CSE 303, Winter 2006, Assignment 3
Due: Friday 3 February, 9:00AM

Last update: January 24

You will modify some small C programs.

1. Get `points.c` from the course website. It defines types for points and lines, some simple functions, a test function, and `main`. You need to add six functions as follows:

   - `intersect1` takes two lines (arguments of type `line_t` i.e., `struct Line2D`) and returns a point (of type `point_t` i.e., `struct Point2D`). The point should be the point on both lines. (Do not worry about parallel lines; floating-point computations should "correctly" use infinity and negative-infinity.)
   - `line_containing` takes two points and returns the line containing both points. (Do not worry about the two points being the same. You will get an answer, but it won't mean much.)
   - `intersect2` takes two *pointers to lines* and returns a *pointer to a point* that holds the intersection. Use `malloc` to heap-allocate a new point for the result to point to. Do not call any helper functions (except `malloc`).
   - `intersect3` behaves just like `intersect2` except it uses `intersect1` as a helper function. It should *not* read any fields of points or lines directly.
   - `intersect4` has return-type `void` and takes three arguments: first a pointer to a point and then two pointers to lines. It mutates the pointed-to point to hold the intersection of the two lines.
   - `intersect5` takes the same arguments as `intersect4` but has return-type `int`. If the slopes of the two lines differ by less than 10^{-10}, then it returns 0 and does not modify the pointed-to point. Otherwise it behaves like `intersect4` and returns 1. (We are essentially checking for parallel lines. Because of rounding errors, floating-point numbers should almost never be tested for equality.)

   Advice: You can comment out parts of `points_test` to test some of your functions before you write the others. No function body needs to be longer than a few lines. The correct output is below. (`XXX` represents numbers that are unpredictable. Each answer of `(Inf,Inf)` could also be `(-Inf,-Inf)`.)

```
y = -1 x + -3
y = 0 x + 3
y = 1 x + 0
y = 1 x + 1
(0,0)
(5,5)
(0,1)
(5,6)
(-6,3)
(Inf,Inf)
(-6,3)
(Inf,Inf)
(-6,3)
(Inf,Inf)
(-6,3)
(Inf,Inf)
(-6,3)
(XXX,XXX)
(XXX,XXX)
ans1=1, ans2=0
```

2. Get `wordfind.c` from the course website. It contains a correct (as far as we know) C program that solves "word finds". You will make 3 changes to this file (see "Changes" below). Note you can do the changes in any order.

   Description: The program takes 3 arguments: (1) a grid-height, (2) a grid-width, and (3) a file-name. The file should contain a "grid" of characters, i.e., grid-height lines, each containing grid-width characters. The program reads the grid into memory (specifically 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 proceeding up, down, left, or right), and prints either "found" or "not found". A blank line or a word too long to fit in the grid causes the program to exit.

   If the input file has any extra lines they are ignored. Lines longer than necessary may lead to the program malfunctioning.

   Changes:

   (a) Change the expected format of the input file. Specifically, expect there to be a space after every character and a blank line after every line of characters. Do *not* put the spaces and blank lines into the grid. Instead, expect the input lines to be longer (and on every other line), but have the `make_grid` function process the input by skipping over the blanks.

   (b) Improve the word-finding to look also for diagonal words (in all four diagonal directions).

   (c) Change the output when a word is found to include the starting position and direction of the word in the grid. Three example outputs would be:

   ```
   found: row 2 column 3 up left
   found: row 1 column 12 down
   found: row 7 column 9 right
   ```

   The row and column are the position of the first letter in the found word. The first line of the grid-file is row 1, the second line row 2, etc. Columns also "start at 1" and proceed left-to-right. The direction is 1 of 8 possibilities. An extra space or two in your output is fine.

   Advice: Understand the code provided to you. You can use `man` to learn about C library functions you are unfamiliar with (e.g., `man fgets`). For the first change, do not use `strncpy`; write your own loop instead. The second change requires *very* little code. For the third change, compute whether you want to print "up, down, or nothing" and separately whether you want to print "left, right, or nothing". Then use a single call to `fprintf` for the whole output line. You can put all your new code for the third change in the if-statement in `find_word`.

   Your instructor found `http://puzzlemaker.school.discovery.com/WordSearchSetupForm.html` fun and easy to use. The output is not quite in the format you want, but you could use `sed` to fix that.

3. **Extra Credit:** Copy your solution to problem 2 to `wordfind_extra.c` and then make this change: No longer take command-line arguments for the width and height. Take just the file-name and determine the width and height based on the file contents. Your solution must be safe even for very large widths and heights, and it must not have space leaks. Advice: "Make an initial guess" for the grid size and then "make a bigger grid and free the old one" as necessary.

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 Instructions: Use `turnin` for course cse303 and project hw3. If you use late-days, use project hw3late1 (for 1 late day) or hw3late2 (for 2) instead of hw3.