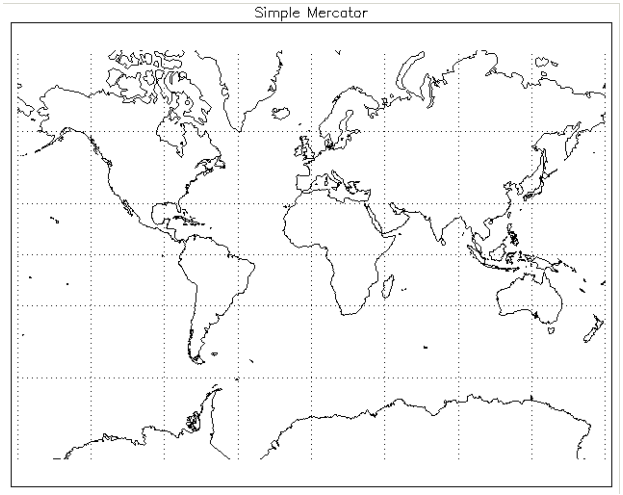
Parsing Input and Formatted Output in C Dealing with Geographic Coordinates

You will provide an implementation for a complete C program that reads geographic coordinates from an input file, does some simple computations with them, and writes precisely-formatted output to a file.

The geographic coordinates will be represented using longitude and latitude values. If you are unfamiliar with the concept, a good quick explanation can be found at http://astro.unl.edu/naap/motion1/tc_units.html. In essence, you can think of each point on the globe as being specified by a longitude/latitude pair, much as a point in the plane is specified by xy-coordinates. But, rather than specifying decimal fractions, we usually express the coordinate in terms of degrees, minutes and seconds, where a degree is divide into 60 minutes and a minute is divided into 60 seconds. The units degrees, minutes and seconds are usually represented by the symbols º, ' and ". But here we will use d, m and s, respectively.

Longitude and latitude are calculated with respect to a fixed "origin"; the equator defines latitude 0d 0m 0s and the prime meridian defines longitude 0d 0m 0s (approximately that of Greenwich, England). So this "origin" lies somewhere in the Gulf of Guinea off the west coast of Africa. By convention, points west of the prime meridian have negative longitude, and points east of the origin have positive longitude. Similarly, points north of the equator have positive latitude, and points south of the equator have negative latitude. So, we can (almost) imagine flattening the globe onto a plane, and think of longitude and latitude as x and y coordinates, respectively:



http://northstar-www.dartmouth.edu/doc/idl/html_6.2/Cylindrical_Projections.html

The drawing above is incomplete; it does not show the extreme north or south latitude regions. In fact, it's not possible to do so with this projection. Longitude values range from -180d to 180d, which both correspond to the same position in the middle of the Pacific Ocean. Latitude values range from -90d to 90d.

For example, McBryde Hall is at the coordinates:

longitude: 80d 25m 19s W
latitude: 37d 13m 49s N

For storage efficiency, longitude and latitude values are often represented in DMS format. For McBryde Hall:

longitude: 0802519W
latitude: 371349N

Of course, these values could also be represented in other ways, as total seconds or as decimal degrees.

Input Files

Your program will read an input file that contains a sequence of lines, each containing two longitude/latitude pairs; in other words, each of these lines specifies the locations of two points on the globe. The first line in the file will specify how many lines of such coordinate data are to be processed (could be less than the number of lines in the file):

```
10
1042239W   324630N    1042612W   320909N
1043816W   324219N    1042517W   324940N
1042700W   323522N    1040758W   320901N
1040145W   324401N    1040933W   323253N
1041656W   322638N    1042332W   322307N
1043656W   321140N    1040553W   320240N
1044529W   321623N    1044109W   320226N
1044825W   320203N    1044235W   320945N
1042308W   325004N    1041611W   322207N
1044652W   321758N    1035303W   324921N
```

The longitude and latitude values are separated by single tab characters; the last character in the second latitude value will be followed immediately by a newline character. That information may be useful in deciding how to write C code to read the values (or not, depending on your approach).

Program Requirements

Your program will be executed from a CentOS command line as follows (assuming the executable is named `c02`):

```
CentOS> c02 <name of input file> <name of output file>
```

The source code for the test driver given with the previous C assignment illustrates dealing with taking file names from the command line, as well as a number of other useful things; you are encouraged to examine that file for inspiration.

Your program must read each line of data from the input file and process that data. The processing involves parsing the given DMS representations for the longitude and latitude values, and printing those values in a more human-friendly format. For example:

```
1042239W --> 104d 22m 39s W
```

There are a number of ways to handle this; study the various options for format specifiers that can be used with `fscanf()` and its relations in the C Standard Library. You can save yourself a lot of pain if you find an efficient way to do this, rather than trying to process the input character by character.

You must also compute the distance between the two points, according to the taxicab metric. In short, the taxicab distance between two points is the length of the shortest path between the points, where the path consists of line segments that are either vertical (along a line of longitude) or horizontal (along a line of latitude). You must then report that distance, both in total seconds[1], and in a human-friendly DMS format. See the output file example below.

Aside from those requirements, your implementation must be in a single C source file, and you must make sensible use of user-defined functions in your solution; some aspects of this assignment will probably find their way into future assignments, and useful functions you design here may be useful later as well. Make useful comments in your implementation. The same general guidelines for commenting that you have been taught in your Java courses should provide sufficient guidance.

Output Files

Here is the output file your program should produce from the input file shown above:

```
First coordinate                  Second coordinate                  seconds     DMS
-------------------------------------------------------------------------------------------
(104d 22m 39s W, 32d 46m 30s N)   (104d 26m 12s W, 32d 09m 09s N)      2454     0d 40m 54s
(104d 38m 16s W, 32d 42m 19s N)   (104d 25m 17s W, 32d 49m 40s N)      1220     0d 20m 20s
(104d 27m 00s W, 32d 35m 22s N)   (104d 07m 58s W, 32d 09m 01s N)      2723     0d 45m 23s
(104d 01m 45s W, 32d 44m 01s N)   (104d 09m 33s W, 32d 32m 53s N)      1136     0d 18m 56s
(104d 16m 56s W, 32d 26m 38s N)   (104d 23m 32s W, 32d 23m 07s N)       607     0d 10m 07s
(104d 36m 56s W, 32d 11m 40s N)   (104d 05m 53s W, 32d 02m 40s N)      2403     0d 40m 03s
(104d 45m 29s W, 32d 16m 23s N)   (104d 41m 09s W, 32d 02m 26s N)      1097     0d 18m 17s
(104d 48m 25s W, 32d 02m 03s N)   (104d 42m 35s W, 32d 09m 45s N)       812     0d 13m 32s
(104d 23m 08s W, 32d 50m 04s N)   (104d 16m 11s W, 32d 22m 07s N)      2094     0d 34m 54s
(104d 46m 52s W, 32d 17m 58s N)   (103d 53m 03s W, 32d 49m 21s N)      5112     1d 25m 12s
```

The correctness of your solution will be determined by comparing the output your program produces on selected test data files with the output produced by the reference solution (which produced the output shown above). Formatting matters to some extent:

- you must spell things correctly; capitalization matters
- you must have whitespace (spaces are suggested, not tabs) where whitespace is shown
- you must not have whitespace where none is shown

The comparisons will be done with a tool that treats each line of output as a sequence of strings, and compares the strings. The exact nature of the whitespace does not matter; that is, having a different number of spaces, or even using tab characters, is OK.

Getting Started

A tar file is available, containing the testing/grading code that will be used to grade your solution:

`generator`	64-bit CentOS test data generator
`GISdata.txt`	file of GIS records; used by `generator`
`c02prof`	64-bit CentOS reference solution; use with files created by `generator` to create reference output files to use with `compare`
`compare`	64-bit CentOS tool to compare reference output files from `c02prof` to output files created by your solution

Download the tar file `C02Files.tar` from the course website and save it on your CentOS 7 installation (or on rlogin), in a directory created for this assignment. Unpack the tar file there. You can do this by executing the command:

```
CentOS> tar xf C02Files.tar
```

You can use `generator` to create input files for testing:

```
generate <# test cases> GISdata.txt <test data file>
```

For example:

```
CentOS> ./generate 10 GISdata.txt TestCases.txt
```

Once you've written a draft solution and compiled it, you can use the test data file from `generator` to see how your solution is performing. Let's assume you've named the executable compiled from your solution `c02`. Then, you can copy your test file, say `TestCases.txt`, into the same directory as `c02`, and execute your program:

```
CentOS> ./c02 TestCases.txt myResults.txt
```

Now, it's possible you will experience runtime errors or simply not produce any useful output at first; in that case, you will need to debug your solution. Once you are producing sensible output, you can check its correctness. First, run the reference solution on the input file created by `generator`:

```
CentOS> ./c02prof TestCases.txt profResults.txt
```

This will create the file `profResults.txt`, which contains the correct output for the given input. The lines in this file will also be tagged with point annotations (e.g., `[20]`), which will be used by the `compare` tool. Then, run the `compare` tool to see how you've done, say by executing:

```
CentOS> ./compare 1 profResults.txt myResults.txt
```

This should produce an output file named `comparison.txt`; examine that file to see how your output would be scored. Diagnose and fix errors, and try again. (Be sure to include the '1' on the command line; the comparison tool is designed to be called sequentially on a number of different test sets; we will use that feature in later assignments.)

As you already know, no single passed test demonstrates that your solution is fully correct. Nor will any number of passed tests, in most cases. Therefore, you should be sure to follow the testing process described above with a reasonable number of different test case files, created by `generator`.

Submission and Grading

You should not submit your solution to the Curator until you can correctly pass tests with the given testing/grading code.

Submit your completed C source file, after careful debugging and testing. It doesn't matter what you name your source file; the Curator renames submissions. Your submission will be compiled, tested and graded by using the supplied tools, but that will be done manually after the due date. A TA will also check to see if your solution violates any of the restrictions given in the header comment for the function; if so, your submission will be assigned a score of zero (0), regardless of how many tests it passes.

If you make multiple submissions of your solution to the Curator, we will grade your last submission. If your last submission is made after the posted due date, a penalty of 10% per day will be applied.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found at:

http://www.cs.vt.edu/curator/

Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
//  On my honor:
//
//  - I have not discussed the C language code in my program with
//    anyone other than my instructor or the teaching assistants
//    assigned to this course.
//
//  - I have not used C language code obtained from another student,
//    the Internet, or any other unauthorized source, either modified
//    or unmodified.
//
//  - If any C language code or documentation used in my program
//    was obtained from an authorized source, such as a text book or
//    course notes, that has been clearly noted with a proper citation
//    in the comments of my program.
//
//  - I have not designed this program in such a way as to defeat or
//    interfere with the normal operation of the grading code.
//
//    <Student Name>
//    <Student's VT email PID>
```

We reserve the option of assigning a score of zero to any submission that is undocumented or does not contain this statement.

Notes

[1] The actual arc length of a degree of longitude varies from about 111.3 km at the equator to a limiting value of 0 at the poles; similarly, the actual arc length of a degree of latitude varies from about 110.6 km at the equator to about 111.7 km at the poles. So, the value you are computing doesn't really convey very precise information.

Change Log

Any changes or corrections to the specification will be documented here.

Version	Posted	Pg	Change
1.00	TBA		Base document.