Meet Structs

For this exercise, you get to work with structs. You'll be defining a new struct type, dynamically allocating an array of that type of struct, filling the array with input from a file and then using it to answer a sequence of queries from the user. On the course homepage, you'll find a source file named gps.c, along with some input files and expected outputs for this exercise. You can also download these with the following curl commands:

curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise13/gps.c
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise13/city-1.txt
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise13/expected-1.txt
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise13/city-2.txt
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise13/expected-2.txt

Program Input

The gps.c program expects a single command-line argument, the name of an input file containing a list of cities, each with a global location. The input file starts with an integer, n, telling you how many cities follow. Each city is given as a name, followed by two real numbers giving the latitude and longitude of the city. We're proving two sample input files, city-1.txt just contains two cities:

```
2
Raleigh 35.817 -78.65
Tokyo 35.683 139.7
```

The city-2.txt file is a little bit longer, with locations for 10 cities:

```
10
Chicago 41.883 -87.633
New-York 40.667 -73.933
Houston 29.767 -95.383
Los-Angeles 34.05 -118.25
Philadelphia 39.95 -75.167
Raleigh 35.817 -78.65
Phoenix 33.45 -112.067
Seattle 47.617 -122.333
Miami 25.783 -80.217
Denver 39.733 -104.983
```

Sample Execution

When it starts up, your program will read the list of cities into a dynamically allocated array of structs (some of this code is already written for you). Then, it will repeatedly read query locations from the user, given as latitude and longitude. For each query, it will report the city that's closest to the location, along with the distance to that city in miles. Running your program as follows should give you the same output as expected-1.txt (typing quit at the end terminates the program, since it stops when it gets input that doesn't parse as two doubles):

```
$ ./gps city-1.txt
40.667 -73.933
Raleigh 421.56
37.567 126.983
Tokyo 716.54
quit
```

Or, you can try running your program on the larger city list;

```
$ ./gps city-2.txt
33.75 -84.383
Raleigh 355.24
36.183 -115.133
Los-Angeles 229.67
```

```
-11.95 -77.033
Miami 2616.04
quit
```

Compiling Your Solution

This program uses a few functions from the math library. When you compile, you'll need to add the -lm option to tell gcc to link with the math library, and you'll need the -D_GNU_SOURCE compiler flag to tell gcc to let you use the M_PI constant for pi (which is commonly supplied by the math.h header, but isn't officially part of the C99 standard).

```
$ gcc -D_GNU_SOURCE -Wall -std=c99 gps.c -o gps -lm
```

Completing the Implementation

In the gps.c file, I've already written some code to help you out. I've marked the places where you'll need to fill in missing code with comments containing "...". Specifically, you'll need to add code for the following:

- Define a struct named City, for holding information about a city. Your struct will need to be able to store a name for the city (a string up to 20 characters in length). It will also need two double fields for storing the city's location, as latitude and longitude.
- Fill in the body of the readCity() function. It should read the name and location of a city from a file and store it in a city struct (both of these are parameters to the function). The city name won't contain any spaces, and you won't need to worry about detecting invalid input (e.g., a city name that's too long or a location that isn't given as a pair of double values). We won't run your program with invalid input files.
- In main(), add code to dynamically allocate an array of City structs, with enough elements to hold all the cities in the input file. I've already written code to read the first integer, n, from the input file. You'll need to declare a pointer-to-city variable to keep up with the location of your dynamically allocated array.
- Right after you make your array of City structs, add code to loop through the array and call your readCity() function to read in each of them. You defined readCity() to take a pointer to a City as its second parameter. Each time you call it, you need to give it the address of one of the elements of your dynamically allocated array. You could do this with pointer arithmetic, if you want.
- Near the end of main, there's a loop to read query locations from the user. Inside this loop, add code to look at all the cities and report the one that's closest to the query location. As in the sample executions, print a line that gives the city name, then a space, then the distance to that city in miles, rounded to the nearest hundredth of a mile. I've provided a function, globalDistance(), to help you out. If you give it two locations expressed in latitude and longitude, it will return the approximate distance between them (assuming the world is a perfect sphere, which it isn't).
- Finally, don't forget to free your dynamically allocated array before you exit.

Once your program is working, submit it to the exercise_13 assignment in Moodle.