TIC-TAC-TOE PROJECT

JOHN P. BAUGH, PH.D.
THE COMPLETE C++ DEVELOPER COURSE

v. 2025.4.21.1

WHAT IS TIC-TAC-TOE?

Tic-tac-toe, also called **noughts-and-crosses** is a very popular and simple game. It involves two players, one player using symbol X (an "x" or "cross") and the other player using O (an "o" or "nought"). The players take turns placing their symbol on 3 x 3 game board, like the following:

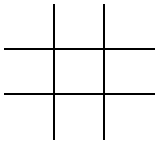

Figure 1 - an empty tic-tac-toe board

When a player gets three in a row (I'm using the term loosely – column or diagonal also work), then that player wins. The only other possibility is the board fills up with no one winning. This results in a **tie**, which in tic-tac-toe is called **the cat's game**. There are many stories about why this term is used, but one of the most popular is that it's like how a cat plays with its own tail. It chases it around and has fun, but no one really wins – hence, it's a cat's game (no one won, get it?!)

The purpose of the game is simple, yet two-fold. You want to get three in a row, but you also want to prevent your opponent from getting three in a row. So, you can block your opponent to prevent them from taking a given strategy to win.

Here are some game board examples:

WINNING EXAMPLES (X WINS)

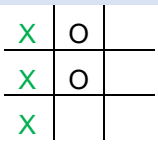

Figure 2 - X wins with 3 in a column

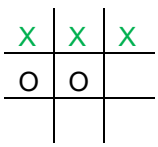

Figure 3 - X wins with 3 across

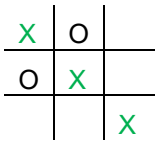

Figure 4 - X wins with 3 with diagonal

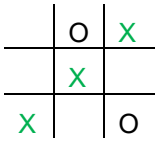

Figure 5 - X wins with 3 with reverse diagonal

CAT'S GAME (NO ONE WINS)

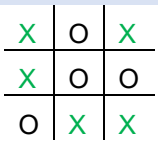

Figure 6 - Cat's game (no one wins)

Although you don't have to be a **domain expert** who knows the ins-and-outs of a topic or field of study, etc., it is important as a software developer to understand the fundamentals of how something works. For instance, you might get hired by a company that does biochemistry research. Even though you don't have to have a degree in biology, chemistry, biochemistry, or genetics to write software for them, it helps to understand some fundamentals, and it is important to understand the problem that they want solved.

Luckily for us, tic-tac-toe is fairly simple as a paper-and-pencil game. So, we can all ***pretty much*** become "domain experts" in it very quickly.

So now that you understand how the basic game works, let's take a look at design ideas for how we might go about structuring our program to handle this type of game.

MY DESIGN AND HINTS

In this section of the document, I'll describe the way I went about solving this project. This ***does not*** mean you have to do it exactly the way I did it, or at all. However, I do expect you to use good **modularization**, and divide some of the complexity of the project across multiple functions.

BASIC FUNCTIONS OVERVIEW

The functions I implemented (in addition to main, of course):

- **runGame** – the game loop
- **initializeGame** – sets the cells of the 2D array to spaces
- **printCurrentBoard** – prints whatever is in the current board
- **getUserInput** – gets the user input, and if valid, sets the game board appropriately
- **cellAlreadyOccupied** – returns true if a given cell (by row and column) is already occupied
- **getWinner** – returns "X", "O" if there is a clear winner, or " " (a space) if there is no winner yet
- **isBoardFull** – returns if the board is full or not

MY SCAFFOLDING / SKELETON PROGRAM

Note in the following code:

We use constants like `ROWS = 3` and `COLS = 3` to make the board size flexible and more readable. You could easily scale the game to a 4×4 or 5×5 grid just by changing these values.

```
#include <iostream>
#include <string>
using namespace std;

const int ROWS = 3;
const int COLS = 3;

void runGame();
void initializeGameBoard(string gameBoard[ROWS][COLS]);
void printCurrentBoard(string gameBoard[ROWS][COLS]);
void getUserInput(bool xTurn, string gameBoard[ROWS][COLS]);
bool cellAlreadyOccupied(int row, int col, string gameBoard[ROWS][COLS]);
string getWinner(string gameBoard[ROWS][COLS]);
bool isBoardFull(string gameBoard[ROWS][COLS]);

int main()
{
```

```
	runGame();


	return 0;
}//end main
```

Notice that for 2D arrays, you must put ***at least*** the second dimension when you pass them as parameters to functions. In the case above, I put ***both dimensions*** because the first is optional, and it just looks cleaner to me.

MORE DETAILS FOR THE FUNCTIONS

- runGame
  - Initializes the winner as a string that is empty, since no one has one yet
    - That string can be X, O, or C, which is the cat's game, that is, a tie
  - I also create the gameboard array of dimensions ROWS x COLS in this function
  - I call **initializeGameBoard**, which will fill the game board 2D array with spaces
  - After that, I call **printCurrentBoard**, which prints the board, including the lines and the contents of the 2D array cells
  - Then, I go into a game loop that continues as long as the winner has not been found yet
    - The game loop must keep track of whose turn it is
    - The game loop also calls **getUserInput** to obtain the user's input and modify the game board if they select a valid move, and tell them to pick another cell if they select an invalid move
    - After **getUserInput** is called, the game board is re-printed with **printCurrentBoard** since the board has been updated
    - After the board is printed, we set the **winner** variable to whatever is returned by **getWinner** (which could be an X, an O, or an empty string if there is still no winner
    - We flip whose turn it is to prepare for the next round
    - At the end of the game loop, we check if the board is full and a winner hasn't been selected yet, in which case we set winner equal to **C**, which means it's the cat's game
  - Still inside my **runGame** function, but after the loop, we tell the user if the cat has won (in other words, a tie), or if there is a winner X or O
- initializeGame
  - Loops through the game board and ensures all elements are set to a space
  - Note that I used a space, NOT an empty string
- printCurrentBoard
  - Takes the current board, prints the lines and the game board as necessary
- getUserInput
  - takes a parameter representing whose turn it is, and the game board
  - Goes into a loop to determine if the input has been valid yet or not
  - Valid selections include values for rows and cols >= 0, and <= 2
  - Then, I use the function **cellAlreadyOccupied** to determine if we keep asking or consider the selected row and column to be legitimate or not

    - After the validation loop, I then put an X or an O inside the gameboard at the row or the column
- cellAlreadyOccupied
    - takes the row and column being tested, and the game board as parameters
    - returns whether the gameboard has a space at that row and column, which would mean the cell is available
- getWinner
    - Takes the game board as a parameter
    - Checks winning conditions for the rows
    - Checks winning conditions for the columns if there was no row winner
    - Checks winning condition for diagonal top-left to bottom-right if no column winner
    - Checks winning condition for diagonal bottom-left to top-right
    - Default return value is empty string – remember that in the game loop, our winner variable is capturing the return value of getWinner, and if it is empty string “”, then the loop will continue
- isBoardFull
    - Takes the game board as a parameter
    - Loops through the game board and determines if all the cells are occupied
    - Hint: there are 9 total cells. If I count nine non-spaces, then that means the board is full
    - This function is crucial for determining the “cat’s game”, that is, that there is a tie – no winner

I strongly recommend not trying to write all the functions at once. For example, maybe try some of the “easier” functions that don’t depend as much on the others, and then build your program up. Some relatively simple functions to work with at first, or at least get started:

- initializeGame
- printCurrentBoard
- isBoardFull
- cellAlreadyOccupied

Once you’ve got these, start using them, and creating more as you can. It’s like a puzzle in some ways, and you can figure out how the different pieces go together. You can test your program **incrementally**, meaning with little pieces at a time. The main can call runGame, runGame can call some of the other functions like initializeGame and printCurrentBoard, and then you can keep going from there.

Hopefully, this document helps you solve the problem! Again – you *don’t* have to do it exactly the way I did. I just created this document to help you understand the game, and ***how I*** went about approaching the problem. There is no one right way to do it!

Good luck!