COMSC-165 Lab 10
Tic-Tac-Toe Game

Write a 1- or 2-player **tic-tac-toe** game, with X's and O's in the standard 3x3 grid. The lab is designed to show how to break down a large programming project into small parts, and by building these parts, construct the full program. it also introduces the idea of "persistence", using file I/O to track a user's wins, losses, and ties.

Create a C++ console application for the game **tic-tac-toe**, with the features introduced in each of the following steps.

As you complete this assignment, post the required file(s) to the COMSC server. For assignments that involve more than one file, you may post them all at once, or one-at-a-time as you complete them, as you prefer. For assignments that involve more than one separate "lab" (like lab 1a, 1b, and 1c), the individual labs are graded in order, starting with "a", which must be fully correct before "b" is graded, and so on. This assignment is worth 50-points, to be awarded after all labs are successfully completed. Use the "Submit your FA2014 work" link on the class website to post your file for this lab, using the **lab10** folder provided.

Lab 10, Tic-Tac-Toe Game [`T3.cpp`]
Create a console application named **T3.cpp**. Do NOT submit until you complete the last step.

STEP 1 of 8: Print The Grid
In `main()`, use a 1D array of `char`s to store the X's and O's that appear in the tic-tac-toe grid squares. Initialize these all to a *blank space*. E.g.:

```
char grid[] = {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '}; // array of nine chars
```

or...

```
char grid[] = "         "; // nine spaces as a string -- includes null terminator as 10th character
```

Write a function with this prototype: `void printGrid(const char*);`, that prints the tic-tac-toe grid with `cout`. (*Hint:* use 5 `cout` statements, each with `endl`, to do this. Use the plus (**+**), dash (**-**), and pipe (**|**) characters to form the grid.) E.g.:

```
 | | 
-+-+-
 | | 
-+-+-
 | | 
```

appearance of the grid at start of game,
with spaces in the 9 grid spaces

Use `grid[0]` for the mark (**X**, **O**, or blank space) in the upper left square of the grid, `grid[1]` for the upper middle, **`grid[2]`** for the upper right, **`grid[3]`** for middle left, **`grid[4]`** for the center, **`grid[5]`** for the middle right, **`grid[6]`** for lower left, **`grid[7]`** for the lower middle, and **`grid[8]`** for the lower right.

To test, place a **`printGrid(grid)`** function call in **`main()`**, compile and run. Customize the appearance of the grid as you wish.

STEP 2 of 8: X's Turn
Write a function with this prototype: **`void goXplayer(char*);`** that prompts the human X-player to take his turn. Validate the selection in two ways: (1) make sure the selection is a cell in the grid, and (2) make sure the cell is not already taken. Include a **`while`** loop to prompt the user until a valid response is given. Be sure to prompt the user to try again in case his selection is not valid. The valid responses are:

q or Q, upper left grid space w or W, upper middle grid space e or E, upper right grid space a or A, middle left grid space s or S, center grid space d or D, middle right z or Z, lower left grid space x or X, lower middle grid space c or C, lower right grid space	*or...* **Q|W|E** **-+-+-** **A|S|D** **-+-+-** **Z|X|C**

Be sure to provide some way for the player to know what are the valid choices.

After collecting the X-player's choice, the function should change the selected grid square to an **`X`**, and return from the **`goXplayer`** function.

To test, place a **`goXplayer(grid)`** function call *after* the **`printGrid(grid)`** call in **`main()`**, and then place another **`printGrid(grid)`** call after **`goXplayer(grid)`**. Compile and run to simulate the X-player taking the first turn.

STEP 3 of 8: O's Turn
Write three functions with these prototypes: **`void goOplayer1(char*);`**, **`void goOplayer2(char*);`**, and **`void goOplayer3(char*);`**, to get the O-player's choice. These are to be three user options for the O-player:

goOplayer1: A 2nd *human* player. To program a *2nd human player*, basically copy **`goXplayer()`** from step 2 above, and change its prompt to refer to the O-player. Then have it set the selected grid cell to an **`O`** instead of an **`X`**.

goOplayer2: A *dumb* computerized player. To program a *dumb computer player*, use **cstdlib**'s **`rand()`** function to randomly select a number between 0 and 8 as the computer's choice:

```
int choice = rand() % 9; // returns a number between 0 and 8
```

Put the above in a **`while`** loop until the computer selects a cell that is not already taken. (Remember that the dumb computer player cannot see the grid, nor does it remember its own selections!)

Be sure to *seed* the random number generator at the *beginning* of **`main()`**, so that the computer does not play each game exactly the same way. You only have to execute this statement *once* in your program, no matter how many times you call **`rand()`**.

```
srand(time(0)); rand(); // requires <ctime> and <cstdlib>
```

goOplayer3: A *smart* computerized player. To program a *smart computer player*, include a lengthy `if - else if - else` structure in `goOplayer()` to check all possible combinations of X and O locations in the grid before making a selection. E.g.,:

```
// try to win (24 combinations to check)
if ((grid[0] == 'O') && (grid[1] == 'O') && (grid[2] == ' '))
{
  grid[2] = 'O'; // O wins! 3-in-a-row across the top
  return;
}
...

// try to block (24 combinations to check)
if ((grid[0] == 'X') && (grid[1] == 'X') && (grid[2] == ' '))
{
  grid[2] = 'O'; // O blocks X from having 3-in-a-row across the top
  return;
}
...
```

The definition of *smart* is this: If possible, win. Else, if possible, block the opponent from winning. Else, make any other valid move. For our purposes, the computer does not have to be unbeatable to be smart.

To test goOplayer1, place a `goOplayer1(grid)` function call *after* the `printGrid(grid)` call in `main()`, and then place another `printGrid(grid)` call after `goOplayer1(grid)`. Compile and run to simulate the first turn.

Repeat for goOplayer2 and goOplayer3, testing each one-at-a-time. Once you are sure that all 3 work correctly, proceed with the next step. Keep all three functions in your program, but use only one of them, per your choice. It will be a user option later in the program for deciding which function to use.

STEP 4 of 8: Check For End Of Game
The game ends when either the X-player or the O-player wins, *or* when all nine grid cells are selected without a winner -- i.e., a *tie*. Write three functions with prototypes: `char check4Xwinner(char*);`, `char check4Owinner(char*)`, and `char check4Tie(char*)`, that each return `'Y'` or `'N'` for *yes* or *no*.

The two winner functions should consist of a lengthy `if - else if - else` structure, checking for end-of-game scenarios. E.g., for `check4Owinner()`:

```
if ((grid[0] == 'O') && (grid[1] == 'O') && (grid[2] == 'O'))
  return 'Y'; // O wins! 3-in-a-row across the top
else if ((grid[3] == 'O') && (grid[4] == 'O') && (grid[5] == 'O'))
  return 'Y'; // O wins! 3-in-a-row across the middle
else if...

else
  return 'N'; // no winner yet
```

There are 8 possible winning scenarios per player.

The tie function should consist of a simple `if - else` structure -- e.g.:

```
if (grid[0] == ' ')
  return 'N'; // it's not a tie
else if (grid[1] == ' ')
  return 'N'; // it's not a tie
...
else // no blanks in the grid
  return 'Y'; // it's a tie
```

To test, place a `check4Xwinner(grid)` function call *after* the `goXplayer(grid)` call in `main()`, and a `check4Tie(grid)` after that. Then place a `check4Owinner(grid)` function call *after* the `goOplayer(grid)` call in `main(grid)`. There is no need to check for a tie after O's turn. Compile and run to simulate the first turn.

STEP 5 of 8: The Next Turn
Place a `while` loop around the statements in `main()`, to allow the game to continue until there is a winner or a tie. Replace the 4 new function calls from step 4 above with `if` statements that print the break out of the loop in `main()` in case there is a winner or a tie, and that print the gate result. E.g.:

```
if (check4Tie(grid) == 'Y')
{
  cout << "it's a tie";
  break;
}
```

Compile and run.

STEP 6 of 8: Add User Option
Place the code in `main()` to allow the user to select which `goOplayer` they want to use -- 2-player, dumb computer, or smart computer. Remember the choice in a variable, and use a switch statement to call the right function.

Compile and run.

STEP 7 of 8: Play It Again, with User Option
Place the code in `main()` inside a while loop, to allow multiple games to be played. Before the closing brace in the new loop, use `cout` and `cin` to ask the X-player if they want to play again.

The user option for `goOplayer` should be made ABOVE the replay loop. The user should make this selection only once per playing session, and it should apply to all replays.

STEP 8 of 8: Keeping Score
Add code in `main()` to keep track of the number of X wins, O wins, and ties. Store the results in a file named **t3.ini**. Use either text format or binary format, as you prefer. Above the first `while` loop in `main()`, declare 3 `int` values to store these statistics: `xwins`, `owins`, and `ties`. Then try to open **t3.ini** and read the three `int`s. If the attempt to open the file fails, which it will upon the first run of your program, set all three `int`

values to zero. Otherwise, read them from the opened file, and close the file. Print the numbers of wins, losses, and ties before the game session starts, and after each game.

Update the values of the three **`int`**s as the games are played (e.g., **`ties++;`**) Do *not* update the **t3.ini** file until all games are complete, and the gaming session is about to end. So before the closing curly-brace of **`main()`**, open **t3.ini** for output, and save the three **`int`**s to it. Compile and run.

*Post **T3.cpp** to the COMSC server.*

GRADING POLICY
Lab work will be returned for you to fix and resubmit for any of the following reasons, or for not following stated specifications, unless directed otherwise by your instructor, Point penalties will be assessed per the syllabus for each time a lab is returned for redo for any listed reason below or specification above. Points will be awarded for an assignment upon final, successful completion of all of its labs. Note that you will NOT receive a list of corrections to make, because to do so would remove any incentive for students to do quality work. Instead, grading of your work will stop upon discovery of the first mistake. Grading will not continue until the mistake is corrected. So to avoid multiple cycles of correction and resubmission, and the cumulative point penalties, make sure that your work is as correct as you can make it before submitting it.

Basic checklist, even if your compiler "forgives" doing otherwise:

1. Spell and case filenames *exactly* as specified. To avoid errors, be sure to configure your Windows computer to NOT hide file name extensions -- refer to the Burns Intro to Programming text, chapter 2.
2. Submit files to the specified folder. Do NOT submit archive files, like ZIP, JAR, or RAR.
3. Include identifying information as the first lines in the file AND as the first lines of your output in main.
4. Do NOT use the statement `system("PAUSE");` or any other system-specific code.
5. Do NOT try to access the value stored in an uninitialized variable.
6. CPP and H files must be free of careless typing mistakes that would prevent compilation.
7. Put an `endl` as the last item in the last-executed `cout` statement before the closing curly brace of `main`.
8. Do NOT forget to use the `cin.ignore` statements for non-string console input, unless specifically directed otherwise.
9. Do NOT forget to use the `fin.ignore` statements for non-string file input, unless specifically directed otherwise.
10. For file I/O, NEVER include a path designation. ALWAYS use the "working folder" for files.
11. Use the proper `#includes` and `usings` as listed in the C++ Library PDF.
12. Do NOT #include C-libraries such as `math.h` -- include their C++ equivalents instead, like `cmath`.
13. Do NOT #include libraries that you do not use.
14. Do NOT #include "**stdafx.h**"; do NOT use `pragma`.
15. Do NOT use C/C++ constructs or library items that are not specifically taught in this class, such as `goto`, `scanf` or `fscanf`, `fixed`, `strcmp_c`, `stricmp`, etc.
16. Do NOT use `void main()` -- use `int main()` always.
17. Do NOT use `return 0;` as the last statement in `int main()`, because it is not required by ANSI Standard C++.
18. If you use output statements in your program for debugging purposes, be sure to delete them before submitting your work.
19. Do *not* use "global variables".

Advanced checklist, even if your compiler "forgives" doing otherwise:

1. `NULL` is ONLY a pointer value -- do NOT use it for integer or other numeric values, even if your compiler allows it.

2. Do NOT declare a static array with a variable size specification, such as `int n =...; int a[n];`, even if your compiler allows it.
3. NEVER #include a CPP file, even if the textbook includes examples of such.
4. For console input and text file input with `>>`, use `.ignore` as explained in the Burns Intro to Programming text, chapter 5, unless specifically directed otherwise.
5. NEVER use `cin >>` to read directly into a numeric variable, like `int` or `double` or `float`. Read as a string (C or C++) and convert to a number using `atoi` or `atof`. Include `cstdlib`!
6. Do NOT use the `inline` keyword. To make class member functions inline, write their definitions inside the class definition.
7. Once the `const` keyword is introduced in this course, use it where applicable. Note that class "getter" functions are not really getter functions unless they have a trailing `const` keyword to flag them as such.
8. Make sure that all value-returning functions return a valid value under any logical circumstance.
9. Exit functions with `return` statements, and NOT `exit` -- do NOT use `exit` under any circumstances.
10. In every H and CPP file, ALWAYS #include libraries and H files for ALL functions, classes, and identifiers used in that file. Exception: use a class "forward declaration" instead of #including the class' H file when only pointers and reference variables appear.
11. Include the `cstring` library for char-array-based string ("C string") *functions*, and the `string` library for STL-based-strings ("C++ strings").
12. When using the random number generator seed function `srand`, call it ONCE ONLY in your program, and "sacrifice" the first value, like this: `srand(time(0)); rand();` at the top of main. Be sure to #include both `ctime` and `cstdlib`.
13. In COMSC 210, use `using namespace std;`. In 165 and 200, use `using std::` statements, properly placed under the #includes of the libraries to which they pertain.
14. `std::string` and `std::getline` belong to the `string` library. In 165 and 200, use `using std::` statements for these. Note that `cin.getline` is a different thing, and only requires `using std::cin;`.
15. Do NOT use `using std::` for anything in the C libraries. Use it for items from C++ libraries, only.
16. Test drivers MUST at least include ifndef testing, object copy testing with assignment after declaration, and const object testing with assignment upon declaration.
17. Test drivers must NOT include console or file input, unless directed otherwise.
18. Do NOT allow memory leaks, unless specifically instructed otherwise.