C Programming File Accesses and Parsing Formatted C Strings

This assignment will explore performing some basic operations on a text file. A file is simply a sequence of bytes, stored on some device. Each byte in the file occurs at a unique position (*offset*), relative to the beginning of the file. For example, here are the first few bytes of a simple text file:

```
Designing score harvester
-------------------------------------------------------------------------
Let's agree on how to write the scores into the log files we post for
the students first.

There will be a number of descriptive header lines, and that has to
be allowed to vary.  Therefore, we will always precede the block of
score data with a fixed header line:

       >>Scores from testing<<
. . .
```

On a storage device, like a disk, the file would be stored as a sequence of ASCII codes for the characters:

'D'	'e'	's'	'i'	'g'	'n'	'i'	'n'	'g'	' '
0	1	2	3	4	5	6	7	8	9

So:

`'D'` is at offset 0
`'e'` is at offset 1
`'s'` is at offset 2
...

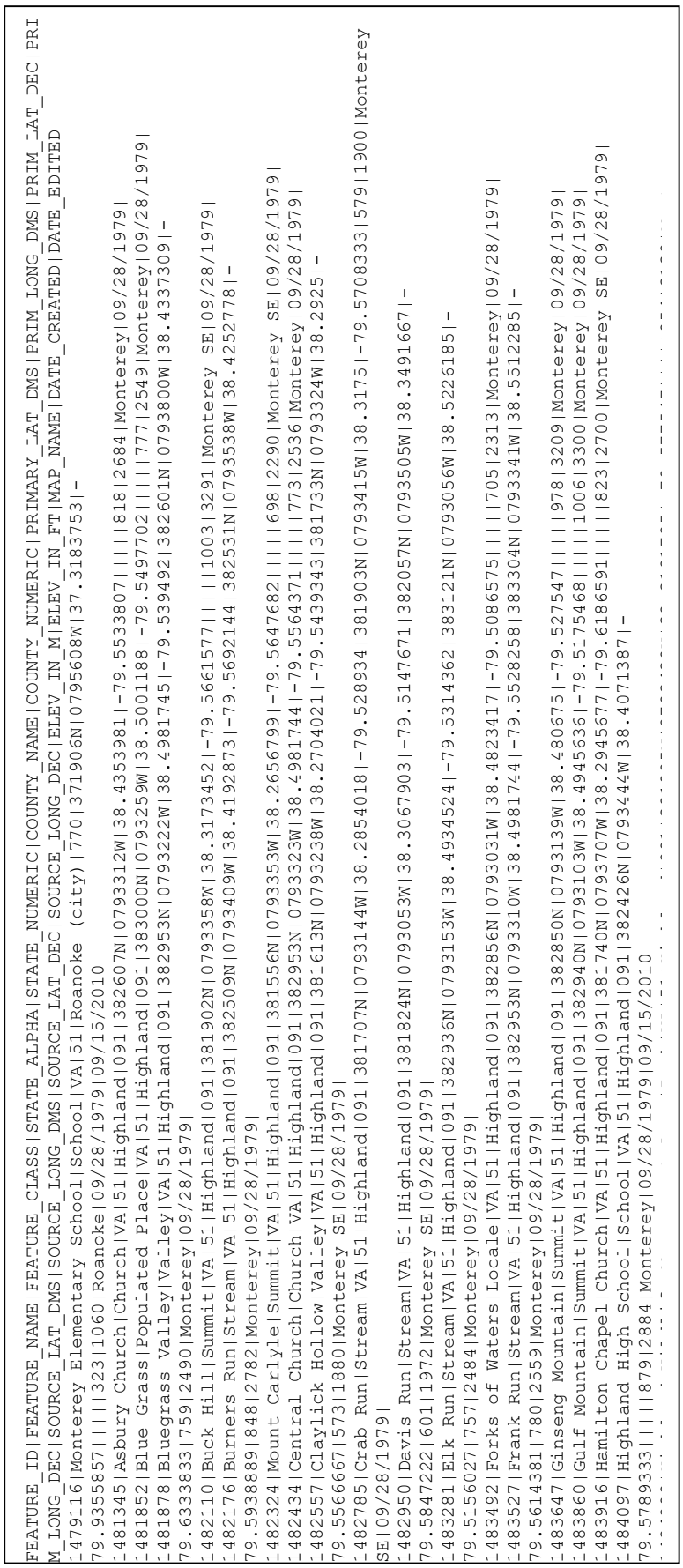
Note that, this is essentially the same as array indexing. For any file, offsets range from 0 to some maximum value (depending on the size of the file). Given any valid offset for a file, there is a unique byte at that offset in the file. Also note that some characters are "invisible". For example, in the file above, there is a line terminator immediately after the character `'r'` in the first line. For a Linux-format[1] file that terminator is a single byte (a *line feed*), and would occur at offset 26 in the given file. For a Windows-format file, that terminator is actually a pair of bytes (a *carriage return* followed by a *line feed*), occurring at offsets 26 and 27.

We will explore how, using the C Standard Library, to:

- Determine the offset of a byte within a file.
- Retrieve a line of data from a file, starting from a given offset within the file.
- Parse a string retrieved from a file, where the string is divided into fields delimited by special characters.

Each of these activities will play a major role in a later, larger project in the course.

The data files used for this are obtained from the USGS Board on Geographic Names[2]. An example file is shown on the following page.



```
FEATURE_ID|FEATURE_NAME|FEATURE_CLASS|STATE_ALPHA|STATE_NUMERIC|COUNTY_NAME|COUNTY_NUMERIC|PRIMARY_LAT_DMS|PRIM_LONG_DMS|PRIM_LAT_DEC|PRI
M_LONG_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
```

Figure 1: Sample GIS Record File

The first line contains field labels.

Each subsequent line is a GIS record, listing attributes of some geographic feature.

The attributes (fields) are separated by pipe characters (' | ').

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

Phase 1: Determining File Offsets of GIS Records

You will implement the following function in C, and place the implementation in a file named `offsetFinder.c`:

```
/**  Reads a GIS record file (as described in the corresponding project
 *   specification), and determines, for each GIS record contained in that
 *   file, the offset at which that record begins.  The offsets are stored
 *   into an array supplied by the caller.
 *
 *   Pre:  gisFile is open on a GIS record file
 *         offsets[] is an array large enough to hold the offsets
 *   Post: offsets[] contains the GIS record offsets, in the order
 *            the records occur in the file
 *   Returns:  the number of offsets that were stored in offsets[]
 */
uint32_t findOffsets(FILE* gisFile, uint32_t offsets[]);
```

Assumptions you may make in designing your solution:

- The GIS record file will correspond exactly to the description given earlier, but it will be a different file than the one shown earlier in this specification.
- No line in the GIS record file will be longer than 500 characters, including the line terminator.
- The caller will ensure that all the stated preconditions are true.

You may not make any assumptions about the number of records that will be in the GIS record file; your solution must be designed to read until it reaches an input failure at the end of the given file.

Some useful functions (`stdio.h`)[3]:

```
char* fgets(char* s, int n, FILE* stream);
```

DESCRIPTION
The fgets() function shall read bytes from stream into the array pointed to by s, until n-1 bytes are read, or a <newline> is read and transferred to s, or an end-of-file condition is encountered. The string is then terminated with a null byte.

RETURN VALUE
Upon successful completion, fgets() shall return s. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and fgets() shall return a null pointer. If a read error occurs, the error indicator for the stream shall be set, fgets() shall return a null pointer.

```
long ftell(FILE *stream);
```

DESCRIPTION
The ftell() function shall obtain the current value of the file-position indicator for the stream pointed to by stream.

RETURN VALUE
Upon successful completion, ftell() shall return the current value of the file-position indicator for the stream measured in bytes from the beginning of the file.

Phase 2: Retrieving a GIS Record Located at a Given File Offset

You will implement the following function in C, and place the implementation in a file named `recordRetriever.c`:

```
/**  Returns the GIS record at the given offset.
 *
 *   Pre:  gisFile is open on a GIS record file
 *         offset is a valid offset for the given GIS record file
 *   Returns:  a valid C-string holding the specified GIS record
 */
char* retrieveRecord(FILE* gisFile, uint32_t offset);
```

Assumptions you may make in designing your solution:

- The GIS record file will correspond exactly to the description given earlier, but it will be a different file than the one shown earlier in this specification.
- No line in the GIS record file will be longer than 500 characters, including the line terminator.
- The caller will ensure that all the stated preconditions are true.

Some useful functions (`stdio.h` and `stdlib.h`)[3]:

```
int fseek(FILE* stream, long offset, int whence);

    DESCRIPTION
    The fseek() function shall set the file-position indicator for the stream pointed to
    by stream. If a read or write error occurs, the error indicator for the stream shall
    be set and fseek() fails.

    The new position, measured in bytes from the beginning of the file, shall be obtained
    by adding offset to the position specified by whence. The specified point is the
    beginning of the file for SEEK_SET, the current value of the file-position indicator
    for SEEK_CUR, or end-of-file for SEEK_END.

    RETURN VALUE
    The fseek() function shall return 0 if it succeeds.

void* calloc(size_t nelem, size_t elsize);

    DESCRIPTION
    The calloc() function shall allocate unused space for an array of nelem elements each
    of whose size in bytes is elsize. The space shall be initialized to all bits 0.

    RETURN VALUE
    Upon successful completion with both nelem and elsize non-zero, calloc() shall return
    a pointer to the allocated space. If either nelem or elsize is 0, then either a null
    pointer or a unique pointer value that can be successfully passed to free() shall be
    returned. Otherwise, it shall return a null pointer.
```

In this case, I recommend using `calloc()` in preference to `malloc()`. Because I'm allocating space for a C-string, the fact that `calloc()` zeros out the allocation means that there will automatically be a terminator on the string (as long as I allow room for it).

Phase 3: Parsing a GIS Record into Fields

Finally, we will consider the problem of decomposing a given string, that is divided into logical parts (fields) which are separated by known delimiters, into a collection of separate strings. In particular, consider the strings shown below, which are GIS records taken from a GIS database file:

```
901051|Becker|Locale|NM|35|Eddy|015|322833N|1040812W|32.4759521|-104.1366141|||||959|3146|Carlsbad East|11/01/1992|

902674|Twin Boils Spring|Spring|NM|35|Eddy|015|323333N|1042326W|32.559118|-104.3906084|||||982|3222|Seven
Rivers|01/01/1993|04/19/2011
```

The line break in the second record is just a result of wrapping a line that's too long to fit the page width. Each record consists of 20 fields, separated by pipe symbols (`'|'`). Some fields are empty. An empty field occurs if there are two adjacent pipe symbols; there are many examples of that in both records. An empty field occurs at the end, if the last character in the record is a pipe symbol; that happens with the first record.

Given the first record above, we want to break it up into a collection of 20 strings, some of which will be empty:

0	901051
1	Becker
2	Locale
3	NM
4	35
5	Eddy
6	015
7	322833N
8	1040812W
9	32.4759521
10	-104.1366141
11	
12	
13	
14	
15	959
16	3146
17	Carlsbad East
18	11/01/1992
19	

Some fields could be interpreted as numbers, or dates, or coordinates, but we'll just think of each of them as a string of characters. Some fields are always the same width, while other fields can vary in width. We won't try to make use of the width when deciding how to break the string into separate fields.

We will assume that each record string will conform to the format shown above. That is, each will contain 19 pipe symbols, separating 20 fields, where some fields may be empty. A few fields are, in fact, guaranteed to be nonempty, but many fields may be empty or not. We will not make any assumptions about which fields may be empty, since doing so doesn't make our task any simpler.

Now, C represents strings as **`char`** arrays with a terminating zero byte (`'\0'`). For each field, we will dynamically allocate a **`char`** array of exactly the right length; so an empty string will be represented by an array of dimension 1, holding a zero byte.

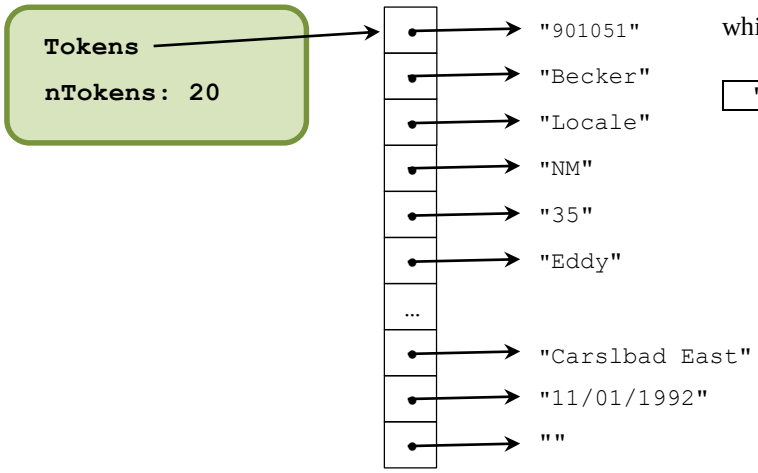
Since we will have multiple fields, and we must use **`char`** pointers when we allocate the arrays, we can use an array of **`char*`** to organize the set of fields we parse out of a record string. It is tempting to just use an array of 20 **`char*`** for this, but we should strive for a more flexible solution.

We will use a **`struct`** type that contains a dynamically allocated array of **`char*`**, and also stores the dimension of that array:

```
/**  A StringBundle contains an array of nTokens pointers to properly-
 *   terminated C strings (char arrays).
 *
 *   A StringBundle is said to be proper iff:
 *     - Tokens == NULL and nTokens == 0
 *   or
 *     - nTokens > 0 and Tokens points to an array of nTokens char pointers,
 *     - each char pointer points to a char array of minimum size to hold
 *       its string, including the terminator (no wasted space)
 */
struct _StringBundle {
   char**   Tokens;    // pointer to dynamically-allocated array of char*
   uint32_t nTokens;   // dimension of array pointed to by Tokens
};
typedef struct _StringBundle StringBundle;
```

The field `Tokens` is a **`char`**`**` because it points to the first element in an array of **`char`**`*` variables, so `Tokens` is a pointer to a pointer to something of type **`char`**.

A `StringBundle` object that results from parsing the first GIS record string shown above would look like this:



which is really an array:

'9'	'0'	'1'	'0'	'5'	'1'	'\0'

You will implement the following function in C, and place the implementation in a file named `StringBundle.c`:

```
/**  Parses *str and creates a new StringBundle object containing the
 *   separate fields of *str.
 *
 *   Pre:     str points to a GIS record string, properly terminated
 *
 *   Returns: a pointer to a new proper StringBundle object
 */
StringBundle* createStringBundle(const char* const str);
```

Your solution must not create any memory leaks. Of course, when your function allocates memory dynamically, and that memory is logically part of the `StringBundle` object being returned to the caller, deallocations of that are the responsibility of the caller.

To facilitate that, you will also be implementing the following function in C, and place the implementation in a file named `StringBundle.c` (along with the previous function):

```
/**  Frees all the dynamic memory content of a StringBundle object.
 *   The StringBundle object that sb points to is NOT deallocated here,
 *   because we don't know whether that object was allocated dynamically.
 *
 *   Pre:     *sb is a proper StringBundle object
 *
 *   Post:    all the dynamic memory involved in *sb has been freed;
 *            *sb is proper
 */
void clearStringBundle(StringBundle* sb);
```

The test code will call this function whenever it has a `StringBundle` object that is no longer needed. The test code will also deallocate any memory that it allocates dynamically, so if Valgrind indicates there are any memory leaks, the fault will lie in one of your functions.

Your solution may use any of the functions declared in the C Standard Library, including the string manipulation functions. In addition to the functions mentioned earlier in this specification, the following functions may be very useful:

```
void* realloc(void* ptr, size_t size);
```

DESCRIPTION
The realloc() function shall change the size of the memory object pointed to by ptr to the size specified by size. The contents of the object shall remain unchanged up to the lesser of the new and old sizes. If the new size of the memory object would require movement of the object, the space for the previous instantiation of the object is freed. If the new size is larger, the contents of the newly allocated portion of the object are unspecified. If size is 0 and ptr is not a null pointer, the object pointed to is freed. If the space cannot be allocated, the object shall remain unchanged.

If ptr is a null pointer, realloc() shall be equivalent to malloc() for the specified size.

If ptr does not match a pointer returned earlier by calloc(), malloc(), or realloc() or if the space has previously been deallocated by a call to free() or realloc(), the behavior is undefined.

RETURN VALUE
Upon successful completion with a size not equal to 0, realloc() shall return a pointer to the (possibly moved) allocated space. If size is 0, either a null pointer or a unique pointer that can be successfully passed to free() shall be returned. If there is not enough available memory, realloc() shall return a null pointer.

```
void free(void* ptr);
```

DESCRIPTION
The free() function shall cause the space pointed to by ptr to be deallocated; that is, made available for further allocation. If ptr is a null pointer, no action shall occur. Otherwise, if the argument does not match a pointer earlier returned by the calloc(), malloc(), realloc(), or if the space has been deallocated by a call to free() or realloc(), the behavior is undefined.

Any use of a pointer that refers to freed space results in undefined behavior.

```
void* memcpy(void* s1, const void* s2, size_t n);
```

DESCRIPTION
The memcpy() function shall copy n bytes from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined.

```
char* strncpy(char* s1, const char* s2, size_t n);
```

DESCRIPTION
The strncpy() function copies not more than n bytes (bytes that follow a null byte are not copied) from the array pointed to by s2 to the array pointed to by s1. If copying takes place between objects that overlap, the behaviour is undefined.
If the array pointed to by s2 is a string that is shorter than n bytes, null bytes are appended to the copy in the array pointed to by s1, until n bytes in all are written.

RETURN VALUE
The strncpy() function returns s1; no return value is reserved to indicate an error.

```
size_t strlen(const char* s);
```

DESCRIPTION
The strlen() function shall compute the number of bytes in the string to which s points, not including the terminating null byte.

RETURN VALUE
The strlen() function shall return the length of s; no return value shall be reserved to indicate an error.

```
int sscanf(const char *s, const char *format, ...);
```

DESCRIPTION
For this function, see the course notes on String I/O. I found the notion of a *scanset* to be particularly useful.

You may, and are encouraged to, write additional helper functions. Any such functions must be declared as **`static`**, in the file `StringBundle.c`, making them private to the C source file you will turn in.

For what it's worth, my solution includes at least two such helper functions. One is a variation on `strcpy()` and the other is a variation on `strtok()`. Each plays a vital role in my design, and each was motivated by the fact that those two C Standard Library functions were not quite what I needed. These functions are described in more detail in the appendix Some Implementation Suggestions.

Supplied Implementation Code

Download the supplied tar file, `c04Implementation.tar`, and unpack it in a CentOS directory. You will find the following files:

```
c04driver.c             test driver; read the comments!
offsetFinder.h*         header file for findOffset() implementation
offsetFinder.c          C shell file for findOffset() implementation
recordRetriever.h*      header file for retrieveRecord() implementation
recordRetriever.c       C shell file for retrieveRecord() implementation
StringBundle.h*         header file for StringBundle functions
StringBundle.c          C shell file for StringBundle functions
dataSelector.h*         header file for test case generator
dataSelector.o*         64-bit Linux binary for test case generator
checkStringBundle.h*    header file for grading function
checkStringBundle.o*    64-bit Linux binary for grading function
runValgrind.sh          a bash script to simplify your use of Valgrind;
                          see the comments for how to run it
GISdata.txt*            a file of GIS records; only used by dataSelector
```

Do not modify the files marked with an asterisk (`*`), because you will not be submitting those files. You may modify the driver file during your testing, but we will use the original version when grading. Compile the code with the command:

```
centos > gcc -o c04 -std=c11 -Wall -W -ggdb3 c04driver.c offsetFinder.c recordRetriever.c
StringBundle.c dataSelector.o checkStringBundle.o
```

The supplied C files include trivial implementations for the required functions, so that the code will compile even if you have not completed all of them. That should not result in runtime crashes during testing, but those implementations will not pass testing.

Invoke the driver as:

```
centos > ./c04 <name of GIS data file> <name for results file> [-repeat]
```

If invoked without the switch `-repeat`, the `dataSelector` will choose a random set of GIS record strings from the specified GIS data file, and use those strings for testing. When invoked with `-repeat`, the most recently set of GIS records will be used again. You should specify the supplied GIS record file.

The results file will show the results of the tests that were performed, along with some useful feedback for certain errors, and score information.

If your solution creates bad pointers, or improperly-terminated C strings, it is possible the testing code will be crashed by a `segfault` error. If that happens, a backtrace in `gdb` may help pin down the error.

Valgrind is likely to be even more helpful with that kind of error, because if there are access errors with memory allocations, Valgrind will show where those allocations were requested, and where (in code) the access errors occurred. See the appendix Using Valgrind for more information.

Supplied Grading Code

Download the supplied tar file, `c04Grading.tar`, and unpack it in a different CentOS directory than the one you used for the implementation. We'll call this the test directory. You will find the following files:

```
gradeC04.sh             bash shell script for grading; read the comments!
c04Files.tar            tar file containing code to compile with your solution
```

Prepare the tar file you intend to submit (see **What to Submit** below). I suggest you name the tar file using your PID; in my case I'll call mine `wmcquain.tar`. Copy your tar file into the directory containing the two files listed above, and run the grading script:

```
centos > ./gradeC04 <name of your tar file>
```

The script will:

- Verify that your file exists and that it's a valid tar file.
- Verify that `c04Files.tar` is present.
- Create a directory, `./build`, where compilation will be performed.
- Create a directory, `./unpack`, and unpack your tar file into it; verify that your tar file contained the three mandatory `.c` files, and copy those files into `./build`.
- Unpack `c04Files.tar` into `./build`, and compile the files there.
- Copy the executable into the test directory, and execute it to perform testing of all three phases.
- Execute the code again, on Valgrind, and gather the Valgrind output.

If any errors occur, with the files, or with compilation, suitable error messages will be written, either to the terminal window or to a log file.

Look for a file named `PID.txt` (`wmcquain.txt` in my case). That will contain grading details for your submission. In my case, the log file starts with the something like the following lines:

```
Grading:  wmcquain.tar
Time:     Wed Jan 6 22:48:48 EST 2021

======================================================================

>>Scores from testing<<
   1 >> score for offset count:  20 / 20
   2 >> score for found offsets:  80 / 80
   3 >> score for retrieved records:  100 / 100
======================================================================
Summary of valgrind results:

Valgrind issues:
==8578==     in use at exit: 0 bytes in 0 blocks
==8578==   total heap usage: 2,726 allocs, 2,726 frees, 275,983 bytes allocated
Invalid reads: 0
Invalid writes: 0
Uses of uninitialized values: 0
======================================================================
. . .
```

If the log file is missing, check for error messages in the terminal window. If your scores are imperfect, or missing, examine the rest of the log file for relevant error messages.

Note: we will **only** use this grading script to evaluate your submission.

What to Submit

For this assignment, you must place the three C source files you have completed into a single, flat[5], uncompressed tar file:

- `offsetFinder.c`
- `recordRetriever.c`
- `StringBundle.c`

You will submit that file to the Curator under the collection point `c04`. You should, of course, test your tar file with the supplied grading script before you submit it. That will ensure that your tar file is correctly structured, and help avoid unpleasant surprises.

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/

Grading

This assignment will be graded automatically, using the same grading code we have supplied, but using the same test cases for everyone.

We will also use Valgrind to check:

- whether you have, in fact, used dynamic allocation
- whether you have deallocated all the arrays properly (checked via the Valgrind log)
- whether your solution performs any invalid reads or invalid writes, indicating that you have array bounds issues
- whether you have allocated excessively large arrays in order to avoid invalid reads and writes
- whether you have any uses of uninitialized values; this could relate to array bounds issues, or failure to properly terminate your C-strings

You should use the supplied script to run your solution on Valgrind and check the resulting log file for indications of errors. A bonus of up to 10% will be applied to your score if your solution exhibits no such bad behavior.

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 each submitted C source 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 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.

Appendix: Quick Overview of `struct` Types

The **`struct`** mechanism in C has many similarities to the class mechanism in Java:

- Bundling of a collection of data values (*fields*) as a unit; data values are referenced by name.
- Can be passed as parameters or used as return values from functions.
- Can be assigned; values of corresponding data members are copied from source to target.
- Deep content (i.e., dynamically-allocated members) is not copied on assignment.

There are also important differences:

- All fields are public; there is no access control.
- Functions cannot be members of a **`struct`** variable.
- May be accessed by name or by pointer; Java objects can only be accessed by reference.
- Can be created by static declarations or by dynamic allocation (not used in this assignment).

Here's a common use for a **`struct`** type in C:

```
struct _Rational {

   int64_t top;
   int64_t bottom;
};
typedef struct _Rational Rational;
```

We could create a `Rational` object in either of these ways (statically or dynamically):

```
Rational R;                |       Rational *pR = malloc(sizeof(Rational));
```

In both cases, the fields in the new `Rational` object are (logically) uninitialized. For now, we will only consider the first case, in which the object was created by a static allocation (so it's stored on the stack).

When we create a **`struct`** variable statically, we can also initialize it (but only in the variable declaration):

```
Rational R = {37, 63};
```

We can implement functions to carry out operations on `Rational` objects (they just can't be member functions). For example:

```
Rational Rational_Add(const Rational Left, const Rational Right) {

   Rational Sum;   // create object to hold the result

   // initialize that object
   Sum.Top    = Left.Top * Right.Bottom + Left.Bottom * Right.Top;
   Sum.Bottom = Left.Bottom * Right.Bottom;

   // return (a copy of) it to the caller
   return Sum;
}
```

Note how the fields of a `Rational` object are accessed here; we use the name of the object, followed by the *field-selector operator* (`'.'`), followed by the name of the field: `Sum.Top`

What about accessing **`struct`** variables by pointer? Here's an alternate approach to implementing an addition function for `Rational` objects:

```
void Rational_Add(Rational* const pSum, const Rational Left, const Rational Right) {

   // initialize Sum
   pSum->Top    = Left.Top * Right.Bottom + Left.Bottom * Right.Top;
   pSum->Bottom = Left.Bottom * Right.Bottom;
}
```

Here, `pSum` is assumed to point to an existing `Rational` object created by the caller. We can access members of `*pSum` in two ways:

```
(*pSum).Top = Left.Top * Right.Bottom + Left.Bottom * Right.Top;
```

or

```
pSum->Top = Left.Top * Right.Bottom + Left.Bottom * Right.Top;
```

In the first approach, `*pSum` is actually a name for the object that `pSum` points to, but we have to put parentheses around that due to precedence rules in C.

In the second approach, the arrow operator (`'->'`) takes a pointer to a **`struct`** variable as its left parameter and the name of a field in that **`struct`** variable as its right parameter.

Most C programmers prefer the second syntax.

Appendix: Some Implementation Suggestions

Incremental development for Phase 3:

Begin by writing your own, simple testing code. All that's needed for this is to create a separate C source file (say, `myTestCode.c`), write a simple `main()` function there, and `include` directives as needed. Then:

- Copy a line from the GIS data file and hardwire it as a string literal into your `main()` function.
- Don't worry about writing your `createStringBundle()` function at first. Instead, write a simpler function in the same file as `main()` and pass it the string literal.
- A simple beginning point is to have your function just try to extract the first field from the GIS record, put that field into a dynamically-allocated **`char`** array, and print that field to standard output. Tinker with that until it's correct.
- At this point, gdb and Valgrind are your friends. Use them wisely.
- Once that works, try to find the next couple of fields, put them into **`char`** arrays, and print them.

Once this is working, put your parsing function(s) into `StringBundle.c`, using the specified interface for the primary function `createStringBundle()`. Compile with the supplied testing code, and see what results you get.

When that is working, try to get all the fields. You'll have to deal with empty fields at this point, and deal with stopping after the last field, whether or not that field is empty. From this point on, `gdb` and `valgrind` are surely your best friends.

Now, some hints about parsing…

As far as the Standard Library goes, the function `strtok()` is the most obvious approach. However, `strtok()` will not handle the empty fields correctly. If `strtok()` sees a sequence of delimiting characters (like two or more pipe symbols in a row), it treats them all as one delimiter and goes on to the next nonempty field. That won't do. Worse, strtok() will attempt to modify the contents of the C-string you are passing to it, and in this case that would violate **`const`** restrictions in the function interface.

On the other hand, processing the string character by character is tedious; that's why I wrote a customized variation of `strtok()` for my solution. You are, of course, free to process character by character if you like.

Another candidate from the Standard Library is `sscanf()`, which is like `fscanf()` except that it reads from a C string instead of a file. If you use `sscanf()`, you just solve two problems:

- how do you specify the delimiters?
- how do you tell `sscanf()` where to start reading, when you call it multiple times on the same string?

The second issue arises from basic differences between file-oriented I/O using streams and `sscanf()`. Using a file stream, `fscanf()` keeps track of where it stops reading in an input file, so it can resume at the correct place in the file when called multiple times. Operating on a character array, `sscanf()` has no automatic way to do the same thing.

For both issues, consult the course notes on I/O and parsing with C strings. Those show how to using *scansets* in `sscanf()`, and how to "restart" `sscanf()` when it's called multiple times on the same string.

The GIS records can vary considerably in content; some have more fields that are empty than others; some have an empty last field, and some do not. By default, the supplied test driver chooses a random set of records for testing. See the comments in `c04driver.c` for more details. You should run the test driver many times, in hopes of triggering every possible scenario. You might also edit `c04river.c` to increase the number of test cases used.

Helper Functions:

Helper functions improve development because they promote unit testing of smaller pieces of your design, they promote readability by shifting details of parts of the computation to different contexts, and they promote reusability. Good C programmers give careful thought about organizing their code to make use of helper functions. Actually, this has nothing to do with the programming language you are using.

The function `strtok()` is likely to be less useful than you might expect[4], but I did write a customized variation of it as a helper function in my solution:

```
/**  Returns string containing copy of next field of a GIS record.
 *
 *   Pre:
 *      When beginning to parse a GIS record string, pChar points to the
 *      first character in a GIS record string.
 *      When parsing a GIS record string for subsequent fields, pChar is
 *      set to NULL.
 *
 *   Returns:
 *      Pointer to a dynamically-allocated C-string holding the contents
 *        of the next field; this may be an empty string, but will never
 *        be NULL.
 *
 *   How it works:
 *
 *   If pChar is NULL, the function begins parsing at the character after
 *   the end of the previous field (which was presumably read during an
 *   earlier call to nextField() on the same GIS record string).
 *
 *   If pChar is not NULL, the function begins parsing at the first
 *   character in the GIS record string.
 *
 *   If pChar points to a delimiter ('|'), an empty string is returned.
 *
 *   Otherwise, a dynamically-allocated char array is created to hold the
 *   next field, using the assumption that no GIS record field will
 *   contain more than 200 characters.
 *
 *   Next, the function uses sscanf() to read characters until a delimiter
 *   is found, or the end of the GIS record string is reached.
 *
 *   Finally, realloc() is used to shrink the array holding the field to
 *   the minimum suitable size.
 *
 *   If pChar is NULL, the function begins parsing at the poi
 *
 *   Hints:
 *      Use sscanf() with a suitable scanset to read the next field.  See
 *      the notes on String I/O for examples.
 *      The function keeps track of the position in the GIS record string
 *      by making use of a static local variable.
 */
static char* nextField(const char* const pChar);
```

You are not required to implement the function described here, but writing it will give you a better understanding of how a proficient C programmer might approach this assignment. The function might also be very useful in future assignments.

I also wrote a function to make a dynamically-allocated copy of a given C-string:

```
/**  Makes a dynamically-allocated copy of a given C-string.  The new
 *   string is of minimum length to hold the contents of the original
 *   string.
 *
 *   Pre:
 *      str points to a proper C-string, or is NULL
 *
 *   Returns:
 *      pointer to a dynamically-allocated C-string holding the
 *      characters in *str; returns an empty string if str == NULL.
 */
static char* copyOf(const char* const str);
```

What the function does is simple, but useful in many programs that manipulate C-strings; that's why I made this a separate function. The decision to have the function return an empty string instead of **NULL** when `str` is **NULL** was based on how I wanted to use the function in my larger program.

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 this assignment on Valgrind:

```
#1071 wmcquain: soln> valgrind --leak-check=full --show-leak-kinds=all --log-file=vlog.txt --track-
origins=yes -v ./c04driver GISdata.txt results.txt
```

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

```
==5181== Memcheck, a memory error detector
==5181== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5181== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==5181== Command: ./c04driver GISdata.txt results.txt
==5181== Parent PID: 3128
--5181--
--5181-- Valgrind options:
--5181--    --leak-check=full
--5181--    --show-leak-kinds=all
--5181--    --log-file=vlog.txt
--5181--    --track-origins=yes
--5181--    -v
. . .
==5181== HEAP SUMMARY:
==5181==     in use at exit: 0 bytes in 0 blocks
==5181==   total heap usage: 2,075 allocs, 2,075 frees, 225,675 bytes allocated
==5181==
==5181== All heap blocks were freed -- no leaks are possible
==5181==
==5181== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==5181== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

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

On the other hand, here's (part of) what I got from an incomplete solution:

```
. . .
==3279== Invalid read of size 1
==3279==    at 0x4015CF: nextField (StringBundle.c:89)
==3279==    by 0x401406: createStringBundle (StringBundle.c:32)
==3279==    by 0x400C4A: main (c04driver.c:57)
==3279==  Address 0x5255216 is 0 bytes after a block of size 134 alloc'd
==3279==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==3279==    by 0x401565: copyOf (StringBundle.c:74)
==3279==    by 0x4013BA: createStringBundle (StringBundle.c:18)
==3279==    by 0x400C4A: main (c04driver.c:57)
==3279==
. . .
==3279== Invalid read of size 1
==3279==    at 0x401128: nextField (checkStringBundle.c:116)
==3279==    by 0x400FB7: refCreateStringBundle (checkStringBundle.c:68)
==3279==    by 0x400DA5: checkStringBundle (checkStringBundle.c:13)
==3279==    by 0x400CD7: main (c04driver.c:65)
==3279==  Address 0x5256176 is 0 bytes after a block of size 134 alloc'd
==3279==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==3279==    by 0x4010BE: copyOf (checkStringBundle.c:101)
==3279==    by 0x400F6B: refCreateStringBundle (checkStringBundle.c:54)
==3279==    by 0x400DA5: checkStringBundle (checkStringBundle.c:13)
==3279==    by 0x400CD7: main (c04driver.c:65)
==3279==
. .
==3279== HEAP SUMMARY:
==3279==     in use at exit: 219,473 bytes in 1,949 blocks
==3279==   total heap usage: 2,077 allocs, 128 frees, 225,461 bytes allocated
==3279==
```

```
==3279== Searching for pointers to 1,949 not-freed blocks
==3279== Checked 358,496 bytes
==3279==
==3279== 3 bytes in 3 blocks are indirectly lost in loss record 1 of 12
==3279==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==3279==    by 0x40153E: copyOf (StringBundle.c:70)
==3279==    by 0x4013F5: createStringBundle (StringBundle.c:27)
==3279==    by 0x400C4A: main (c04driver.c:57)
==3279==
==3279== 20 bytes in 20 blocks are definitely lost in loss record 2 of 12
==3279==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==3279==    by 0x4015E4: nextField (StringBundle.c:90)
==3279==    by 0x401406: createStringBundle (StringBundle.c:32)
==3279==    by 0x400C4A: main (c04driver.c:57)
==3279==
. . .
==3279== 570 bytes in 72 blocks are definitely lost in loss record 7 of 12
==3279==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==3279==    by 0x40163A: nextField (StringBundle.c:94)
==3279==    by 0x401406: createStringBundle (StringBundle.c:32)
==3279==    by 0x400C4A: main (c04driver.c:57)
==3279==
. . .
==3279== 800 bytes in 5 blocks are indirectly lost in loss record 10 of 12
==3279==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==3279==    by 0x401440: createStringBundle (StringBundle.c:39)
==3279==    by 0x400C4A: main (c04driver.c:57)
==3279==
==3279== 1,508 (80 direct, 1,428 indirect) bytes in 5 blocks are definitely lost in loss record 11 of
12
==3279==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==3279==    by 0x401380: createStringBundle (StringBundle.c:11)
==3279==    by 0x400C4A: main (c04driver.c:57)
==3279==
==3279== 216,635 bytes in 1,640 blocks are still reachable in loss record 12 of 12
==3279==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==3279==    by 0x4012EB: copyString (dataSelector.c:47)
==3279==    by 0x401259: loadRecords (dataSelector.c:37)
==3279==    by 0x400C16: main (c04driver.c:49)
==3279==
==3279== LEAK SUMMARY:
==3279==    definitely lost: 1,410 bytes in 204 blocks
==3279==    indirectly lost: 1,428 bytes in 105 blocks
==3279==      possibly lost: 0 bytes in 0 blocks
==3279==    still reachable: 216,635 bytes in 1,640 blocks
==3279==         suppressed: 0 bytes in 0 blocks
==3279==
==3279== ERROR SUMMARY: 16 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.

Change Log

Version	Posted	Pg	Change
1.00	Jan 19		Base document

Notes

[1] For a discussion of the history of this, see https://en.wikipedia.org/wiki/Newline.

[2] The GIS record files were obtained from the website for the USGS Board on Geographic Names (www.usgs.gov/core-science-systems/ngp/board-on-geographic-names/download-gnis-data). 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:

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.

The exact meaning of the various fields is not important for this assignment.

[3] The descriptions given here were obtained by Google searches targeting the Open Group website (e.g., for "fgets opengroup"). I find that to be the most useful site for concise descriptions of the C Standard Library. Other sites may be better if you need examples showing how to use the functions.

[4] The `strtok()` function modifies the string, which may violate **`const`** restrictions. Also, `strtok()` chops the string it's applied to into a sequence of substrings, but you usually need to copy those substrings into stand-alone C-strings in order to avoid logical errors. In addition, `strtok()` treats a sequence of delimiting characters as if there were a single delimiter, so it can miss empty fields.

[5] A *flat* tar file is one that does not imply a directory structure for the files it contains. In other words, if you unpack a flat tar file, the files will all be created in the target directory, not in subdirectories. You can easily tell whether a tar file is flat by executing "`tar tvf`" on the tar file and seeing if the file names show path information; if they do, the tar file is not flat.