Programming Project: Tic-tac-toe
CPSC 298-6 Programming in C++
jbonang@chapman.edu

Introduction

A long time ago, when I was little, I was fortunate enough to take a field trip to the California Museum of Science and Industry in Exposition Park near downtown Los Angeles. At the time, the museum had a section called the Mathematica Room. One exhibit in the Mathematica Room astounded me; it was intriguing, fascinating. At the time, I didn't see how it was possible. The Mathematica room hosted a Tic-tac-toe "computer." The computer was imbued with intelligence, enough that the best I could do was a draw. Sometimes it would beat me. I couldn't believe it. How could a machine do that? How did it always know the right move to make?

Figure 1: The Tic-tac-toe computer in the Mathematica room at the California Museum of Science and Industry. (Only known photograph of the Tic Tac Toe computer, taken from *Westways* magazine, circa 1971.) (http://www.whmsicl.digitalspacemail8.net/CMSI3.html)

At the time, computers were inaccessible to me. I had never seen one except on television. Of course, things have changed and computers are ubiquitous today. A Tic-tac-toe computer is unlikely to trigger such fascination anymore, unless, that is, you build the computer yourself. Or, equivalently, write the software.

In this assignment, you'll create a Tic-tac-toe game that pits a human against the computer. The human player will make a move followed by the computer player's counter move. You'll write the software to implement the game and determine the computer player's move.

Your software implementation will adhere to the rules of Tic-tac-toe. (https://en.wikipedia.org/wiki/Tic-tac-toe) The human player will be X and the computer player will be O. X goes first.

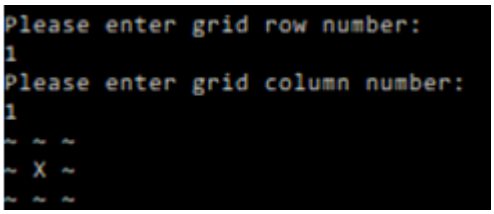
Your Tic-tac-toe game will be a command line game, played in a terminal window. The 3-by-3 Tic-tac-toe grid will initially be displayed as a series of tilde characters separated by spaces as shown below.

When the user makes a move, an X will be rendered in the appropriate grid location.

The computer player will then make a counter move, which will be rendered as an O on the grid.

Your game will consists of turns; within each turn you'll have the human player make a move followed by the computer player's counter-move, until the game ends. Your software will report that either the human or computer player has one, or that the game ended in a draw.

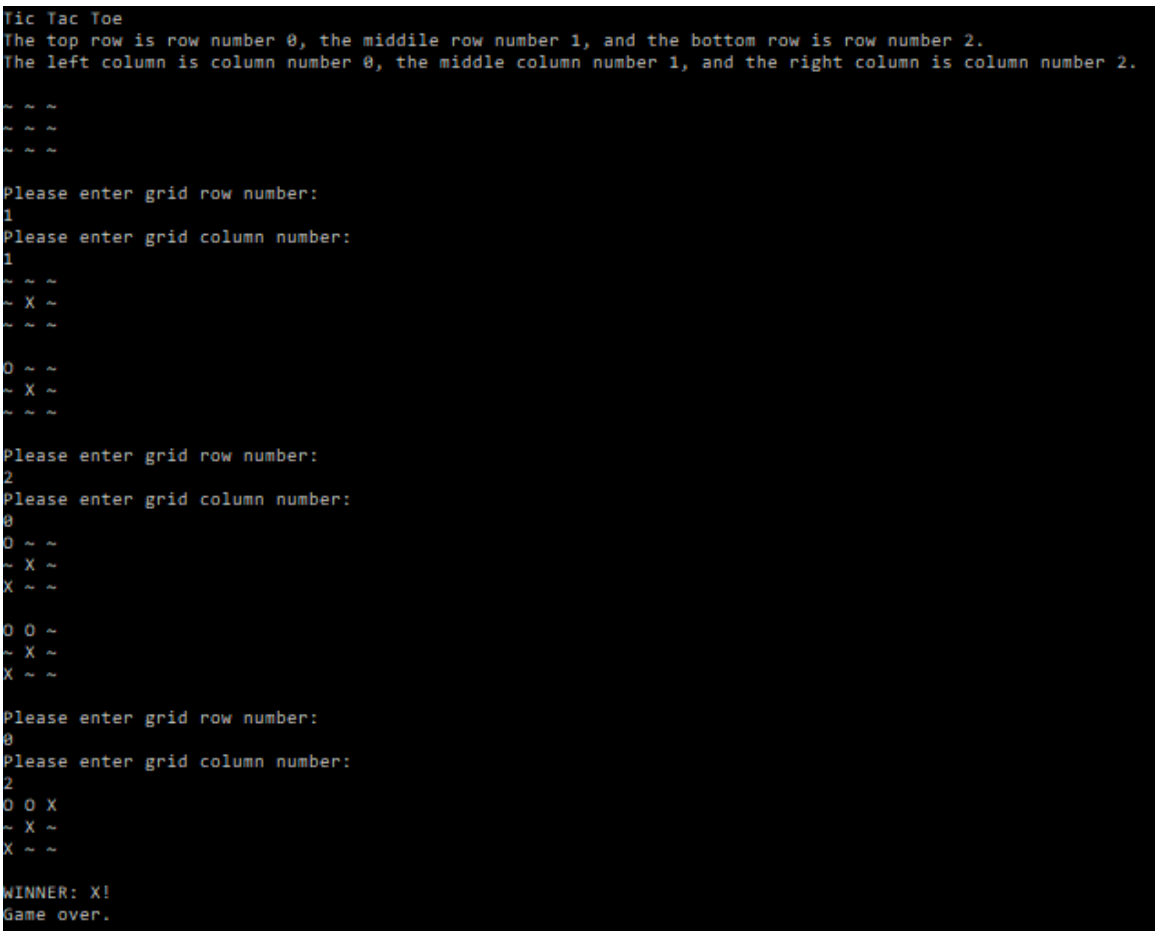
Your software will prompt the human player to enter the grid coordinates of their move at the start of the turn. After they enter the row and column number, the updated Tic-tac-toe grid will be displayed.



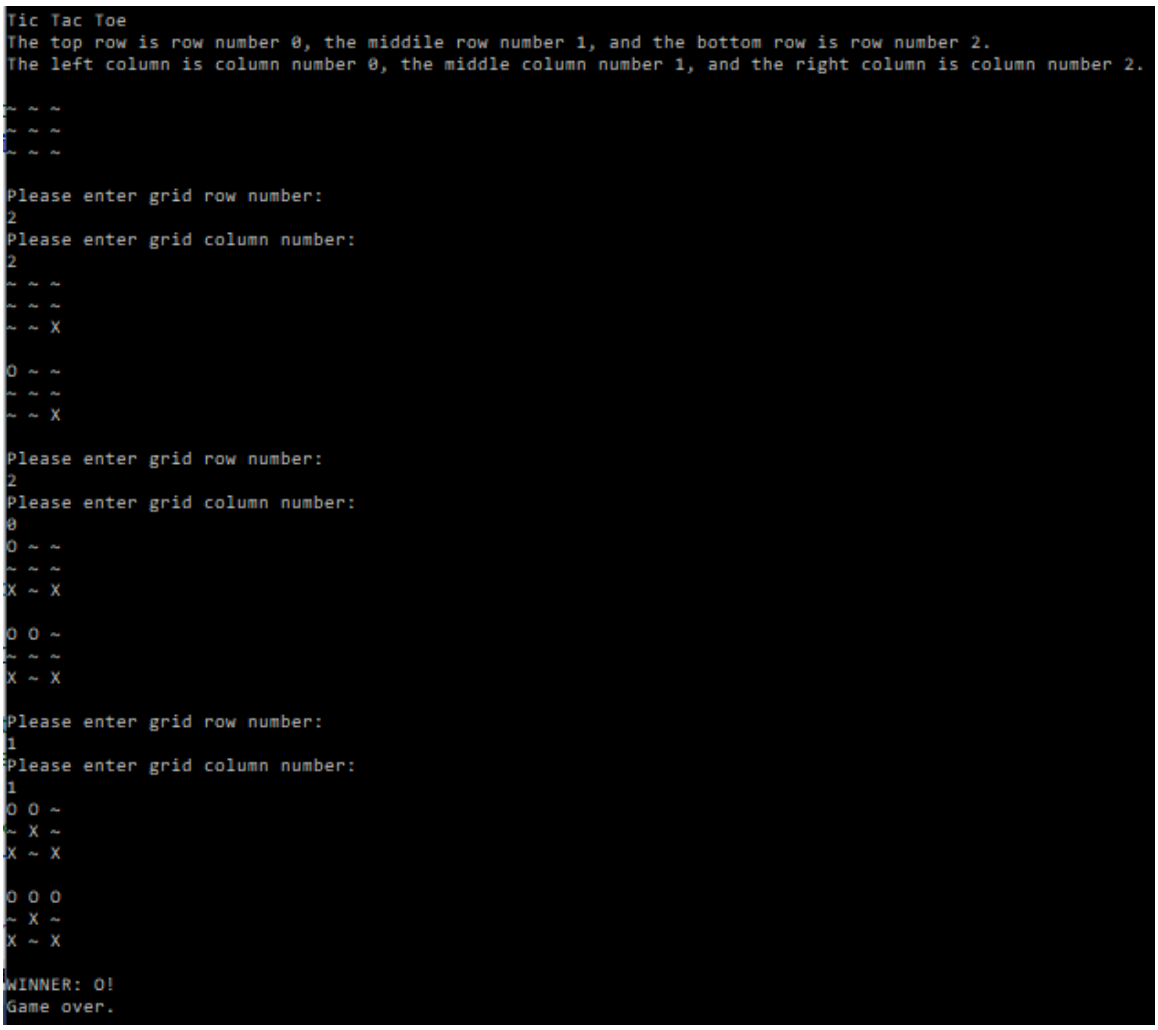
Immediately following this, the display will be rendered again, showing the computer's counter move.

Win, Lose and Draw

The figures below illustrate complete games. The first figure shows the complete game history of a game ending in a victory for the human player, X.



The next figure shows the complete game history of a game ending in a victory for the computer player, O.



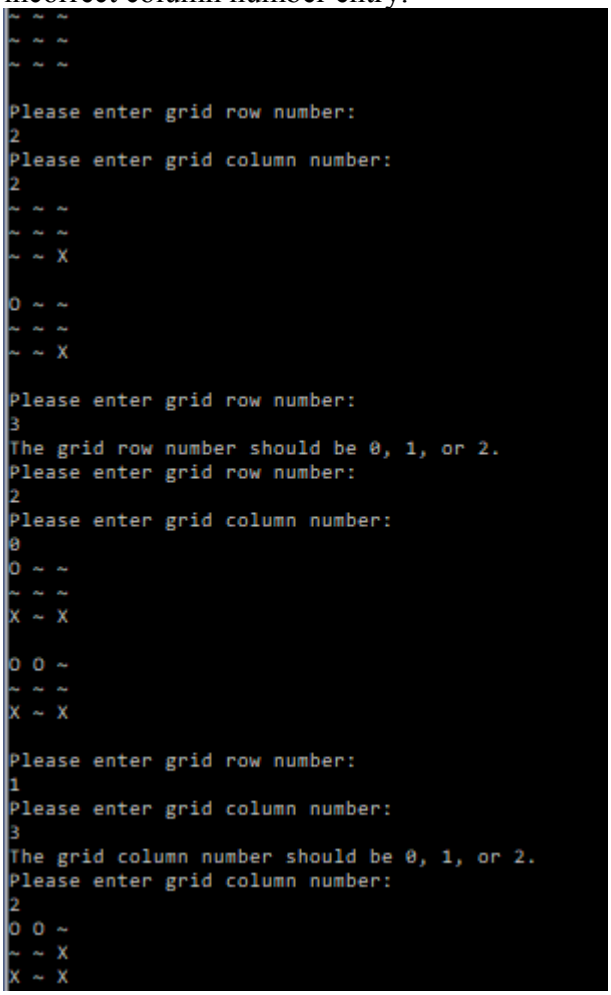
Lastly, the figure below shows the game history of a game ending in a draw, with neither player winning.

```
Tic Tac Toe
The top row is row number 0, the middile row number 1, and the bottom row is row number 2.
The left column is column number 0, the middle column number 1, and the right column is column number 2.

~ ~ ~
~ ~ ~
~ ~ ~

Please enter grid row number:
0
Please enter grid column number:
1
~ X ~
~ ~ ~
~ ~ ~

O X ~
~ ~ ~
~ ~ ~

Please enter grid row number:
1
Please enter grid column number:
0
O X ~
X ~ ~
~ ~ ~

O X O
X ~ ~
~ ~ ~

Please enter grid row number:
1
Please enter grid column number:
2
O X O
X ~ X
~ ~ ~

O X O
X O X
~ ~ ~

Please enter grid row number:
2
Please enter grid column number:
0
O X O
X O X
X ~ ~

O X O
X O X
X O ~

Please enter grid row number:
2
Please enter grid column number:
2
O X O
X O X
X O X

Draw!
Game over.
```

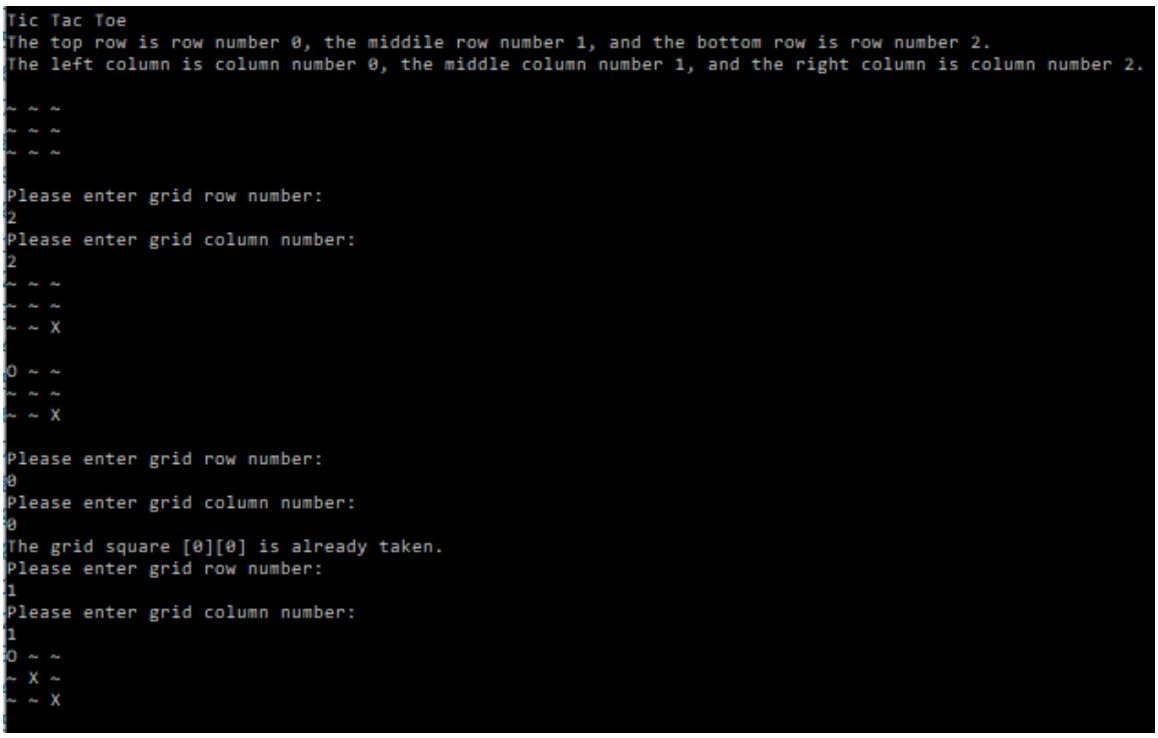
Checking User Input

If the user types in an invalid row number, such as 3, output a message indicating the row number should be 0, 1 or 2. Similarly, if the user enters an invalid value for the column, output a message indicating the column number should be 0, 1 or 2.

An excerpt from a game history shows both an incorrect row number entry and an incorrect column number entry.



If the user enters a grid square that is already occupied by an X or an O, inform them that the grid square is already taken and ask them to choose another.



The Game's Artificial Intelligence

You'll implement the computer player's move selection. It can be as simple as choosing the next open grid square. This is straightforward and easy to implement, and is all that is required for this assignment. It makes for terrible game play though. If you implement something more sophisticated, you'll earn extra credit points. For instance, even a random choice among the grid squares that remain open would do better. The better the AI, the more extra credit points you receive.

The Assignment

Create a C++ class that implements the Tic-tac-toe game described in the introduction.

The class must implement a public `play` function which contains the game loop. Each iteration of the loop should implement a turn consisting of the human player's move followed by the computer player's counter-move. After each move, a check is made to determine if a player has won or a draw is reached.

Dynamically Allocate the C++ Class in main

The C++ class must be dynamically allocated in the main program using the new operator - something similar to the following:

```
TicTacToeGame* p_game = new TicTacToeGame();
```

You must also delete the memory associated with the object at the end of the main program.

Dynamically Allocate the Game Grid

The game grid is effectively a 3-by-3 two-dimensional array. However, it is to be implemented as an array of pointers to character.

The C++ class should contain a private member that is a pointer to a pointer to a character type, as shown below.

```
char** pp_cGrid; ///< Pointer (p) to (pointer to character (p_c))
```

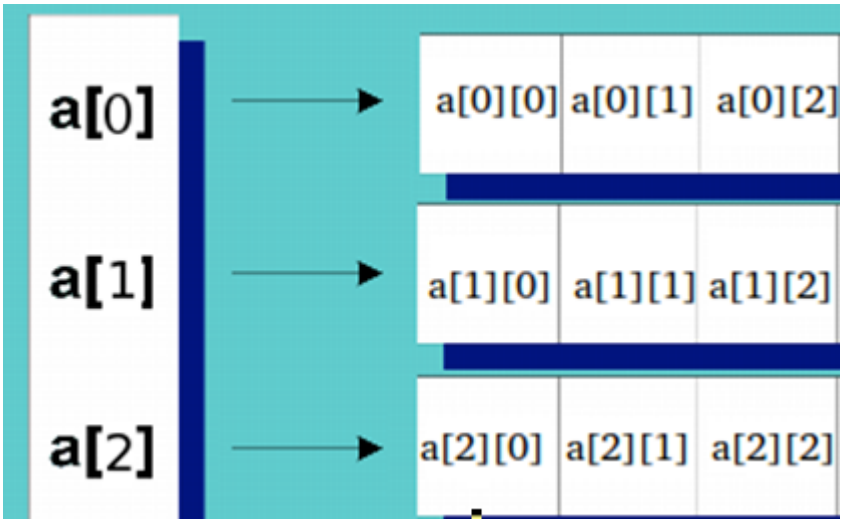
The entire "array" will be dynamically allocated. First, dynamically allocate the C++ array of pointers to "pointers to char" using the new operator so that it contain exactly three elements.

```
pp_cGrid = new char* [3];    // array of three pointers to char
```

Next, you'll allocate the three char arrays and set the pointers within pp_cGrid to point to them.

Suppose you name your array "a" (instead of pp_cGrid). You'll first allocate the array of three pointers shown in the left hand column. (This would be equivalent to a = new char* [3];

```
char **a = new char* [3];
```



Then you'll dynamically allocate each of the character arrays, and set the pointer within one of the elements of a to point to the newly allocated memory.

```
a[0] = new char[3];
```

You'll do the same for the a[1] and a[2] elements.

These pointers may be accessed using array notation as shown in the figure above. a[1][0] gets the second (index 1) row and the first (index 0) column. Note that this acts as a two-dimensional array as the figure suggests.

Functionality

You'll need member functions to obtain the user move, to determine the computer's counter move and to check for a win or a draw. You'll also need member functions to display the grid and to initialize it. (It should be initialized to contain all '~' characters.)

You'll to check for the user-input errors described in the introduction. If the user enters an invalid row or column number, 3 for example instead of 0, 1 or 2, your program must report the error and prompt the user to try again.

If the user enters a grid square that is already occupied by an X or an O, the program must report the error and let the user enter a new grid square. Be careful, this requires nested for loops and the logic isn't completely straightforward.

You do **not** need to handle invalid input such as the user entering the 'c' character instead of an integer. (Exceptions can easily handle that, as you'll see in a later assignment.)

You can implement a simple "get next available grid square" algorithm to determine where the computer should place its 'O' as its counter-move. (Iterate over the grid to determine a square that contains a '~' character and not an 'X' or an 'O'.) If you implement something more sophisticated, you'll be eligible for extra credit.

The main function should create an instance of the class dynamically, call the play() function to run the game loop, deallocate the class when the game is over and return 0

Deallocate the memory for the grid in the class constructor.

Completing the Assignment

Show a screen capture of one full game where the computer player wins.

Show a screen capture of one full game where it ends in a draw.

If possible, show a screen capture of one full game where the human player wins. (If you AI is really good, this may not be possible.)

Submit three source files: TicTacToe.h, TicTacToe.cpp, main.cpp.

Requirements

Grading will reflect how well the program meets the following user interface requirements and implementation requirements. The requirements are specified using traditional shall statements, still a common practice in software engineering though now supplemented by user stories.

User Interface Requirements

Upon entry of an incorrect row number by the user, the game *shall* report that the row number is incorrect and prompt the user to reenter it.

Upon entry of an incorrect column number by the user, the game *shall* report that the column number is incorrect and prompt the user to reenter it.

Upon entry of a row and column by the user corresponding to an occupied grid square (one already containing an X or an O), the game *shall* report the error and prompt the user to enter a new row and column.

Implementation Requirements

The game implementation *shall* be contained within a C++ class.

The game implementation *shall* represent the Tic Tac Toe grid using a C++ array of pointers to character. (The C++ array of pointers to characters acts as a two-dimensional array.)

The game implementation *shall* maintain a C++ pointer to pointer to character member variable to point to the array of pointers to character.

The game implementation *shall* dynamically allocate the C++ array of pointers to characters using the new operator to contain exactly three elements.

The game implementation *shall* allocate the char arrays of size 3 pointed to by the pointers in the C++ array of pointers to characters using the new operator to contain exactly three elements.

The game implementation *shall* deallocate the three character arrays of size 3 pointed to by the pointers in the C++ array of pointers to characters using the delete[] operator in the class destructor.

The game implementation *shall* deallocate the array of pointers to pointer to character using the delete[] operator in the class destructor.
The C++ class containing the game implementation shall be dynamically allocated in the main function.

The memory allocated for the C++ class containing the game implementation *shall* be dellocated in the main function using the delete operator when the game is over and just before the main function returns.