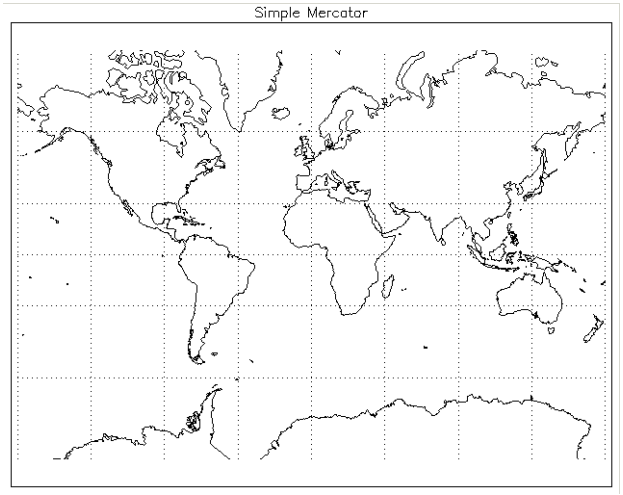
Parsing Input and Formatted Output in C Dealing with Geographic Coordinates

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 (south pole) to 90d(north pole).

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:

	total seconds	decimal degrees
longitude:	-289519	-80.421944
latitude:	134029	37.230277

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:

```
(1043408W, 323223N)   (1041614W, 322806N)
(1042311W, 324522N)   (1044700W, 322107N)
(1042410W, 325018N)   (1045001W, 323905N)
(1040324W, 324707N)   (1040127W, 323935N)
(1041133W, 323039N)   (1042920W, 320711N)
(1041312W, 324758N)   (1044543W, 323228N)
(1042301W, 325032N)   (1044349W, 321516N)
(1041007W, 322104N)   (1043954W, 320824N)
(1041841W, 321604N)   (1041342W, 322514N)
(1041408W, 320737N)   (1035852W, 320721N)
```

Each longitude/latitude pair will be formatted exactly as shown above. The longitude value is preceded by a left parenthesis, and followed by a comma and a single space, and the latitude value is followed by a right parenthesis. Longitude values will be formatted as a sequence of 7 decimal digits (0-9), possibly with a leading 0, and followed by a single character ('W' or 'E'). Latitude values will be formatted as a sequence of 6 decimal digits, possibly with a leading 0, and followed by a single character ('N' or 'S').

The pairs are separated by a single tab character, and the final parenthesis is followed immediately by a newline character. You should use this information when deciding how to write C code to read the values on each line of the input file.

No information is give about the number of lines of data in the input file; your program must be designed so that it correctly reads all the given data lines and stops after the last line has been read.

Advice on Reading the Input File

The correct pattern is known as *read to input failure*.

Do not depend on testing for an *end-of-file* condition. It is true that the C Standard Library specifies conditions under with the end-of-file indicator is to be set. However, testing for EOF is logically inferior in this case. A robust design would not only stop reading if the end-of-file indicator has been set, but also stop reading if the input data did not match the specified format.

Remember that `fscanf()`, and related input functions, have a return value:

> Upon successful completion, these functions return the number of successfully matched and assigned input items; this number can be 0 in the event of an early matching failure.

The read to input failure pattern terminates reading whenever the return value from reading does not match our expectations:

```
while ( call_to_read_function(. . .) == expected_value ) {

   process the data that was just read
}
```

In this case, it's possible to write a single `fscanf()` call, taking advantage of the options for format strings, that will take a single line of the input file, separate out the parts you want from the parts you don't want, and store those parts into variables you've declared. That's exactly how I handled this.

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 34m 08s W, 32d 32m 23s N)   (104d 16m 14s W, 32d 28m 06s N)      1331     0d 22m 11s
(104d 23m 11s W, 32d 45m 22s N)   (104d 47m 00s W, 32d 21m 07s N)      2884     0d 48m 04s
(104d 24m 10s W, 32d 50m 18s N)   (104d 50m 01s W, 32d 39m 05s N)      2224     0d 37m 04s
(104d 03m 24s W, 32d 47m 07s N)   (104d 01m 27s W, 32d 39m 35s N)       569     0d 09m 29s
(104d 11m 33s W, 32d 30m 39s N)   (104d 29m 20s W, 32d 07m 11s N)      2475     0d 41m 15s
(104d 13m 12s W, 32d 47m 58s N)   (104d 45m 43s W, 32d 32m 28s N)      2881     0d 48m 01s
(104d 23m 01s W, 32d 50m 32s N)   (104d 43m 49s W, 32d 15m 16s N)      3364     0d 56m 04s
(104d 10m 07s W, 32d 21m 04s N)   (104d 39m 54s W, 32d 08m 24s N)      2547     0d 42m 27s
(104d 18m 41s W, 32d 16m 04s N)   (104d 13m 42s W, 32d 25m 14s N)       849     0d 14m 09s
(104d 14m 08s W, 32d 07m 37s N)   (103d 58m 52s W, 32d 07m 21s N)       932     0d 15m 32s
```

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:

`generate`	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 copying the posted tar file into the directory, then 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, using the same, fixed set of test cases for each submission.

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.

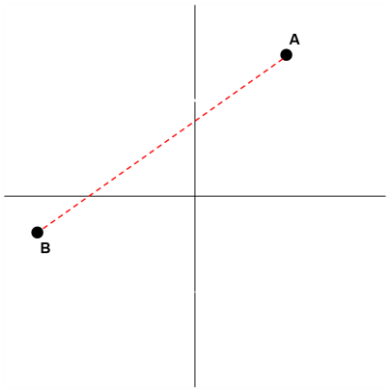
Notes

[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, because the Earth is not exactly spherical, 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.

Appendix Taxi-cab Metric

A *metric function* (or simply a *metric*) defines the distance between pairs of elements in a set. For a given set of elements, there may be a number of different ways to define a metric on that set; which one we use depends on how we want to think about distance.

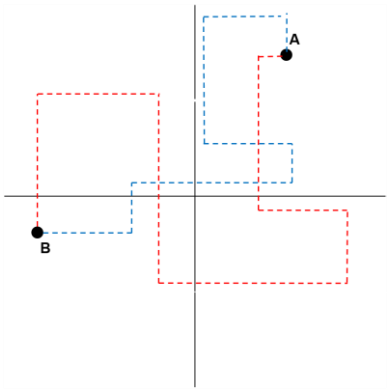
You recall from math classes that we usually define the distance between two points in the xy-plane as the length of the line connecting those two points:



$$d = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

This is the *Cartesian metric* for the xy-plane.

A taxi-cab path in the xy-plane is a connected sequence of line segments, each of which is parallel to either the x-axis or the y-axis:



Of course, given two points, there are infinitely many different taxi-cab paths from one to the other, some longer than others.

The taxi-cab metric on the xy-plane defines the distance between two points as the minimum length of all taxi-cab paths between the two points. There's a simple way to define a function to compute the taxi-cab distance; doing so is part of the assignment, so we won't give such a function here.

Change Log

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

Version	Posted	Pg	Change
3.00	Aug 26		Base document.