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 (CRLF, `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>
```

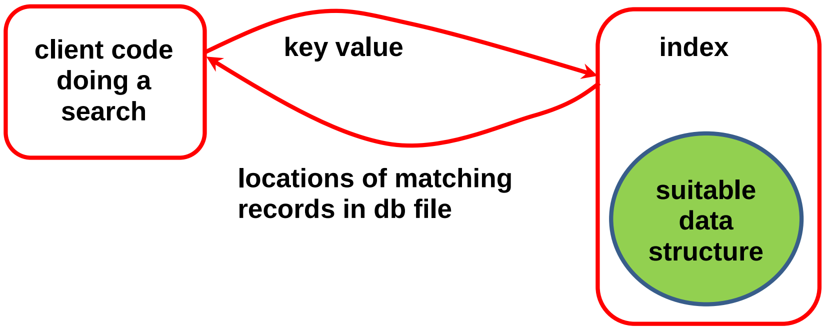
The command-line specifies:

- the name of the file containing the search commands to be carried out, and other information described below
- 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:


client code doing a search
key value
index
locations of matching records in db file
suitable data structure

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.

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.

See the section on **Index Data Structures and Algorithms** for the requirements your implementations of the two index structures must satisfy.

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 retrieved while building the index structures.

Command File

The execution of the program will be driven by a script file. 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 first two lines in the command file that are not comments will specify the name of the GIS record file you will be processing, and the size to use for the hash table (details on that later):

```
db_file<tab><name of GIS record file>
table_sz<tab><size for hash table>
```

The named files will be files that are supplied for testing, and should both be in the same directory as your executable when you run your program. The remaining 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 one or both given feature IDs, 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.

Index Data Structures and Algorithms

Feature ID Index

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

The FID index must support searches that are O(log N), where N is the number of records in the database file; that means you need to use binary search within the FID index, and so the entries there must be sorted by FID. There are two ways to achieve this:

- keep the entries in sorted order as you add them to the array
- put all the entries into the array and then sort them

The first approach is somewhat less efficient, but can be done rather easily by shifting entries in the array when a new entry is added. The second approach requires using some sensible sorting algorithm.

The Standard Library, from `stdlib.h`, supplies a function that performs Quicksort on an array:

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

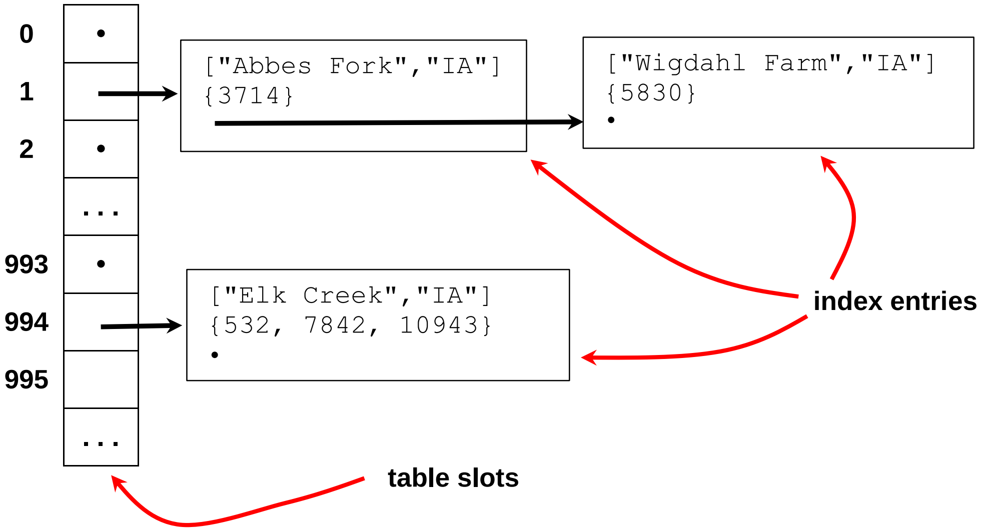
There's a relevant example of how to use this in the **Advice** section later in this specification.

When implementing the FID index, make the initial array of size 256. If the array becomes full as you build the index, resize it by doubling the size. This may be accomplished by using `realloc()`. You may have to resize the array more than once.

Feature Name Index

You will implement the feature name index as a *hash table*, using an array of pointers to `struct` variables as the physical structure. The `struct` variables in the array will store three pieces of information: a string that you construct from the feature name and the state abbreviation for a feature, an array of file offsets of matching records, and a pointer.

Here's an example of the sort of table you'll be implementing:



Dots represent `NULL` pointers, and the notation used for the strings just indicates that the two parts are combined into a single string in some way that you can choose. You are free to combine the feature name and state abbreviation in any way you like.

Elements are mapped into a hash table by making use of a "magic function" that takes a *key value* as input and produces a nonnegative integer value, which is called the *hash value* of the key. The function is called a *hash function*. For this assignment, the key values will be the feature name/state abbreviation pairs.

In the example above, let's assume the array has dimension 1000; in other words, we'd say the hash table has 1000 *slots*. The hash function has taken the key `["Wigdahl Farm","IA"]` and has produced an integer **K** (which we do not know), such that **K** % 1000 == 1.

That is, the hash function supplies a nonnegative integer, computed in some fashion from the key for some record, and that integer is modded by the table size to produce a valid table index at which the record will be stored. We call that slot the *home slot* for the key.

Ideally, the hash function would never map two different keys to the same table slot. Do not count on achieving that except in special circumstances. If the hash function does map two different keys to the same table slot, we say the keys have *collided* in that slot. For this assignment, we will deal with collisions by simply treating each table slot as a linked list of *index entries* (`struct` variables), where each index entry corresponds to a different key value.

Searching in a hash table proceeds by using the hash function to compute a table index from the key we are interested in, and then searching the list in that table slot to see if there's an entry for the given key value. In the ideal case, searches in the hash table are O(1), which is why you are using one.

The assignment also involves an annoyance: feature names are not necessarily unique, even within the same state. For example, the GIS database mentioned earlier lists three different places in Virginia named Blacksburg! Since we never want to put two different entries (`struct` variables) that store the same key in a hash table, each entry in the table needs to store the file offset for every record that matches the given key. That's illustrated in the table above with the key field `["Elk Creek","IA"]`, which has three matches.

There are lots of other, interesting things to say about the general concept of a hash table, but we will leave that for a course on data structures.

The question remains: what sort of function could we use to compute an integer from a string? There are actually lots of ways to do this. You will use the following hash function, which is based on one that was implemented as part of the first UNIX implementation at Bell Labs:

```
uint32_t elfhash(const char* str) {

   assert(str != NULL );      // self-destuct if called with NULL

   uint32_t hashvalue = 0,    // value to be returned
            high;             // high nybble of current hashvalue

   while ( *str ) {           // continue until *str is '\0'

      hashvalue = (hashvalue << 4) + *str++;     // shift high nybble out,
                                                 //   add in next char,
                                                 //   skip to the next char

      if ( high = (hashvalue & 0xF0000000) ) {   // if high nybble != 0000
         hashvalue = hashvalue ^ (high >> 24);   //   fold it back in
      }

      hashvalue = hashvalue & 0x7FFFFFFF;        // zero high nybble
   }

   return hashvalue;
}
```

All of the hash table requirements are the same as in the previous hash table assignment, so you can just use your solution to that here (with a suitable interface, as discussed earlier). However, we will also provide an implementation of that hash table, and you may also use that if you wish. See the discussion of the posted code later in this specification.

Permitted Assumptions and Some Advice

You may assume that:

- no GIS record will ever be longer than 500 characters
- no line in the command file will ever be longer than 500 characters
- the GIS record file and the command file will follow the specified syntax described earlier

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. Implement one index first, and test searches using that index. Then work on the other index.

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**.

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 in the first appendix, 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.

What to Submit

You will submit your solution in a single, flat, 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 tar file with a complete set of test data (scripts and GIS files), and code to automate full grading, is posted.

```
README                      SO READ IT!!
dev/                        empty directory; develop your solution here
grading/                    full grading harness
   gradeC08.sh              bash script to automate grading procedure
   compare                  utility used by the grading script
   gisdata/                 directory holding GIS data files used in grading
   scripts/                 directory holding command files used in grading
   logs/                    directory holding reference output logs used in grading
suppliedcode/               directory holding supplied hash table implementation
   StringHashTable.h           StringHashTable header file
   StringHashTable.o           64-bit implementation of the StringHashTable
```

See the README file for details.

Your solution is expected to produce a completely clean run on Valgrind: no memory leaks, no out of bounds accesses, no uses of uninitialized variables. Failure to do so will result in a penalty of up to 10%.

Note that this includes the same test data we will use to grade your solution, using the supplied grading script. It is essential that you test your submission by running the grading script, as described in the README. Failure to do so may result in a score of zero.

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
3.00	Oct 31		Base document.
3.10	Nov 13	9	Added Valgrind requirement to grading description.
3.15	Dec 1	5	Fixed description of output for the details_of command.

Appendix Enumerated Types, qsort(), and Static Table Lookup

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;
...
```

Enumerated types let me use descriptive labels in my code, which makes it easier to understand the logic of the code.

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.

A 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. Suppose we have defined the Vehicle type shown below, and 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 interface uses void* parameters because the qsort() interface requires that. On one hand, this is a way to achieve generic programming in C. On the other hand, this makes compile-time type checking impossible.

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.

Appendix: Using Valgrind

Valgrind is a tool for detecting certain memory-related errors, including out of bounds accessed to dynamically-allocated arrays and memory leaks (failure to deallocate memory that was allocated dynamically). A short introduction to Valgrind is posted on the Resources page, and an extensive manual is available at the Valgrind project site (www.valgrind.org).

For best results, you should compile your C program with a debugging switch (-g or -ggdb3); this allows Valgrind to provide more precise information about the sources of errors it detects. I ran my solution for an earlier assignment on Valgrind:

```
#1051 wmcquain: soln> valgrind --leak-check=full --show-leak-kinds=all --log-file=vlog.txt --track-
origins=yes -v c07driver NM_1000.txt
```

And, I got good news... there were no detected memory-related issues with my code:

```
==19625== Memcheck, a memory error detector
==19625== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==19625== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==19625== Command: c07driver NM_1000.txt
==19625== Parent PID: 3530
==19625==
--19625--
--19625-- Valgrind options:
--19625--    --leak-check=full
--19625--    --show-leak-kinds=all
--19625--    --log-file=vlog.txt
--19625--    --track-origins=yes
--19625--    -v. . .
==19625==
==19625== HEAP SUMMARY:
==19625==     in use at exit: 0 bytes in 0 blocks
==19625==   total heap usage: 10,047 allocs, 10,047 frees, 992,594 bytes allocated
==19625==
==19625== All heap blocks were freed -- no leaks are possible
==19625==
==19625== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==19625== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

That's the sort of report you want to see when you try your solution with Valgrind.

On the other hand, here’s (part of) what I got from a substantially incomplete solution to the same assignment:

```
. . .
==7273== Memcheck, a memory error detector
==7273== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7273== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7273== Command: ./c07_wmcquain NM_1000.txt
==7273== Parent PID: 7272
==7273==
--7273--
--7273-- Valgrind options:
--7273--    --leak-check=full
--7273--    --show-leak-kinds=all
--7273--    --log-file=vgrindLog.txt
--7273--    --track-origins=yes
--7273--    -v
. . .
==7273== Invalid read of size 1
==7273==    at 0x4E82F19: vfprintf (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4E89338: printf (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4034B0: getNumRecords (in /home/...c07_wmcquain)
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==  Address 0x5203c94 is 0 bytes after a block of size 4 alloc'd
==7273==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==    by 0x403568: readLine (in /home/...c07_wmcquain)
==7273==    by 0x403496: getNumRecords (in /home/...c07_wmcquain)
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
```

```
==7273==
==7273== Invalid read of size 1
==7273==    at 0x4E71024: ____strtol_l_internal (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4E6D7EF: atoi (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4034BC: getNumRecords (in /home/...c07_wmcquain)
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==  Address 0x5203c94 is 0 bytes after a block of size 4 alloc'd
==7273==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==    by 0x403568: readLine (in /home/...c07_wmcquain)
==7273==    by 0x403496: getNumRecords (in /home/...c07_wmcquain)
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
. . .
==7273==
==7273== HEAP SUMMARY:
==7273==     in use at exit: 218,895 bytes in 5,206 blocks
==7273==   total heap usage: 10,047 allocs, 4,841 frees, 991,559 bytes allocated
==7273==
==7273== Searching for pointers to 5,206 not-freed blocks
==7273== Checked 70,272 bytes
==7273==
==7273== 24 bytes in 1 blocks are definitely lost in loss record 1 of 13
==7273==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==7273==    by 0x402E05: refCreateTable (in /home/...c07_wmcquain)
==7273==    by 0x401A09: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
==7273== 32 bytes in 1 blocks are indirectly lost in loss record 2 of 13
==7273==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==    by 0x401148: addOffset (StringHashTable.c:47)
==7273==    by 0x40140F: addEntry (StringHashTable.c:103)
==7273==    by 0x401B2F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
. . .
==7273==
==7273== 15,056 bytes in 941 blocks are indirectly lost in loss record 8 of 13
==7273==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==7273==    by 0x4010B0: createStringNode (StringHashTable.c:34)
==7273==    by 0x401433: addEntry (StringHashTable.c:110)
==7273==    by 0x401B2F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
. . .
==7273== 64,463 (24 direct, 64,439 indirect) bytes in 1 blocks are definitely lost in loss record
12 of 13
==7273==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==7273==    by 0x4012D9: createTable (StringHashTable.c:78)
==7273==    by 0x4019F1: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
==7273== 134,841 bytes in 999 blocks are definitely lost in loss record 13 of 13
==7273==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==    by 0x403568: readLine (in /home/...c07_wmcquain)
==7273==    by 0x401A9E: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
==7273== LEAK SUMMARY:
==7273==    definitely lost: 154,324 bytes in 2,378 blocks
==7273==    indirectly lost: 64,439 bytes in 2,827 blocks
==7273==      possibly lost: 132 bytes in 1 blocks
==7273==    still reachable: 0 bytes in 0 blocks
==7273==         suppressed: 0 bytes in 0 blocks
==7273==
==7273== ERROR SUMMARY: 2010 errors from 11 contexts (suppressed: 0 from 0)
==7273==
==7273== 1 errors in context 1 of 11:
==7273== Invalid read of size 1
==7273==    at 0x4E71024: ____strtol_l_internal (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4E6D7EF: atoi (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4034BC: getNumRecords (in /home/...c07_wmcquain)
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
```

```
==7273==   by 0x400D4B: main (c07driver.c:55)
==7273==  Address 0x5203c94 is 0 bytes after a block of size 4 alloc'd
==7273==   at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==   by 0x403568: readLine (in /home/...c07_wmcquain)
==7273==   by 0x403496: getNumRecords (in /home/...c07_wmcquain)
==7273==   by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
==7273==   by 0x400D4B: main (c07driver.c:55)
==7273==
. . .
==7273== 2000 errors in context 3 of 11:
==7273== Invalid read of size 1
==7273==   at 0x4C30834: __GI___rawmemchr (vg_replace_strmem.c:1410)
==7273==   by 0x4EB4A41: _IO_str_init_static_internal (in /usr/lib64/libc-2.17.so)
==7273==   by 0x4EA2566: __isoc99_vsscanf (in /usr/lib64/libc-2.17.so)
==7273==   by 0x4EA2506: __isoc99_sscanf (in /usr/lib64/libc-2.17.so)
==7273==   by 0x4035DE: nextField (in /home/...c07_wmcquain)
==7273==   by 0x401AC8: testBuildingTable (in /home/...c07_wmcquain)
==7273==   by 0x400D4B: main (c07driver.c:55)
==7273==  Address 0x52063b6 is 0 bytes after a block of size 134 alloc'd
==7273==   at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==   by 0x403568: readLine (in /home/...c07_wmcquain)
==7273==   by 0x401A9E: testBuildingTable (in /home/...c07_wmcquain)
==7273==   by 0x400D4B: main (c07driver.c:55)
==7273==
==7273== ERROR SUMMARY: 2010 errors from 11 contexts (suppressed: 0 from 0). . .
```

As you see, Valgrind can also detect out-of-bounds accesses to arrays. In addition, Valgrind can detect uses of uninitialized values; a Valgrind analysis of your solution should not show any of those either. So, try it out if you are having problems.

Interpreting Valgrind Error Messages

Here's one of the error messages from the Valgrind report above, reporting a memory access error:

```
==7273== Invalid read of size 1                                              1
==7273==   at 0x4E82F19: vfprintf (in /usr/lib64/libc-2.17.so)               2
==7273==   by 0x4E89338: printf (in /usr/lib64/libc-2.17.so)                 3
==7273==   by 0x4034B0: getNumRecords (in /home/...c07_wmcquain)             4
==7273==   by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)         5
==7273==   by 0x400D4B: main (c07driver.c:55)                                6
==7273==  Address 0x5203c94 is 0 bytes after a block of size 4 alloc'd       7
==7273==   at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)                   8
==7273==   by 0x403568: readLine (in /home/...c07_wmcquain)                  9
==7273==   by 0x403496: getNumRecords (in /home/...c07_wmcquain)            10
==7273==   by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)        11
==7273==   by 0x400D4B: main (c07driver.c:55)                               12
```

Here's what it tells me:

1 a one-byte value (most likely a **`char`**, **`int8_t`** or **`uint8_t`**) is being written at an invalid location
2 this occurred in `printf()`, which is not likely to be the culprit
3-5 `printf()` was called from `getNumRecords()`, which was called by `testBuildingTable()`

I suspect the issue is in `getNumRecords()` or `testBuildingTable()`; I could be wrong…

7 the invalid write was 0 bytes after an allocation of size 4, so it was immediately after an allocation; perhaps the allocation was too small
8-12 the allocation was created by a call to `realloc()` from `readLine()`, which was called from `getNumRecords()`

So, I need to look at `readLine()`, `getNumRecords()`, and possibly `testBuildingTable()`. BTW, the Valgrind report would have included line numbers in those functions, if they had been compiled with `-ggdb3`.

Now, let's consider a memory leak message from the same Valgrind report:

```
==7273== 15,056 bytes in 941 blocks are indirectly lost in loss record 8 of 13      1
==7273==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)                          2
==7273==    by 0x4010B0: createStringNode (StringHashTable.c:34)                    3
==7273==    by 0x401433: addEntry (StringHashTable.c:110)                           4
==7273==    by 0x401B2F: testBuildingTable (in /home/...c07_wmcquain)               5
==7273==    by 0x400D4B: main (c07driver.c:55)                                      6
```

Here's what it tells me:

1 there were 941 blocks, totaling 15056 bytes, that were not freed
2 the blocks were created by a call to `calloc()` from line 34 inside the `createStringNode()` function
3 `createStringNode()` was called by the `addEntry()` function

This code seems to be responsible for adding **`StringNode`** objects to the **`StringHashTable`**. Those **`StringNode`** objects should have been deallocated by my `clearTable()` function. So, perhaps there is an error in `clearTable()`, or perhaps `clearTable()` is not being called when it should be.

Then again, the report implies that each leak may be 16 bytes in size (do the math)… so this is not about leaking full **`StringNode`** objects. Checking line 34 in my `StringHashTable.c` file, I see that this is where I allocate the array to store the record locations, and the specification calls for that array to (initially) be sized to 4 values of type **`uint32_t`**, which amounts to 16 bytes. So, I'm probably forgetting to deallocate those arrays…

Appendix: Function Pointers

In C, a function name is a pointer; we take advantage of that in this assignment. Recall that we have the following hash table type:

```
struct _StringHashTable {

   StringNode** table;        // physical array for the table
   uint32_t tableSz;          // number of slots in the table
   uint32_t numEntries;       // number of entries in the table (not
                               //   necessarily the number of nonempty slots)

   uint32_t (*hash)(const char* str);    // pointer to the hash function
};
typedef struct _StringHashTable StringHashTable;
```

Suppose you've created such a hash table object. How would you call the hash function from within your implementation? Basically just like any other function call. For example, you could use code like this to call the hash function when adding an entry to your hash table:

```
bool StringHashTable_addEntry(StringHashTable* const pTable,
                              char* key, uint32_t location) {

   uint32_t hashValue = pTable->hash(key);
   . . .
```

The parameter `pTable` is a pointer to a `StringHashTable` object, which has a field called `hash`, which happens to be a pointer to the hash function you want to use.

So, you have to dereference the pointer, `pTable`, to access the pointer hash, then just pass in the pointer to the key you want to hash… simple.

Appendix

Orthodromic (great-circle) Distance

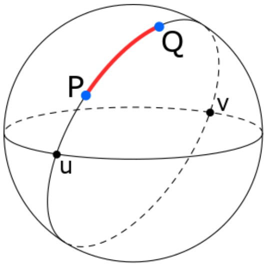


Figure 1

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*Θ .

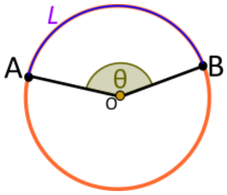


Figure 2

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*.

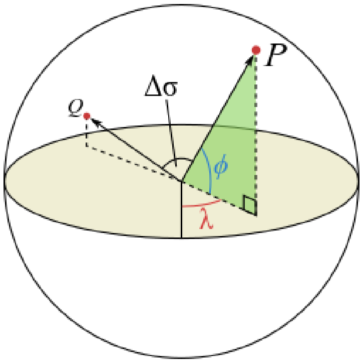


Figure 3

And, we can calculate the central angle, Δσ, 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