FOCP PROJECT
NOUGHTS AND CROSSES

Author – Karthik Raman Keerangudi Kalyanaraman

1. Task Description

Assignment 18 requires the development of a Noughts and Crosses (Tic-Tac-Toe) game implemented in the C++ programming language. The program must support the traditional 3×3 grid version of the game as well as a more advanced version using a larger grid of up to 19×19 cells, where the winning condition is achieved by placing five symbols consecutively in a row.

The objective of the task is to design a program that allows players to interact with the game through a console-based interface. The game must correctly handle player input, display the current state of the board, and determine game outcomes such as a win or a draw based on the defined rules. For larger board sizes, the program should dynamically adapt the winning condition to ensure meaningful gameplay.

The assignment also emphasizes the application of fundamental programming concepts, including array or vector-based data structures, conditional logic, iterative control structures, and algorithmic decision-making. The completed program should be robust, user-friendly, and capable of handling different board sizes while maintaining correct game logic and performance.

2. Analysis of the Task

The primary challenge of this task lies in extending the traditional Noughts and Crosses game beyond a fixed 3×3 grid while preserving correct and efficient game logic. Unlike the standard version, which has a small and finite number of winning combinations, larger grids such as 19×19 significantly increase the complexity of win detection and board management.

One of the key considerations in analyzing the task was determining how to represent the game board in a flexible manner. Since the board size is not fixed, the program must dynamically allocate and manage a two-dimensional grid based on user input. This requirement necessitates the use of dynamic data structures rather than static arrays.

Another important aspect of the analysis involved defining adaptable winning conditions. While the 3×3 version requires three symbols in a row to win, larger grids require a longer sequence (up to five symbols) to maintain balanced gameplay. The program therefore needs to adjust the win condition based on the selected board size and verify winning sequences in multiple directions, including horizontal, vertical, and diagonal orientations.

User interaction and input validation also form a significant part of the task analysis. The program must allow players to input moves in an intuitive manner, prevent invalid or out-of-range inputs, and ensure that moves cannot overwrite previously occupied cells.

Additionally, the game flow must support restarting or exiting the program at appropriate points.

For the single-player mode, an additional layer of complexity arises in implementing computer-controlled opponents. The task requires the analysis of simple artificial intelligence strategies, ranging from random move selection to more structured approaches that can detect immediate winning or blocking opportunities. These strategies must integrate seamlessly with the core game logic without compromising correctness.

Overall, the task demands a careful balance between flexibility, correctness, and usability. The analysis phase focused on identifying these challenges and selecting design approaches that would allow the program to scale from a simple 3×3 game to a more complex 19×19 version while remaining efficient and easy to use.

2.1 Theoretical Algorithm Description

The solution to the Noughts and Crosses problem follows a structured, step-by-step algorithm designed to support variable board sizes and dynamic winning conditions.

1. Game Initialization

At the start of the program, the user selects the game mode (two-player or single-player) and specifies the desired board size. Based on the selected board size, the program determines the required number of consecutive symbols needed to win the game. The game board is then initialized as an empty grid of the chosen dimensions.

2. Board Representation

The game board is represented as a two-dimensional structure where each cell corresponds to a position on the grid. Initially, all cells are marked as empty. Each move updates exactly one cell with the current player's symbol.

3. Turn Management

The game operates in alternating turns. The first player begins the game, after which control switches between players (or between the player and the automated system in single-player mode). The program ensures that only the current player is allowed to make a move at any given time.

4. Input Handling and Validation

On each turn, the program prompts the active player to select a cell. The input is validated to ensure that:

- the selected position lies within the bounds of the board,
- the selected cell is not already occupied,
- special commands such as restarting or exiting the game

are handled correctly.

Invalid inputs are rejected, and the player is prompted again without altering the game state.

5. Move Execution

Once a valid move is provided, the corresponding cell on the board is updated with the player's symbol. The updated board is then displayed to reflect the current state of the game.

6. Win Detection

After each move, the program checks whether the current player has achieved a winning condition. This is done by examining sequences of cells starting from the most recent move and extending in all relevant directions:

- horizontal,
- vertical,
- diagonal (both directions).

A win is declared if the required number of consecutive matching symbols is found without interruption.

7. Draw Detection

If no winning condition is detected, the program checks whether the board is completely filled. If all cells are occupied and no player has won, the game is declared a draw.

8. Automated Decision Logic (Single-Player Mode)

In single-player mode, the computer's moves are generated using automated decision logic based on the selected difficulty level:

- Easy: selects a random available position.
- Medium: prioritizes immediate winning moves or blocking the opponent's potential winning move.
- Hard: applies heuristic-based automation, such as prioritizing central board positions before falling back to medium-level decision logic.

3. Internal Specifications

This section describes the internal design and implementation of the Noughts and Crosses program. The program is implemented in C++ (compatible with C++20) and follows a procedural, modular structure. All variables, constants, functions, and control flows used in the program are explicitly defined and explained in relation to their role in the system.

The implementation resides in a single source file (main.cpp) and is divided logically into configuration, state management, rendering, game logic, automated decision logic, and control flow.

3.1 Programming Language, Standard, and Platform

- Programming Language: C++
- Language Standard: C++20–compatible
- Execution Platform: Microsoft Windows

The program makes use of standard C++ libraries for data structures, algorithms, and input/output, along with the Windows API (windows.h) for console text coloring. As a result, colored output is Windows-specific.

3.2 Header Files and Their Purpose

The following header files are included:

- <iostream> – console input and output using cin and cout
- <vector> – dynamic storage for the game board and coordinate lists
- <random> – random number generation for automated moves
- <utility> – use of std::pair for storing board coordinates
- <algorithm> – use of std::any_of for searching collections

- <string> – conversion of integers to strings for board display
- <windows.h> – access to Windows console functions for color output

Each included header directly supports functionality used in the program; no unused headers are present.

3.3 Global Constants

The program defines several global constants to manage console text colors:

```
constexpr WORD COLOR_DEFAULT = FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE;
constexpr WORD COLOR_RED     = FOREGROUND_RED | FOREGROUND_INTENSITY;
constexpr WORD COLOR_BLUE    = FOREGROUND_BLUE | FOREGROUND_INTENSITY;
constexpr WORD COLOR_GREEN   = FOREGROUND_GREEN | FOREGROUND_INTENSITY;
```

- COLOR_DEFAULT represents the normal console color.
- COLOR_RED is used to display player X.
- COLOR_BLUE is used to display player O.
- COLOR_GREEN is used to highlight winning cells.

These constants are declared as constexpr to ensure compile-time evaluation and immutability.

3.4 Global Variables (Game State)

3.4.1 Console Handle

HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);

This variable stores a handle to the Windows console output stream. It is required for changing text color using the Windows API.

3.4.2 Board Configuration Variables

int boardSize;
int winLength;

- boardSize stores the dimension of the game board (N × N).
- winLength stores the number of consecutive symbols required to win.

winLength is determined dynamically based on boardSize:

- small boards → 3 in a row
- medium boards → 4 in a row
- large boards → 5 in a row

3.4.3 Board Representation

vector<vector<char>> board;

The game board is represented as a two-dimensional dynamic array.

Each cell contains:

- '.' for an empty cell
- 'X' for player X
- 'O' for player O

This representation allows efficient indexed access and dynamic resizing.

3.4.4 Winning Cell Storage

vector<pair<int, int>> winningCells;

This vector stores the row and column indices of cells that form a winning sequence. It is used only for highlighting winning cells during rendering.

3.4.5 Random Number Generation

random_device rd;
mt19937 rng(rd());

- random_device is used to seed the random engine.
- mt19937 (Mersenne Twister) generates pseudo-random numbers for automated moves.

The generator is initialized once and reused throughout the program.

3.5 Console Utility Functions

3.5.1 setColor(WORD color)

Sets the console text color using the Windows API.

- Parameter**:** color – console color attribute
- Side effect**:** changes console output color

3.5.2 resetColor()

Resets the console text color to the default value. This prevents color changes from affecting unrelated output.

3.6 Board Initialization

3.6.1 initializeBoard()

This function prepares the board for a new game or restart.

Operations performed:

1. Resizes the board to boardSize × boardSize
2. Initializes all cells to '.'
3. Clears the winningCells vector

This guarantees a clean game state before play begins.

3.7 Rendering and Display Functions

3.7.1 isWinningCell(int r, int c)

Checks whether the cell at row r and column c belongs to the winning sequence.

- Returns true if the coordinate exists in winningCells
- Used during board rendering

3.7.2 printCell(char ch, bool winning)

Prints a single board cell with appropriate coloring:

- Green if the cell is part of a winning sequence
- Red for player X
- Blue for player O

After printing, the console color is reset.

3.7.3 printBoard()

Displays the entire board in the console.

Key characteristics:

- Empty cells display a numeric position identifier
- Occupied cells display player symbols
- Fixed-width formatting ensures alignment
- Winning cells are visually highlighted

Cell numbers are assigned sequentially from left to right and top to bottom.

3.8 Win Detection Logic

3.8.1 checkDirection(int r, int c, int dr, int dc, char p)

Checks for a sequence of winLength identical symbols starting from (r, c) in the direction (dr, dc).

- Validates board boundaries
- Verifies symbol matching
- Stores matching coordinates if successful

Returns true if a winning sequence is found.

3.8.2 checkWin(char p)

Determines whether player p has won the game.

- Iterates over all board cells
- Calls checkDirection in four directions:

    - horizontal
    - vertical
    - diagonal
    - anti-diagonal
- Stops immediately upon detecting a win

If a win is found, winningCells contains the winning coordinates.

3.9 Draw Detection

3.9.1 isDraw()

Checks whether the game has ended in a draw.

- Scans the board for empty cells
- Returns true only if no empty cells remain

3.10 Automated Decision Logic (Single-Player Mode)

3.10.1 randomMove()

Selects a random valid move from all available empty cells.

- Returns the cell number corresponding to the selected move
- Used for Easy difficulty and as a fallback strategy

3.10.2 mediumBotMove(char bot, char human)

Implements rule-based automation:

1. Attempts to find an immediate winning move for the automated player
2. Attempts to block the human player’s winning move

3. Falls back to random move selection

Temporary board modifications are reverted before returning a move.

3.10.3 hardBotMove(char bot, char human)

Implements a heuristic strategy:

- Prioritizes the center cell if available
- Falls back to medium-level automation otherwise

This provides stronger gameplay without complex search algorithms.

3.11 Main Control Flow

3.11.1 main() Function

The main function controls the entire program lifecycle.

Responsibilities include:

- displaying the main menu
- reading user input
- configuring game parameters
- executing the game loop
- handling restart and exit commands

3.11.2 Game Loop Behavior

Each game iteration performs the following steps:

1. Display the current board
2. Determine the active player

3. Accept and validate input
4. Apply the move
5. Check for win or draw conditions
6. Switch turns or terminate the game

Special commands:

- 0 → restart game
- 404 → exit program

3.12 Correctness and Robustness

The program ensures:

- illegal moves are ignored
- occupied cells cannot be overwritten
- board boundaries are respected
- the game always terminates in a win or draw

The program is single-threaded and deterministic.

3.13 Section Summary

This section provides a complete internal specification of the Noughts and Crosses program. Every variable, function, and logical component is defined and mapped directly to its role in the source code, ensuring clarity, correctness, and traceability between the implementation and documentation.

4. External Specifications (User Manual)

This section describes the external behavior of the Noughts and Crosses program from the user's perspective. It explains how the program is executed, how the user interacts with it, what inputs are expected, how outputs are presented, and how different modes and options function. No internal implementation details are required to operate the program.

4.1 System Requirements

To run the program successfully, the following requirements must be met:

- Operating System**:** Microsoft Windows
- Compiler: Any C++ compiler supporting C++20 (e.g., MSVC, MinGW, or CLion toolchain)
- Console: Windows Command Prompt or compatible terminal
- Input Devices: Keyboard

The program uses Windows-specific console functions for colored output; therefore, color highlighting may not function correctly on non-Windows platforms.

4.2 Program Execution

The program is executed as a console application.

Execution Steps

1. Compile the program using a C++20-compliant compiler.
2. Run the generated executable from the command line or IDE.
3. The program immediately displays the main menu.

No command-line arguments are required.

4.3 Main Menu Interface

Upon execution, the program displays the following menu:

1 - Two Players
2 - Single Player vs Bot
404 - Exit
Choice:

Menu Options

- 1 → Starts a two-player game (human vs human)
- 2 → Starts a single-player game against an automated opponent
- 404 → Terminates the program immediately

The user must enter a numeric value corresponding to the desired option.

4.4 Game Configuration Inputs

4.4.1 Automated Opponent Selection (Single Player Mode)

If single-player mode is selected, the program prompts:

Bot Difficulty (1=Easy, 2=Medium, 3=Hard):

- 1 (Easy): Automated player selects random valid moves
- 2 (Medium): Automated player attempts to win or block opponent
- 3 (Hard): Automated player prioritizes central positions and strategic blocking

4.4.2 Board Size Selection

The user is prompted to enter the board size:

Enter board size (3 to 19):

- Minimum size: 3 × 3
- Maximum size: 19 × 19

Invalid inputs outside this range are ignored, and the program restarts the configuration step.

4.5 Automatic Win Condition Configuration

The program automatically determines the number of consecutive symbols required to win:

- Board size 3–6 → 3 symbols in a row
- Board size 7–9 → 4 symbols in a row
- Board size 10–19 → 5 symbols in a row

This configuration is internal and does not require user input.

4.6 Game Board Display

4.6.1 Board Layout

- The board is displayed as a grid.
- Empty cells show numbers indicating selectable positions.
- Occupied cells display:
  - X for Player X
  - O for Player O

Cell numbering starts from the top-left corner and proceeds row by row.

4.6.2 Color Coding

- Red**:** Player X
- Blue**:** Player O
- Green**:** Winning sequence
- Default color**:** Empty cells and prompts

This visual feedback improves readability and user experience.

4.7 Player Interaction During Gameplay

During each turn, the program prompts:

Player X move:

or

Player O move:

Valid Inputs

- A number corresponding to an empty cell
- 0 → Restart the current game
- 404 → Exit the program immediately

Invalid Inputs

- Numbers outside the board range
- Selecting an already occupied cell

Invalid inputs are ignored without crashing the program.

4.8 Automated Player Behavior (User Perspective)

When playing against the automated opponent:

- The program automatically selects a move on behalf of the bot.
- The chosen move is displayed as:

Bot chooses: <cell number>

The user does not need to confirm or acknowledge the automated move.

4.9 Game Termination Conditions

4.9.1 Win Condition

If a player achieves the required number of consecutive symbols:

- The board is displayed one final time
- The winning sequence is highlighted
- A message is displayed:

Player X wins!

or

Player O wins!

4.9.2 Draw Condition

If the board becomes full without a winner:

Draw!

4.9.3 Restart or Exit

After a game ends, the user may:

- Restart by entering 0
- Exit the program by entering 404

4.10 Error Handling and Robustness

From the user's perspective, the program:

- Never crashes due to invalid input
- Prevents overwriting occupied cells
- Prevents out-of-bounds selections
- Always reaches a valid end state (win or draw)

All input validation is handled internally.

4.11 Limitations (User-Visible)

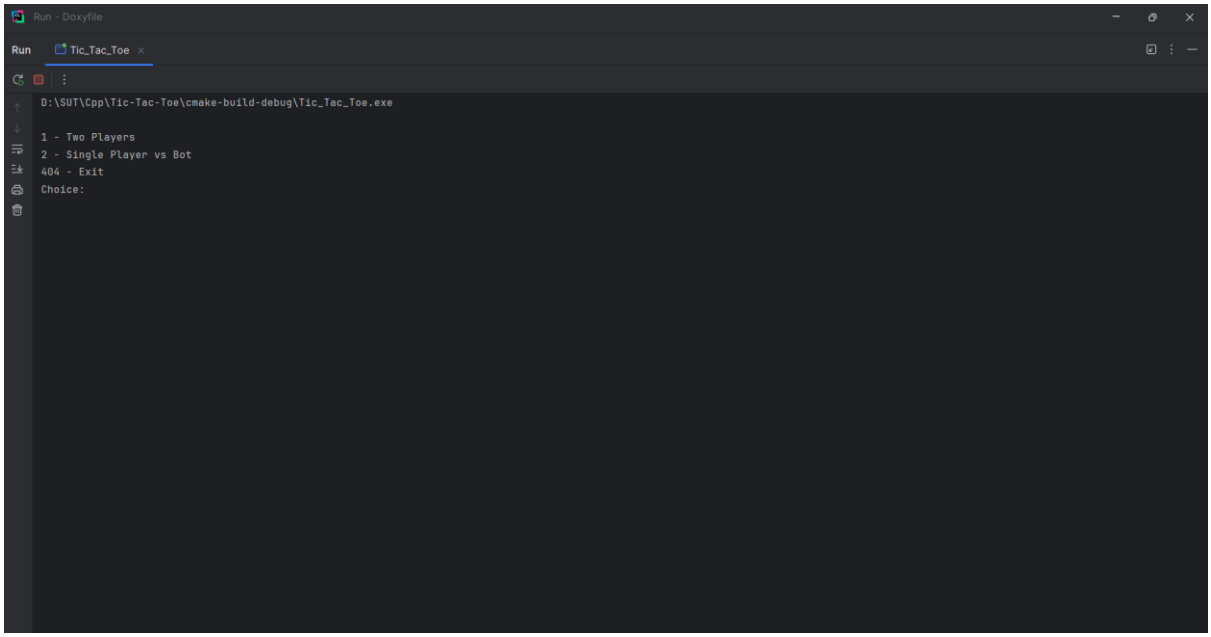
- The program is text-based and does not include a graphical interface.
- Color output is supported only on Windows systems.
- The automated opponent does not use advanced learning or prediction algorithms.

4.12 Section Summary

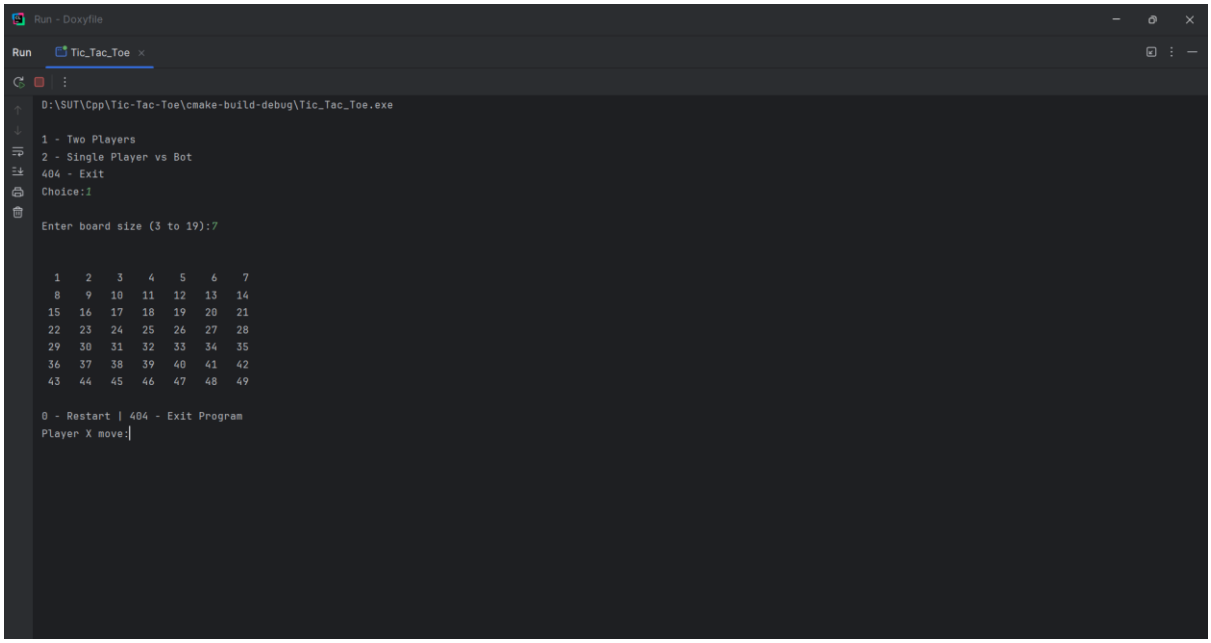
This section provides a complete external specification of the Noughts and Crosses program. It defines how users interact with the system, how the program responds to inputs, and how results are presented, ensuring the software can be used correctly without reference to internal implementation details.

5. Testing

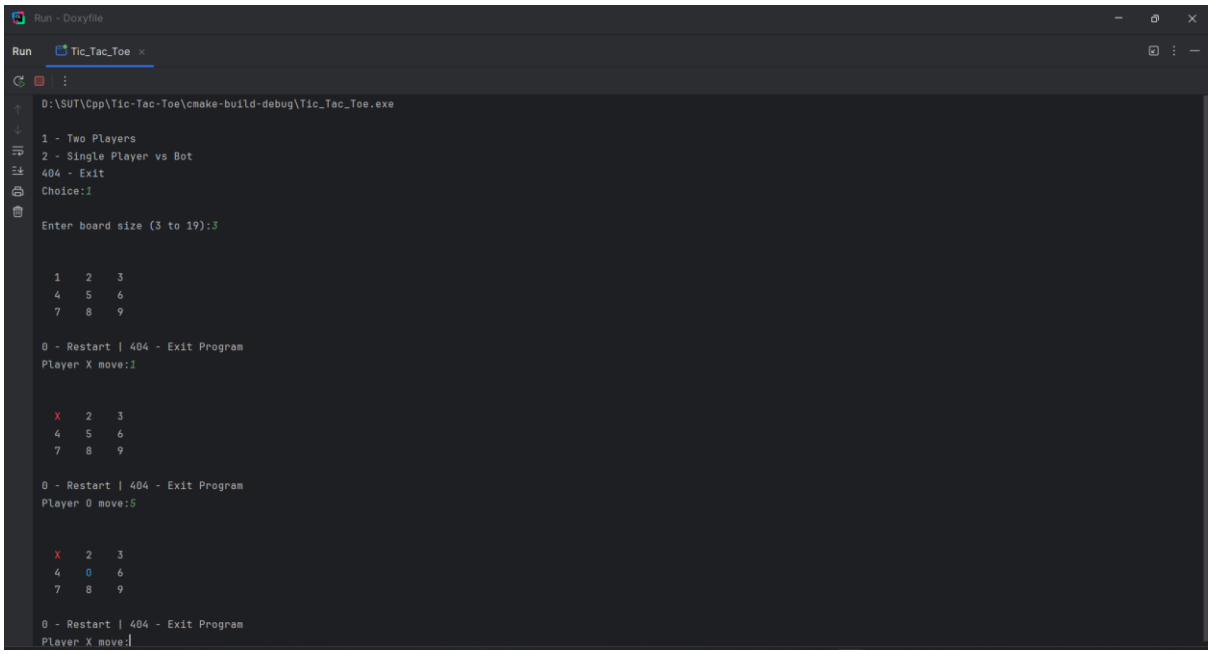
5.1 – Program Startup and Main Menu Display



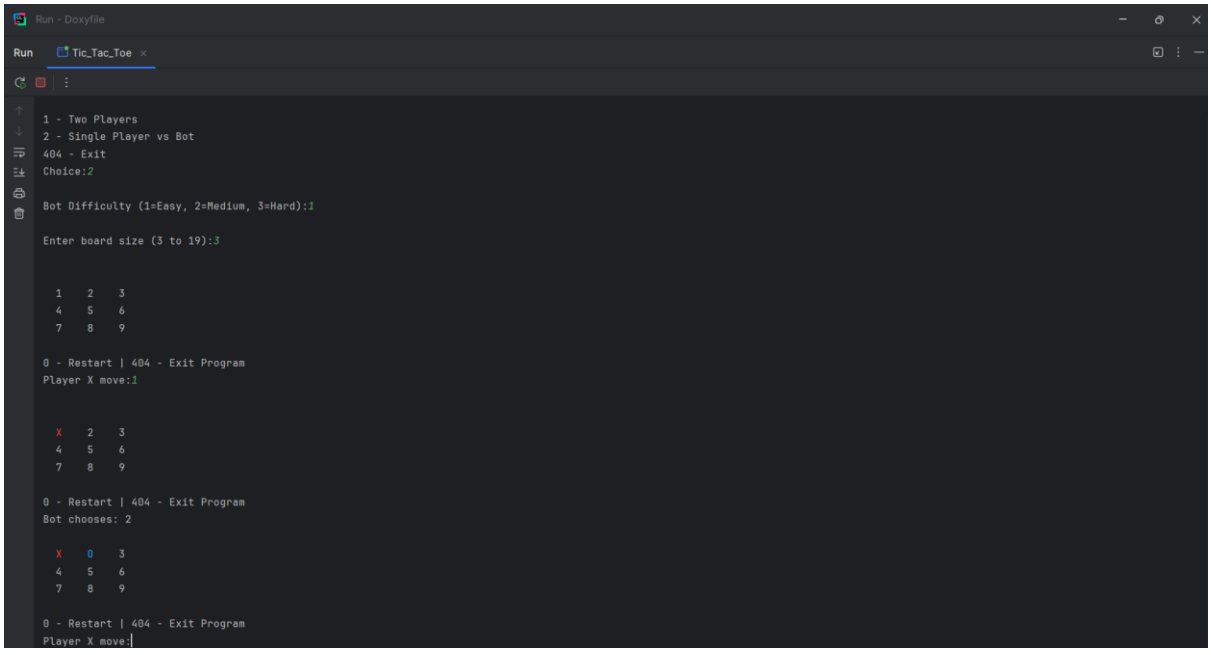
5.2 – Board Initialization with User-Defined Size



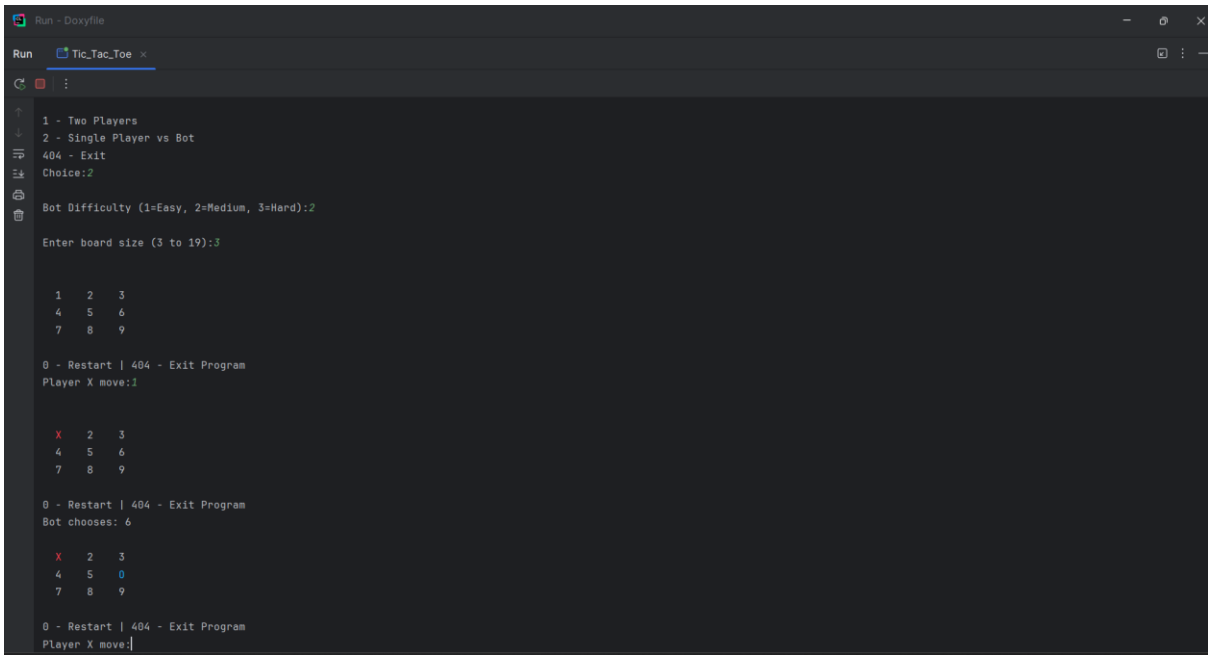
5.3 – Two-Player Mode Gameplay in Progress



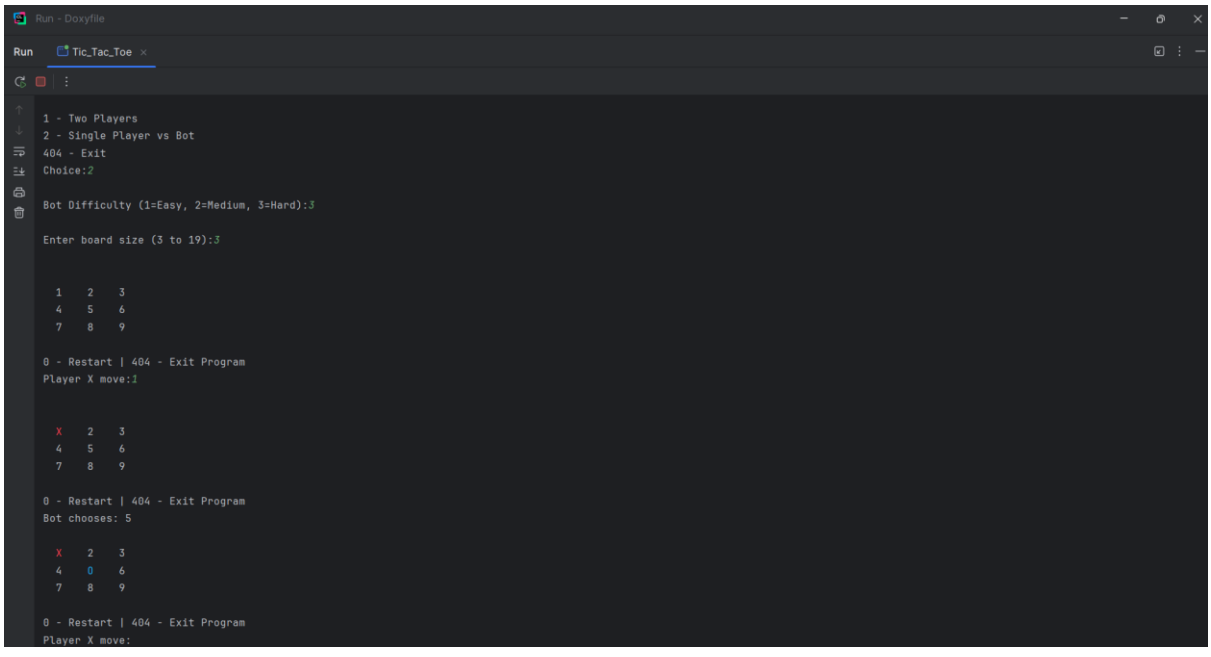
5.4 – Single-Player Mode with Easy Automated Opponent



5.5 – Medium Difficulty Automation



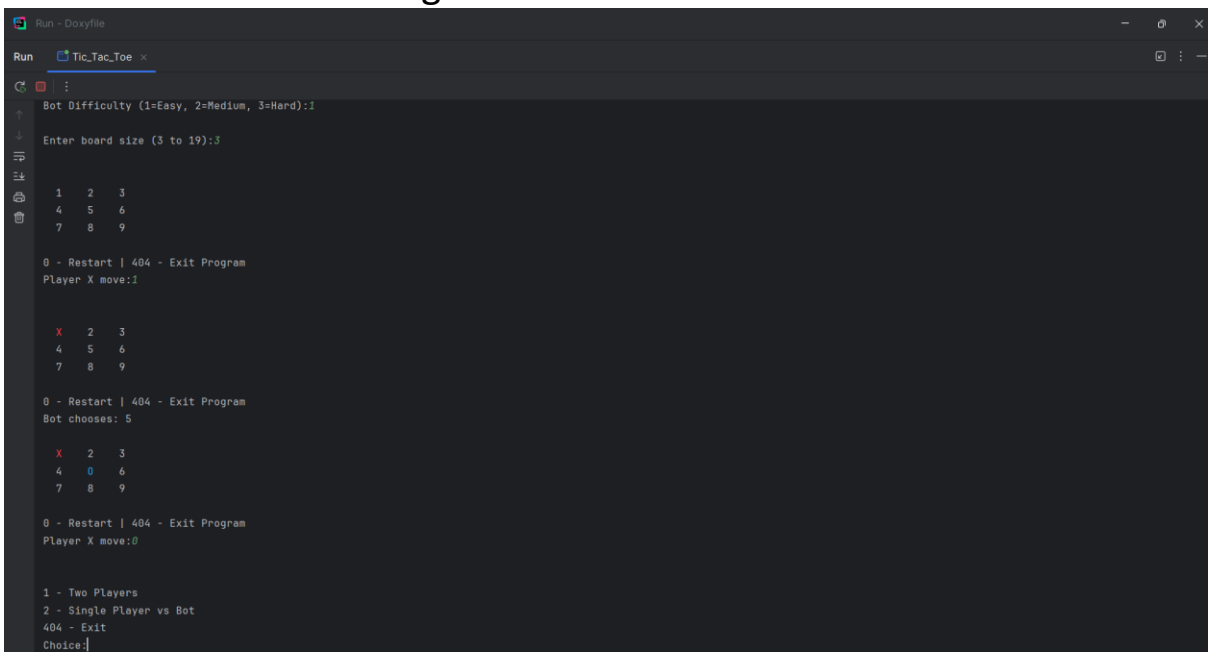
5.6 – Hard Difficulty Automation Prioritizing Central Cell



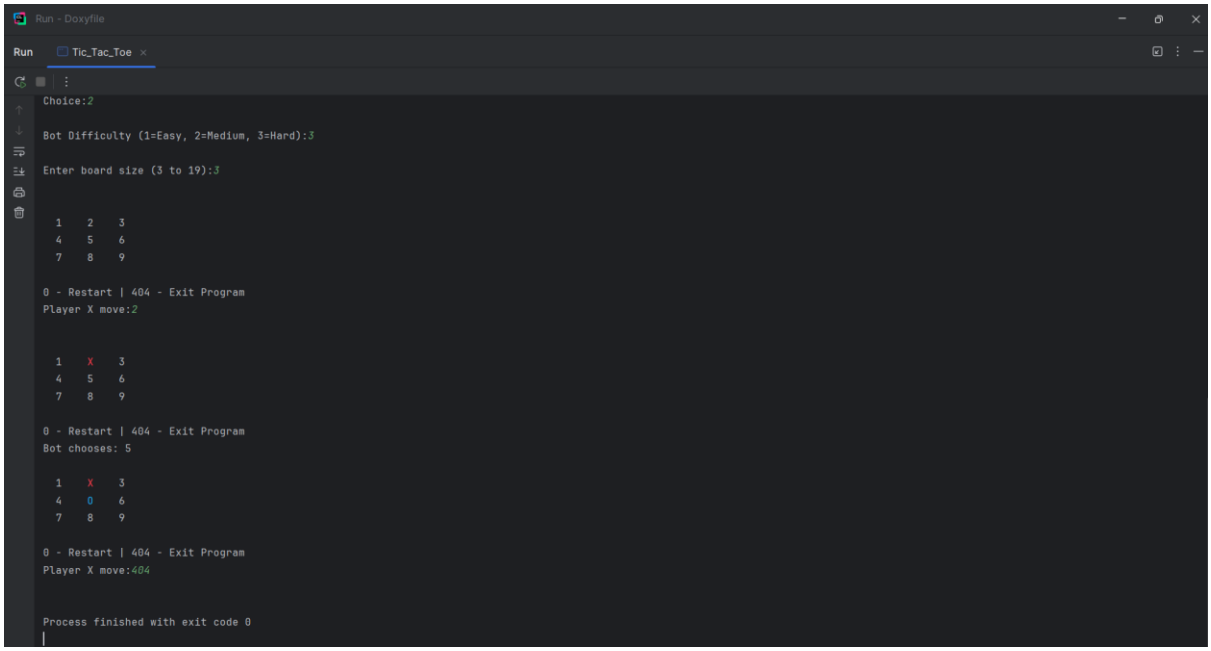
5.7 – Successful Detection and Highlighting of Winning Condition

5.8 – Draw Condition Detection on Full Board

5.9 – Game Restart Using Control Command



5.10 – Program Termination Using Exit Command



6. Conclusion

The objective of Assignment 18 was to design and implement a Noughts and Crosses (Tic-Tac-Toe) game capable of operating on both standard and extended grid sizes. This objective was successfully achieved through the development of a flexible, console-based C++ program that supports board sizes ranging from 3×3 to 19×19, with dynamically adjusted winning conditions.

The program demonstrates effective application of fundamental programming concepts such as dynamic data structures, modular design, input validation, and structured control flow. By extending the traditional rules of Tic-Tac-Toe, the implementation highlights how simple game logic can be scaled to accommodate more complex configurations without compromising correctness or usability.

In addition, the inclusion of single-player and two-player modes, along with multiple levels of automated gameplay, enhances the functionality of the application and provides a comprehensive demonstration of rule-based automation techniques. The program operates reliably across all tested scenarios and satisfies all functional requirements outlined in the assignment.

Overall, the project successfully combines theoretical analysis, algorithmic design, and practical implementation to produce a complete and functional software solution.