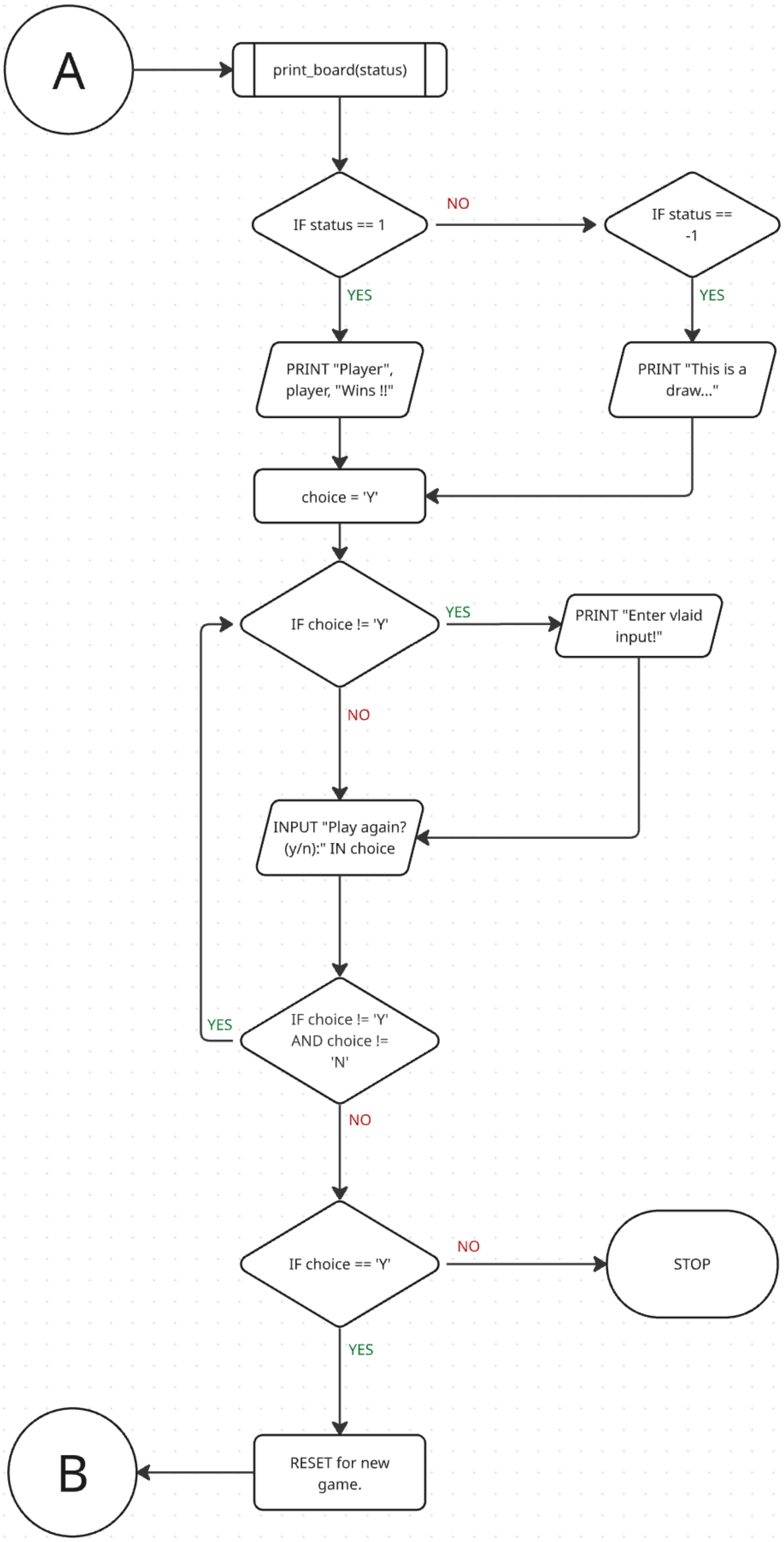
After each match, players are prompted:

```
Play again? (y/n):
```

If “y” is selected, the board resets automatically, allowing a new round without restarting the program.

Flowchart Representation


START
INITIALIZE board[9], win_pos[3],
INITIALIZE player = 1, mark = 'X', flag = 0, status = 0, box
IF TRUE
NO
YES
print_board(status)
IF player == 1
YES
mark = 'X'
NO
mark = 'O'
B
IF flag
YES
PRINT "Invalid input by ", player
NO
INPUT "Enter the position for Player " IN box
IF (box out of range) OR (box is already occupied) OR (invalid char is stored)
YES
flag = 1
NO
board[box - 1] = mark
check_win()
STORE result in status
IF status
NO
flag = 0; CLEAR input buffer; SWITCH player
YES
A


A
print_board(status)
IF status == 1
NO
IF status == -1
YES
YES
PRINT "Player", player, "Wins !!"
PRINT "This is a draw..."
choice = 'Y'
IF choice != 'Y'
YES
PRINT "Enter vlaid input!"
NO
INPUT "Play again? (y/n):" IN choice
YES
IF choice != 'Y' AND choice != 'N'
NO
IF choice == 'Y'
NO
STOP
YES
RESET for new game.
B

Difficulties Faced and Their Resolutions

Throughout the development of the **Tic Tac Toe Game in C**, several challenges were encountered that tested both logic and debugging skills. Each issue provided an opportunity to understand the inner workings of C programming more deeply and strengthen the overall structure of the project. The following sections describe the key problems faced, their causes, and how they were effectively resolved.

1. Input Validation Errors

One of the earliest challenges was dealing with invalid player inputs. When a user entered characters or symbols instead of numbers, the program either entered an infinite loop or crashed unexpectedly. This occurred because the `scanf()` function attempted to read an integer but encountered incompatible data types, leaving unread characters in the input buffer.

To solve this, input validation was implemented using:

```
if (scanf("%d", &box) != 1)
```

and a buffer-clearing statement:

```
while (getchar() != '\n');
```

This ensured that any invalid input was discarded safely before the program accepted new data. As a result, the game became much more stable and user-friendly.

2. Overwriting of Player Moves

At first, players were able to overwrite each other's moves, which broke the fairness of the game. This happened because there was no condition preventing users from selecting a position that was already filled.

The solution was to add a validation check:

```
if (box < 1 || box > 9 || board[box - 1] != box + '0')
```

This ensured that each move could only be placed in an unoccupied box. Once a cell contained an 'X' or 'O', the program disallowed any further changes to it, thus enforcing proper game rules.

3. Incorrect Win Detection

Another major difficulty was the program’s failure to correctly detect diagonal victories. Initially, the game recognized horizontal and vertical wins but sometimes ignored diagonal ones.

This was due to a **mismatch between logical indices and their displayed positions**. After carefully analyzing the board layout, the formula:

```
n = j + i * 3
```

was applied to maintain consistent mapping between the 1D array indices and the 3×3 grid. This correction ensured that all eight winning combinations — three rows, three columns, and two diagonals — were properly detected.

4. Draw Condition Not Recognized

In early tests, when all boxes were filled without any winner, the program continued prompting players for moves indefinitely. This occurred because there was no condition to check whether the grid was full.

The solution involved counting the number of filled cells within the `check_win()` function. When all nine boxes contained either ‘X’ or ‘O’, and no winning combination was found, the function returned `-1`, indicating a draw. This addition made the game logic complete and realistic.

5. Replay Feature Malfunction

While adding the replay feature, an error was observed where the board displayed mixed symbols from the previous match. This happened because the board array was not being reinitialized properly before a new round began.

To fix this, a reset loop was added:

```
for (int i = 0; i < 9; i++)
    board[i] = i + '1';
```

This restored the board to its default numbered state, allowing the players to start fresh each time without restarting the entire program.

6. Unicode character Compatibility

By default windows consoles aren't compatible with UTF-8 encoding. We used Unicode characters to print a neat grid. To display it properly onto windows terminals we first need to enable it which is done using the lines below.

```
#ifdef _WIN32
    SetConsoleOutputCP(CP_UTF8);
#endif
```

This line enabled UTF-8 output on Windows systems, making the interface work universally across different platforms.

7. Optimization of check_win() function

In check_win() function we were initially writing an if condition for every combinations, which means total 8 else if ladder.

```
if (board[pos1] == board[pos2] && board[pos2] == board[pos3])//1st row
      return 1;
else if (board[2] == board[4] && board[4] == board[6])//lft dig
      return 1;
```

Then we optimized our approach by using arrays and loops. First we defined the winning combinations in a global win_lines array.

```
int win_lines[8][3] = {
    {0, 1, 2}, // 1st row
    {3, 4, 5}, // 2nd row
    {6, 7, 8}, // 3rd row
    {0, 3, 6}, // 1st col
    {1, 4, 7}, // 2nd col
    {2, 5, 8}, // 3rd col
    {0, 4, 8}, // rht dig
    {2, 4, 6}  // lft dig
};
```

And used a loop to traverse through all combinations and check if any one of them exists. If a winning combination is found it returns 1.

```
for (int i = 0; i < 8; i++)
    {
        int pos1 = win_lines[i][0];
        int pos2 = win_lines[i][1];
        int pos3 = win_lines[i][2];
        if (board[pos1] == board[pos2] && board[pos2] == board[pos3])
        {
            win_pos[0] = pos1;
            win_pos[1] = pos2;
            win_pos[2] = pos3;
            return 1;
        }
    }
```

8. Optimization of print_board() function

In print_board() function we were initially printing the tic tac toe grid using static print statements instead of using a loop. Initially this was not a problem but to incorporate color coded symbols and win positions we had to add checks using if statements to see which color to code the symbol at each cell, and if we had continued our same approach then the code was getting too repetitive as shown below.

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

// --- Cell 0 ---
if (board[0] == 'X')
    printf("\033[31m");
else if (board[0] == 'O')
    printf("\033[32m");
if (status && (win_pos[0] == 0 || win_pos[1] == 0 || win_pos[2] == 0))
    printf("\033[4;34m");
printf("%c\033[0m\033[1m │ ", board[0]);

// --- Cell 1 ---
if (board[1] == 'X')
    printf("\033[31m");
else if (board[1] == 'O')
    printf("\033[32m");
if (status && (win_pos[0] == 1 || win_pos[1] == 1 || win_pos[2] == 1))
    printf("\033[4;34m");
printf("%c\033[0m\033[1m │ ", board[1]);

// --- Cell 2 ---
if (board[2] == 'X')
    printf("\033[31m");
else if (board[2] == 'O')
    printf("\033[32m");
if (status && (win_pos[0] == 2 || win_pos[1] == 2 || win_pos[2] == 2))
    printf("\033[4;34m");
printf("%c\033[0m\033[1m", board[2]);

printf("\n\t   ───┼───┼───\n\t    ");

// ...
// The same pattern continues for board[3], board[4], board[5], etc.
// Each cell would require its own repeated color + underline checks.
```

```
    // ...
    // --- Cell 8 ---
    if (board[8] == 'X')
        printf("\033[31m");
    else if (board[8] == 'O')
        printf("\033[32m");
    if (status && (win_pos[0] == 8 || win_pos[1] == 8 || win_pos[2] == 8))
        printf("\033[4;34m");
    printf("%c\033[0m\033[1m", board[8]);

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

To solve this issue we made the use of loops which significantly reduced the lines used to print the grid, as shown below.

```
for (int i = 0; i < 3; i++)
    {
        printf("\033[1m\n\t    ");
        for (int j = 0; j < 3; j++)
        {
            char mark = board[j + i * 3];
            if (mark == 'X')
                printf("\033[31m");
            else if (mark == 'O')
                printf("\033[32m");
            if (status)
                for (int k = 0; k < 3; k++)
                    if (win_pos[k] == j + i * 3)
                        printf("\033[4;34m");
            printf("%c\033[0m\033[1m", mark);
            j != 2 ? printf(" │ ") : 0;
        }
        i != 2 ? printf("\n\t   ───┼───┼───") : 0;
    }
```

Lessons Learned

Overcoming these challenges taught valuable lessons about programming discipline and systematic debugging. The process reinforced several key principles:

1. **Input Validation is Essential** – Never trust user input; always verify data types and boundaries.
2. **Modular Code Design** – Keeping related logic within functions like `print_board()` or `check_win()` improves readability and maintainability.
3. **Logical Consistency** – Always map visual representation and data structure correctly to avoid index mismatches.
4. **Cross-Platform Awareness** – Programs should be tested across environments to ensure consistent behavior.
5. **Incremental Debugging** – Testing one feature at a time helps isolate bugs and leads to efficient troubleshooting.

In conclusion, every difficulty faced during the development process contributed to stronger problem-solving ability, deeper command over C syntax, and a practical understanding of structured programming. These lessons not only improved the project's final outcome but also provided foundational skills for tackling more advanced software development challenges in the future.

Output, Conclusion, and Future Scope

Sample Output

```
┌─────────────────┐
│ TIC TAC TOE GAME│
└─────────────────┘
Player 1: X (RED)
Player 2: O (GREEN)

    1 │ 2 │ 3
   ───┼───┼───
    4 │ 5 │ 6
   ───┼───┼───
    7 │ 8 │ 9

Player 1 [X], enter position (1-9): 5
Player 2 [O], enter position (1-9): 1
...
Player 1 Wins!!
Play again? (y/n): y
```

Conclusion

The Tic Tac Toe Game project successfully demonstrates **logical reasoning, function-based programming, and clean console design**. Through iterative testing and debugging, a fully functional and user-friendly game was developed. The project deepened understanding of **C fundamentals**, including arrays, conditionals, loops, and function calls.

Future Scope

1. **Single Player Mode (AI Opponent)** — using Minimax or random algorithms.
2. **Graphical User Interface (GUI)** — using C graphics or external libraries.
3. **Scoreboard System** — to keep track of multiple rounds.
4. **Online Multiplayer Support** — via network programming concepts.
5. **Super Tic Tac Toe** — advance version of super tic tac toe