C Programming Structured Types, Files and Parsing, Indexing

Geographic information systems organize information pertaining to geographic features and provide various kinds of access to the information. A geographic feature may possess many attributes (see below). In particular, a geographic feature has a specific location. There are a number of ways to specify location. For this project, we will use latitude and longitude, which will allow us to deal with geographic features at any location on Earth. A reasonably detailed tutorial on latitude and longitude can be found in the Wikipedia at en.wikipedia.org/wiki/Latitude and en.wikipedia.org/wiki/Longitude.

The GIS record files were obtained from the website for the USGS Board on Geographic Names (geonames.usgs.gov). The file begins with a descriptive header line, followed by a sequence of GIS records, one per line, which contain the following fields in the indicated order:

Figure 1: Geographic Data Record Format

Name	Type	Length/ Decimals	Short Description
Feature ID	Integer	10	Permanent, unique feature record identifier and official feature name
Feature Name	String	120	
Feature Class	String	50	See Figure 3 later in this specification
State Alpha	String	2	The unique two letter alphabetic code and the unique two number code for a US State
State Numeric	String	2	
County Name	String	100	The name and unique three number code for a county or county equivalent
County Numeric	String	3	
Primary Latitude DMS	String	7	The official feature location *DMS-degrees/minutes/seconds* *DEC-decimal degrees.* *Note: Records showing "Unknown" and zeros for the latitude and longitude DMS and decimal fields, respectively, indicate that the coordinates of the feature are unknown. They are recorded in the database as zeros to satisfy the format requirements of a numerical data type. They are not errors and do not reference the actual geographic coordinates at 0 latitude, 0 longitude.*
Primary Longitude DMS	String	8	
Primary Latitude DEC	Real Number	11/7	
Primary Longitude DEC	Real Number	12/7	
Source Latitude DMS	String	7	Source coordinates of linear feature only (Class = Stream, Valley, Arroyo) DMS-degrees/minutes/seconds DEC-decimal degrees. *Note: Records showing "Unknown" and zeros for the latitude and longitude DMS and decimal fields, respectively, indicate that the coordinates of the feature are unknown. They are recorded in the database as zeros to satisfy the format requirements of a numerical data type. They are not errors and do not reference the actual geographic coordinates at 0 latitude, 0 longitude.*
Source Longitude DMS	String	8	
Source Latitude DEC	Real Number	11/7	
Source Longitude DEC	Real Number	12/7	

Elevation (meters)	Integer	5	Elevation in meters above (-below) sea level of the surface at the primary coordinates
Elevation (feet)	Integer	6	Elevation in feet above (-below) sea level of the surface at the primary coordinates
Map Name	String	100	Name of USGS base series topographic map containing the primary coordinates.
Date Created	String		The date the feature was initially committed to the database.
Date Edited	String		The date any attribute of an existing feature was last edited.

Notes:

- See https://geonames.usgs.gov/domestic/states_fileformat.htm for the full field descriptions.
- The type specifications used here have been modified from the source (URL above) to better reflect the realities of your programming environment.
- Latitude and longitude may be expressed in DMS (degrees/minutes/seconds, 0820830W) format, or DEC (real number, -82.1417975) format. In DMS format, latitude will always be expressed using 6 digits followed by a single character specifying the hemisphere, and longitude will always be expressed using 7 digits followed by a hemisphere designator.
- Although some fields are mandatory, some may be omitted altogether. Best practice is to treat every field as if it may be left unspecified. Certain fields are necessary in order to index a record: the feature ID, the feature name, and the state abbreviation. Every record will always include each of those fields.

In the GIS record file, each record will occur on a single line, and the fields will be separated by pipe (`'|'`) symbols. Empty fields will be indicated by a pair of pipe symbols with no characters between them. See the posted `VA_Highland.txt` file for many examples.

GIS record files are guaranteed to conform to this syntax, so there is no explicit requirement that you validate the files. On the other hand, some error-checking during parsing may help you detect errors in your parsing logic.

The file can be thought of as a sequence of bytes, each at a unique offset from the beginning of the file, just like the cells of an array. So, each GIS record begins at a unique offset from the beginning of the file.

Line Termination

Each line of a text file ends with a particular marker (known as the line terminator). In MS-DOS/Windows file systems, the line terminator is a sequence of two ASCII characters (CR + LF, `0X0D0A`). In Unix systems, the line terminator is a single ASCII character (LF). Other systems may use other line termination conventions.

Why should you care? Which line termination is used has an effect on the file offsets for all but the first record in the data file. As long as we're all testing with files that use the same line termination, we should all get the same file offsets. But if you change the file format (of the posted data files) to use different line termination, you will get different file offsets than are shown in the posted log files. Most good text editors will tell you what line termination is used in an opened file, and also let you change the line termination scheme.

All that being said, when this project is graded, Unix line termination will be used, which will also be true of all the test data files supplied for the project.

Figure 2: Sample Geographic Data Records

Note that some record fields are optional, and that when there is no given value for a field, there are still delimiter symbols for it.

Also, some of the lines are "wrapped" to fit into the text box; lines are never "wrapped" in the actual data files.

```
FEATURE_ID|FEATURE_NAME|FEATURE_CLASS|STATE_ALPHA|STATE_NUMERIC|COUNTY_NAME|COUNTY_NUMERIC|PRIMARY_LAT_DMS|PRIM_LONG_DMS|PRIM_LAT_DEC|PRIM_LON
G_DEC|SOURCE_LAT_DMS|SOURCE_LONG_DMS|SOURCE_LAT_DEC|SOURCE_LONG_DEC|ELEV_IN_M|ELEV_IN_FT|MAP_NAME|DATE_CREATED|DATE_EDITED
1479116|Monterey Elementary School|School|VA|51|Roanoke (city)|770|371906N|0795608W|37.3183753|-
79.9355857|||||323|1060|Roanoke|09/28/1979|09/15/2010
1481345|Asbury Church|Church|VA|51|Highland|091|382607N|0793312W|38.4353981|-79.5533807|||||818|2684|Monterey|09/28/1979|
1481852|Blue Grass|Populated Place|VA|51|Highland|091|383000N|0793259W|38.5001188|-79.5497702|||||777|2549|Monterey|09/28/1979|
1481878|Bluegrass Valley|Valley|VA|51|Highland|091|382953N|0793222W|38.4981745|-79.539492|382601N|0793800W|38.4337309|-
79.6333833|759|2490|Monterey|09/28/1979|
1482110|Buck Hill|Summit|VA|51|Highland|091|381902N|0793358W|38.3173452|-79.5661577|||||1003|3291|Monterey SE|09/28/1979|
1482176|Burners Run|Stream|VA|51|Highland|091|382509N|0793409W|38.4192873|-79.5692144|382531N|0793538W|38.4252778|-
79.5938889|848|2782|Monterey|09/28/1979|
1482324|Mount Carlyle|Summit|VA|51|Highland|091|381556N|0793353W|38.2656799|-79.5647682|||||698|2290|Monterey SE|09/28/1979|
1482434|Central Church|Church|VA|51|Highland|091|382953N|0793323W|38.4981744|-79.5564371|||||773|2536|Monterey|09/28/1979|
1482557|Claylick Hollow|Valley|VA|51|Highland|091|381613N|0793238W|38.2704021|-79.5439343|381733N|0793324W|38.2925|-
79.5566667|573|1880|Monterey SE|09/28/1979|
1482785|Crab Run|Stream|VA|51|Highland|091|381707N|0793144W|38.2854018|-79.528934|381903N|0793415W|38.3175|-79.5708333|579|1900|Monterey
SE|09/28/1979|
1482950|Davis Run|Stream|VA|51|Highland|091|381824N|0793053W|38.3067903|-79.5147671|382057N|0793505W|38.3491667|-79.5847222|601|1972|Monterey
SE|09/28/1979|
1483281|Elk Run|Stream|VA|51|Highland|091|382936N|0793153W|38.4934524|-79.5314362|383121N|0793056W|38.5226185|-
79.5156027|757|2484|Monterey|09/28/1979|
1483492|Forks of Waters|Locale|VA|51|Highland|091|382856N|0793031W|38.4823417|-79.5086575|||||705|2313|Monterey|09/28/1979|
1483527|Frank Run|Stream|VA|51|Highland|091|382953N|0793310W|38.4981744|-79.5528258|383304N|0793341W|38.5512285|-
79.5614381|780|2559|Monterey|09/28/1979|
1483647|Ginseng Mountain|Summit|VA|51|Highland|091|382850N|0793139W|38.480675|-79.527547|||||978|3209|Monterey|09/28/1979|
1483860|Gulf Mountain|Summit|VA|51|Highland|091|382940N|0793103W|38.4945636|-79.5175468|||||1006|3300|Monterey|09/28/1979|
1483916|Hamilton Chapel|Church|VA|51|Highland|091|381740N|0793707W|38.2945677|-79.6186591|||||823|2700|Monterey SE|09/28/1979|
1484097|Highland High School|School|VA|51|Highland|091|382426N|0793444W|38.4071387|-79.5789333|||||879|2884|Monterey|09/28/1979|09/15/2010
1484099|Highland Wildlife Management Area|Park|VA|51|Highland|091|381905N|0793439W|38.3181785|-79.577547|||||954|3130|Monterey SE|09/28/1979|
. . .
```

Assignment

You will implement a system that indexes and provides search features for a file of GIS records, as described above.

Your system will build and maintain several in-memory index data structures to support these operations:

- Retrieving data for all GIS records matching a given feature name and state
- Retrieving data for the unique GIS record matching a given feature ID
- Reporting the distance between two features specified by their feature IDs
- Displaying the in-memory indices in a human-readable manner; this is purely for debugging purposes and will not be evaluated for grading purposes.

You will implement a single software system in C to perform all system functions.

Program Invocation

The program will take the names of two files from the command line: (assuming that your executable is named `gis`):

```
gis <command script file name> <log file name>
```

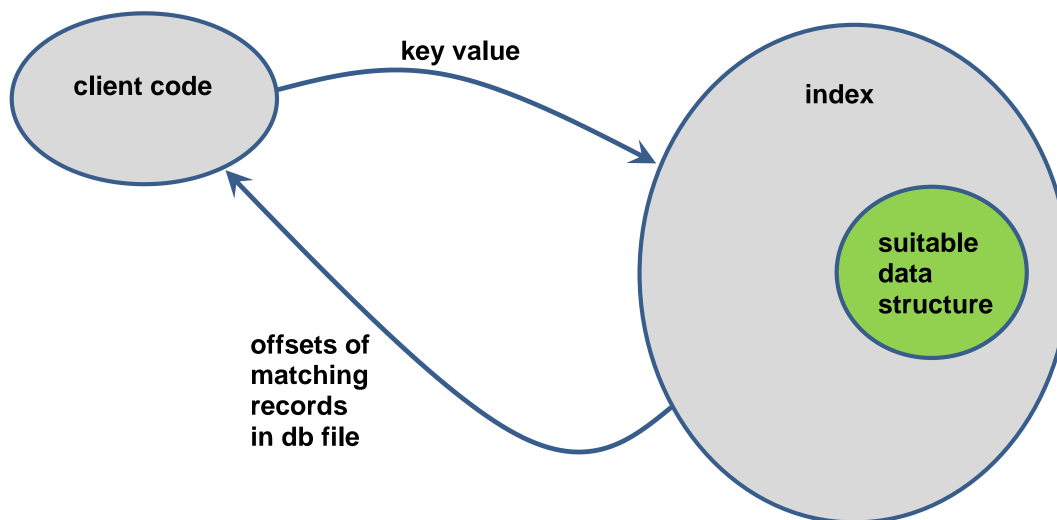
The command-line specifies:

- the name of the file containing the search commands to be carried out, and specifying the name of the database file to be used
- the name of the file to which the program will write its output

The database file will exist, and conform to the format description given above. If the command script file is not found the program should write an error message to the console and exit gracefully. The log file may or may not already exist; if it does not exist, a file with the specified name should be created; if it does exist, the old contents should be replaced.

System Overview

Among other requirements, your solution will implement indexing to support finding records that match certain criteria. The basic notion of an index is that the user gives the index a key value, and the index returns a collection of locations at which matching records can be found, or an indication that no records match the given key:



The client code neither knows, nor needs to know, how the index manages information internally (i.e., what data structure is used); the client only needs to understand the "public" aspect of the transaction. That is, "I send the index a key value of the appropriate type" and "I get back a collection of one or more file offsets in some agreed format".

It's not the job of the index to actually retrieve records for the client. In fact, the index doesn't need to know where the records are stored, or in what format the records are stored.

The GIS records will be indexed by the Feature Name and State (abbreviation) fields. This *name index* will support finding offsets of GIS records that match a given feature name and state abbreviation. It is entirely possible that the file may contain several records that have the same Feature Name and State fields; for example, there are, or were, actually three places in Virginia named Blacksburg.

You will use an array of **`struct`** variables as the physical data structure for the name index; it's up to you to decide exactly what fields those **`struct`** variables will have, but they absolutely will not contain any information except the feature name, the state abbreviation, and the file offset(s) of matching record(s).

The GIS records will also be indexed by their Feature ID fields. The *feature ID index* will support finding offsets of the unique GIS record that matches a given feature ID. The feature ID is a unique identifier; that is, there will never be two different features with the same ID.

You will also use an array of **`struct`** variables as the physical data structure for the feature index; it's up to you to decide exactly what fields those **`struct`** variables will have, but they absolutely will not contain any information except the feature ID and the file offset of matching record.

Each index must support searches that are O(log N), where N is the number of records in the database file; that means you cannot use linear search within an index. Read the discussion of static lookup tables, and general advice, at the end of the specification for specific suggestions on how to do this.

When searches are performed, complete GIS records will be retrieved from the GIS database file that your program maintains. The only complete GIS records that are stored in memory at any time are those that have just been retrieved to satisfy the current search, or individual GIS records created while building the index structures.

Command File

The execution of the program will be driven by a script file. The first line in the file will consist of the name of the database file to be used with the script. Lines beginning with a semicolon character (`';'`) are comments and should be ignored. Blank lines are possible. Each command consists of a sequence of tokens, which will be separated by single tab characters. A line terminator will immediately follow the final token on each line. The command file is guaranteed to conform to this specification, so you do not need to worry about error-checking when reading it.

The commands involve searches related to the indexed records:

```
exists<tab><feature name><tab><state abbreviation>
```

Report the number of records in the database file that match the given feature name and state abbreviation, in the following format:

```
N occurrences:  <feature name>  <state abbreviation>
```

See the posted log files for examples.

```
details_of<tab><feature name><tab><state abbreviation>
```

For each record in the database file that matches the given feature name and state abbreviation, report the values of the following information, with descriptive labels: file offset of record, feature ID, feature name, feature type, state abbreviation, county, primary longitude, and primary latitude.

If more than one record matches the given criteria, list them in ascending order by feature ID. (This may be easier to achieve if you make certain decisions about how to organize your feature name index.)

See the posted log files for examples.

```
distance_between<tab><featureID><tab><featureID>
```

Compute and display the *orthodromic distance* between the two features. See the following section on great circles for instructions on how to do this. The calculation will require a number of trigonometric functions from the C math libraries, so this time we will use the `-lm` switch when doing builds. Round the distance to the nearest tenth of a kilometer when you print it.

If there is no record in the database file for either given feature ID, log an informative message to that effect.

See the posted log files for examples.

The script files are guaranteed to conform to the syntax descriptions given above, so you do not need to worry about checking for such errors.

Your program must echo each comment found in the command file to the log file, and echo each command, labeled and numbered (see posted log files for the format).

Sample command scripts, and corresponding log files, will be provided on the website. As a general rule, every command should result in some output. In particular, a descriptive message must be logged if a search yields no matching records.

Permitted Assumptions and Some Advice

You may make the following assumptions when implementing your solution:

- the GIS database file will never contain more than 200,000 records
- there will never be more than 100 records that match a given Feature Name and State abbreviation
- no GIS record will ever be longer than 300 characters

It would certainly be possible to implement a solution without those assumptions, but it would require using linked lists, or carefully resizing arrays. The former will undoubtedly arise in some future assignment, and the latter will be involved in a later assignment this semester.

The following observations are purely advisory, but are based on my experience, including that of implementing a solution to this assignment. These comments are advice, not requirements.

First, and most basic, analyze what your GIS system must do and design a sensible, logical framework for making those things happen.

Second, and also basic, practice incremental development! This is a fairly sizeable program, especially so if it's done properly. My solution, including comments, approaches 1000 lines of code. It takes quite a bit of work before you have enough working code to test on full input files, but incremental testing is extremely valuable. Try to think of implementing the system feature by feature, and test as you go.

Record your design decisions in some way; a simple text file is often useful for tracking your deliberations, the alternatives you considered, and the conclusions you reached. That information is invaluable as your implementation becomes more complete, and hence more complex, and you are attempting to extend it to incorporate additional features.

Write useful comments in your code, as you go. Leave notes to yourself about things that still need to be done, or that you are currently handling in a clumsy manner.

Take advantage of tools. You should already have a working knowledge of gdb. Use it! The debugger is invaluable when pinning down the location of segfaults; but it is also useful for tracking down lesser issues if you make good use of breakpoints and watchpoints. Some memory-related errors yield mysterious behavior, and confusing runtime error reports. That's especially true when you have written past the end of a dynamically-allocated array and corrupted the heap. This sort of error can often be diagnosed by using valgrind.

Enumerated types are extremely useful for representing various kinds of information, especially about type attributes of structured variables. For example, if implementing a different sort of system, we might find the following type useful:

```
// Vehicle.h
enum _VehicleMake {ACURA, CHEVROLET, DODGE, FORD, . . . , VOLVO};
typedef enum _ VehicleMake VehicleMake;
...
struct _Vehicle {
   VehicleMake  make;
   char*        model;       // could also be an enumerated type
   uint16_t     year;
   char*        licenseNum;
   char*        vin;
   ...
};
typedef struct _Vehicle Vehicle;
...
```

You will need to use static tables of structures to organize language information; by *static*, I mean a table that has static storage duration, and is private to the file in which it's created. In some cases, such a table may be initialized directly, with fixed data, when it is declared. For example:

```
// VehicleData.c
#define NUMRECORDS 50

static Vehicle VehicleTable[NUMRECORDS] = {
   {FORD, "Fiesta", 1978, "LXR 804", . . .},
   {GMC, "Safari", 1997, "ZFL 8473", . . .},
   ...
   {CORD, "812 Westchester", 1937, "PAM 445", . . .}
};
```

In other cases, the table may be declared as a static, file-scoped array, and initialized after your program starts, by reading data from a file. That will be the case for the index structures in this assignment.

Once a static table has been created, we can provide access to it by writing functions that can be called from other parts of the system. For example:

```
// VehicleData.c
uint16_t lookupYearByLicense(char* license) { . . . }
```

Since this function is implemented in the same file as the table, the function can access the table to perform a search. We would put the declaration of the function in a header file, so it can be called from elsewhere:

```
// VehicleData.h
uint16_t lookupYearByLicense(char* license);
```

Obviously, the sample code shown above does not play a role in my solution. On the other hand, I used this approach to implement the data structures (arrays) that my indexing depends on.

Write lots of "utility" functions because they simplify things tremendously; e.g., string manipulators, comparison functions, command handlers, I/O functions, etc. My solution involves all of those.

Data types, like the structure shown above, play a major role in a good solution. I wrote a number of them, in addition to the ones I used in indexing.

One key to becoming proficient and productive in C, as in most programming languages, is to take full advantage of the library that comes with that language.

Explore string.h carefully (Google and find it at pubs.opengroup.org). Useful functions include strncpy(), strncmp(), memcpy() and strtok(). There are lots of useful functions in the C Standard Library, not just in string.h.

There are some advanced parsing features in C, involving the concept of a scanset used in a format specifier. Consult the notes on this, linked from the Assignments page.

Another potentially useful function in the Standard Library, from stdlib.h, performs Quicksort:

```
void qsort(void *base, size_t nelem, size_t width,
           int (*compare)(const void*, const void*));
```

The interface may seem a bit confusing:

base — a pointer to the array that is to be sorted; it's specified as void* so you can use qsort() on an array holding any kind of data that you like

nelem — the number of elements to be sorted

width — the size (in bytes) of each element in the array; this is usually specified by using sizeof()

int (*compare) (const void *left, const void *right)
the name of a function that takes two void* parameters and returns an integer value; this function is used by qsort() to compare the elements in the array; returning < 0, 0 or >0 depending on whether *left < *right, *left == *right, or *left > *right.

Here's a little example. Recall the Vehicle type defined earlier; suppose we had an array of Vehicle objects, and we wanted to use qsort() to put them in ascending order by the model field. The following comparison function will do:

```
int compareModels(const void *left, const void *right) {

   const Vehicle* pLeft  = (Vehicle*) left;
   const Vehicle* pRight = (Vehicle*) right;

   return ( strcmp( pLeft->model, pRight->model) );
}
```

The call to qsort() would look something like:

```
qsort(vehicleList, nVehicles, sizeof(Vehicle), compareModels);
```

The use of void* in the implementation of the comparison function is typical, old-school C. It would be dangerous, if a call to the comparison function passed pointers to anything other than Vehicle objects, and there's no way to check for that since the use of void* prevents type-checking. On the other hand, since the calls to the comparison function are made by qsort(), as long as we provide qsort() with an array of Vehicle objects, all will be well.

Now, why might sorting be useful? Since each of the index structures uses an array of struct variables to hold information, the only way to achieve log(N) searches is if the arrays are sorted on the struct element that is used as the search key.

One approach would be to build the array, and then sort it. Another would be to keep the array in sorted order as it's built, by placing each new entry into the array so that everything stays sorted. Either approach is acceptable. The second approach would be better if your application required adding new elements on the fly, rather than all at once.

What to Submit

You will submit your solution in a single uncompressed tar file to the Curator, via the collection point C08. That file must include all of the `.c` and `.h` files involved in your solution, and nothing else.

Your submission will be graded by running the supplied test/grading code on it, with several different scripts and data files.

Grading

A number of GIS record files, test scripts and matching logs will be supplied on the course website, as well as the `compare` utility that will be used to evaluate the log files produced by your solution. You will find the following files in the posted testing tarball:

`compare`	comparison tool used in grading; run w/o parameters for instructions
`VA_Highland.txt`	GIS record files, as described in this specification
`VA_Montgomery.txt`	
`script*.txt`	sample test scripts
`reflog*.txt`	corresponding reference log files

Each script file is designed to be used with a specific GIS record file; see the header comments in the scripts. For example:

```
VA_Highland.txt
; Test script 3:  very small db file, testing only the "distance_between" command
;
; Execution:  gis VA_Highland.txt script03.txt log03.txt
;
; Distance searches that should find a single match:
distance_between 1488515      1492871
distance_between 1486995      1496110
distance_between 1498517      1499527
;
; Distance search where first FID doesn't match a record:
distance_between 1486238      1488515
;
; Distance search where second FID doesn't match a record:
distance_between 1488515      1486238
;
; Distance search where neither FID matches a record:
distance_between 1484844      1487459
```

The reference log files are intended to be used with the `compare` utility, so they contain point-value annotations on each line. Here's the beginning of the reference log, `reflog03.txt`, produced from `script03.txt`:

```
[ 0]  VA_Highland.txt
[ 0]  ; Test script 3:  very small db file, testing only the "distance_between" command
[ 0]  ;
[ 0]  ; Execution:  gis script03.txt log03.txt
[ 0]  ;
[ 0]  ; Distance searches that should find a single match:
[ 0]  Cmd   1:  distance_between     1488515      1492871
[ 0]
[10]  First:     ( 079d 36m  1s West,  38d 23m 54s North )  Hannah Field Airport, VA
[10]  Second:    ( 079d 26m 40s West,  38d 25m 55s North )  Doe Hill, VA
[10]  Distance:  17.3km
[ 0]  ----------------------------------------------------------------------
. . .
```

You will not write the point annotations into your log files, but you should match the remaining output more or less exactly. If a line in a log file is annotated with 0 points, it doesn't matter what the content of your matching line is, but there must be a corresponding line in your log (since the comparison tool compares files line by line, and is easily confused).

To use the test script above with your solution, you'd execute the following commands:

```
CentOS> gis VA_Highland.txt script03.txt log03.txt

CentOS> compare 3 reflog03.txt log03.txt
```

Then, examine the output from `compare`, which will be written to the file `compare03.txt`. This will echo lines of your log file, along with score information, and a summary score at the end.

Additional test data files will be added to the posted tar file over the next week or so.

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 `.c` file containing `main()`:

```
//  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 Curator System.
//
//    <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.

Change Log

Version	Posted	Pg	Change
1.00	Oct 25		Base document.
1.01	Oct 31	6	Changed maximum length a GIS file line from 200 to 300.
1.02	Nov 5	2	Clarified the guarantee that the given GIS records will always specify the fields needed for indexing.
		4,5	Moved specification of the name of the database file into the commands file (makes testing simpler).
1.03	Nov 21	10	Corrected example of how to execute program to correspond to the change made on Nov 5.

Orthodromic (great-circle) Distance

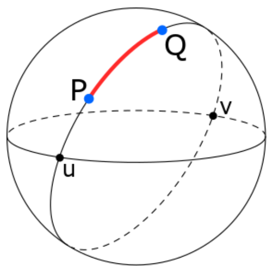


Figure 1

Given a sphere, two points on its surface are *antipodal* if the line connecting the points contains the center of the sphere. u and v are antipodal.

Given a sphere, a *great circle* is a circle whose radius equals that of the sphere and having the same center as the sphere.

Given two points P and Q on the surface of a sphere, if the two points are not at opposite ends of a diameter of the sphere, there is a unique great circle containing P and Q.

The length of the shorter arc between P and Q is the *orthodromic distance* between them; that's shown in red in the diagram. (Of course, if P and Q are antipodal, there are infinitely many great circles containing them, and the distance between them is just half the circumference of the sphere.)

Now, there's a simple fact about the length of an arc of a circle. Given two points on a circle of radius R, A and B, let the angle formed by the radii containing A and B be Θ. This is called the *central angle*.

Then the length of the arc L between A and B is simply $R\Theta$.

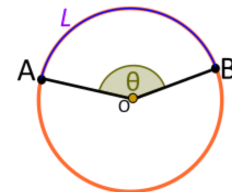


Figure 2

But, how do we calculate the central angle of that arc, if the points are specified by giving their coordinates as longitude and latitude values? The key lies in spherical trigonometry.

First of all, let's pass a plane through the center of the sphere (for our GIS system, this would be the plane determined by the equator). Then, let's choose a line in that plane from the center of the sphere to the surface (for our GIS system, that would simply be the zero meridian).

Given the point P, we have a unique right triangle determined by the radius containing P and a perpendicular from P to the plane of the equator:

The angle lambda between the zero meridian line and the base of that triangle is simply the longitude of P.

The angle ϕ between the base and hypotenuse of the triangle is simply the latitude of P.

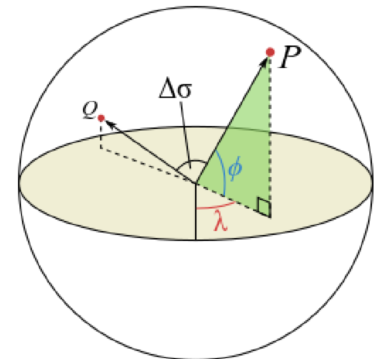


Figure 3

And, we can calculate the central angle, $\Delta\sigma$, by using the *spherical law of cosines*:

$$\Delta\sigma = \arccos\left(\sin\phi_1 \times \sin\phi_2 + \cos\phi_1 \times \cos\phi_2 \times \cos\left(\Delta\lambda\right)\right)$$

where

ϕ_1 is the latitude of P

λ_1 is the longitude of P

ϕ_2 is the latitude of Q

λ_2 is the longitude of Q

$\Delta\lambda = |\lambda_1 - \lambda_2|$

Note that all the angles are expressed in radians (remember those?); you'll have to convert since the GIS data files give longitude and latitude in degrees. And, we'll use the decimal values for longitude and latitude that are given in the GIS data files, because those a slightly more precise than the values you'd get by converting the DMS values.

Standard C does not define a value for π, and you need a value for converting degrees to radians. So you should add the following statement in your code:

```
#define PI      3.14159265358979323846   // best approximation as double
```

Now, the Earth is not a sphere. In fact, the radius through the poles is about 6334.439 km, while the radius at the equator is about 6378.137 km. For the purposes of this assignment, we will trust the IUGG and include the following statement:

```
#define RADIUS 6371.0088                 // IUGG mean radius of Earth, in km
```

Credits

Figure 1, 3:
By CheCheDaWaff - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=48187293

Figure 2:
By Lfahlberg - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/wiki/File:Angle_central_convex.svg