CSE 303, Spring 2007, Assignment 3
Due: Tuesday 24 April, 9:00AM

Last updated: April 15

You will modify two small C programs. Note: Line counts provide an approximate sense of how much work is required; it is *not* necessary for your solutions to be the same number of lines.

1. Get `triangles.c` from the course website. It defines types for triangles and equilateral triangles (defined by one point, a side-length, and an angle as discussed below), as well as some several functions. You need to add 5 functions and modify one of the existing functions as described below.

   To use the math functions needed for part (e), you should compile like this (the `-lm` is important and must be last): `gcc -Wall -o triangles triangles.c -lm`

   (a) `newShiftedTri2` should behave just like `newShiftedTri1` (creating a new heap-allocated triangle moved over by amounts `delta_x` and `delta_y`), but it should use `initTri` as a helper function. Sample solution is 8 lines long, but only 3 statements.

   (b) `shiftPoint` should take two pointers to 2D-points, two doubles `delta_x` and `delta_y`, and have return type `void`. It should update the fields pointed to by the first argument to be the fields pointed to by the second argument shifted by the delta amounts. Sample solution is 2 lines.

   (c) `newShiftedTri3` should behave just like `newShiftedTri1` but it should use `shiftPoint` as a helper function (multiple times). Sample solution is 8 lines. Hint: You need to use the address-of operator on a field of a struct, which is a combination of features you did not see in class.

   (d) `sameTriangle` has been started. It should return 1 if the two arguments point to the "same triangle" (*not* in the pointer-equality sense). Notice we use the `samePoint` helper function which allows for small variations, which is important because floating-point computations can have rounding errors. Extend `sameTriangle` so it returns 1 if the two triangles have the same points even if the points are in different fields for the two triangles. Sample solution is 6 more lines.

   (e) `fromEqui` takes a pointer to an equilateral triangle and returns a pointer to a new heap-allocated triangle that describes the same geomtric object. The `#include <math.h>` provides definitions for `M_PI` (i.e., π), `cos`, and `sin` that you should use. An equilateral triangle is defined as follows:

   - One point is in `origin`.
   - A second point is distance `length` away from the first point in direction `angle` radians.
   - A third point is distance `length` away from the first point in direction `angle` + $\pi/3$ radians.

   The formulas $x = r\cos\ \theta$ and $y = r\sin\ \theta$ should be all the geometry you need. Sample solution is 10 lines.

   (f) `sameEqui` should take pointers to two equilateral triangles and return 1 if the two arguments point to the "same triangle". Use `fromEqui` and `sameTriangle` as helper functions. Do not leak space. Sample solution is 8 lines.

   Advice: You can comment out parts of `triangle_test` to test some of your functions before you write the others. Compare against sample output on the course website, but small rounding differences are possible and other tests may be wise.

2. Get `boggle_helper.c` from the course website. It provides a correct (as far as we know) program for helping a Boggle (TM) player find words in a grid. (Boggle is a very simple game; the goal is to find words be starting anywhere in a grid of letters and "moving" one space at a time.) You will make some useful changes and additions to the provided code. They can be done in any order, but (a) and (b) require similar techniques and (c) and (d) require some common changes.

   Description: The program takes 2 arguments, a grid-size and a file-name. The file should contain a square "grid" of English characters with side-length grid-size. The program reads the grid into a heap-allocated array of array of characters and then repeatedly prompts the user (on stdout) for a word (on stdin). It looks for the word in the grid, starting at any position and at each "move" going up, down, left, or right. It prints "found" and the starting row/column or "word not found". A blank input line causes the program to exit. (If a word appears more than once, it just finds one place.)

Changes:

(a) Before prompting for any words, exit the program with an appropriate message if a character in the grid is not an English letter. Also, if it is a capital letter, change it lower case (so that we are case-insensitive).

(b) Before searching for a word provided by the user, check if it has any non-English characters (and if so print an appropriate message and immediatly prompt for a new word). Also, convert any capital letters to lower case (so that we are case-insensitive).

(c) Change the output for a found word to include the full list of grid positions that "spell out" the word. For example, an output might look like: `found: (2,3) (3,3) (3,4) (4,4)`. For what is printed, rows and columns are counted from 1 starting at the top and left.

(d) Change the search so that it does *not* allow using the same grid position more than once in a word (like in Boggle).

(e) Change the search so that it *does* allow moving in any of the four diagonal directions (like in Boggle).

Advice:

- Understand the code provided to you. Do not just start hacking it up.
- You can use `man` to learn about C library functions you are unfamiliar with (e.g., `man fgets`).
- Parts (a) and (b) require loops in the right places. Use library functions `isalpha` and `tolower`.
- For parts (c) and (d), *instead* of passing around the starting row and column, pass around pointers to *arrays* where the i^{th} elements are the row and column visited on the i^{th} step of the search. You can just use two arrays for the whole search for one word; their size need not be bigger than the length of the word (or alternately, the number of characters in the grid). For part (c) you can use this information directly. For part (d), check at each step that "where you are" is "somewhere you have not already been". For both, update one position of the array before recurring. Be sure to deallocate the arrays you create whether or not the word is found.
- For part (e), make two small related changes.

3. **Extra Credit:** Extend problem 2 to print all the "found words" before exiting (when the user enters a blank line). However, do not print the same word more than once and do not print a word if it is a prefix of another found word. Your solution should not use space unnecessarily and should be correct no matter how many words the user finds.

Assessment: Your solutions should be:

- Correct C programs that compile without warnings using `gcc -Wall`.
- In good style, including indentation and line breaks
- Of reasonable size

Turn-in: Use the `turnin` command (man turnin) for course cse303 and project hw3. In particular, type:

```
turnin -ccse303 -phw3 triangles.c boggle_helper.c
```

If you use one late-day (see the syllabus) use the project hw3late1 instead of hw3 and similarly hw3late2 for two late days. If you do the extra credit, turn in a different program appropriately named.