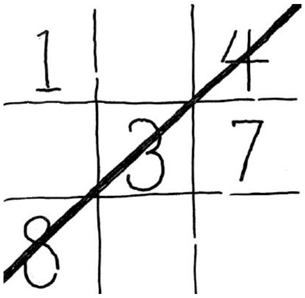
CISC 3110

Lab 2: Tic-Tac-Math and Multiple Files

Objectives:

- To gain experience writing and editing multiple-source-file programs.
- To practice designing modular programs.
- To practice using a build automation system (in this course, make).

Part I: Overview



Tic-Tac-Math is a lot like Tic-Tac-Toe. The game is played on a 3x3 grid; players take turns marking one cell at a time. The difference is that instead of placing Xs and Os on the grid, the players place the numbers 1 to 9. And, instead of trying to make a row, column, or diagonal filled with 3 Xs or 3 Os, the players try to make a row, column, or diagonal whose cells sum to 15. One wrinkle: each number can only be placed ONCE. Play a couple games with a classmate.

Let's think about implementing this game. (For now, we won't use classes; everyone has different amounts of experience with them, so for now we'll just use functions.) We have a couple questions to think about before we start coding:

1) How should we represent the board? You might be tempted to say "a two-dimensional array!" But C++'s multi-dimensional arrays cause great confusion even among experienced programmers, so I stay away from them. In this case, a simple way to start is to represent the board as a 9-element array of `int`.

2) And what behaviors (functions) do we need? There's infinite variety here, but generally speaking we try to have each function to one small well-defined task, So (in addition to `main()`, of course), we might have

- A function to `getAMove()` from the user (we'll assume that two players are sharing one keyboard).
- A function to `validate()` the input. (What should that function do, exactly?)
- A function to `playAMove()` onto the board
- A function to `display()` the current state of the board.
- A function to `checkForWin()`.

Notice I haven't given you the complete function signatures, and I certainly haven't described how they should be used (are they all called from `main()`? Or do some of these call the others?)

Part 2: Design

Using principles of modular design, organize **these functions** into modules. Again, you have some choices—which functions belong together in one module? Here's a hint: which functions need to deal with user input? Which functions need to deal with the gameboard?

For this application activity, *design* the source/header files you will need to write. You do *not* need to implement any of these functions. But, thinking about interface vs implementation, you should be able to write *function declarations* separately from the *function definitions*. What other design decisions can you make?

What if you need to change the way the board is represented (say you realize that an array of `int` isn't going to work very well)? Your design should anticipate this kind of change, and minimize the impact (that is, minimize the amount of *code* that has to be changed to accommodate the new representation).

Write complete `.h` files and a complete `makefile` for the project (another opportunity practice using `vim` and/or `emacs`!). Remember, you don't need to *implement* these functions yet, but you *do* need to have a general sense of what their behavior will be.

Email me these files by midnight, Monday September 19.