C Programming Parsing Formatted C Strings

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.

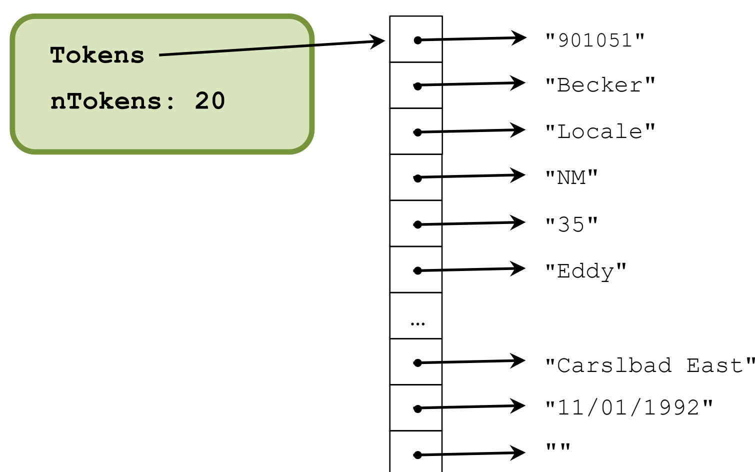
Since we will have multiple fields, and we must use **`char`** pointers when we allocate the arrays, we can use an array of **`char`**`*` to organize the set of fields we parse out of a record string. It is tempting to just use an array of 20 **`char`**`*` for this, but we should strive for a more flexible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



You will implement the following function, which takes a pointer to a GIS record string, creates a corresponding `StringBundle` object, and returns a pointer to that new `StringBundle` object:

```
/**  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 may use any of the functions declared in the Standard Library, including the string manipulation functions. In particular, the following functions may be useful, or even necessary:

```
malloc(), calloc(), realloc(), free()
strncpy(), memcpy()
strlen()
sscanf()
```

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, by calling another function you'll be implementing:

```
/**  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.

You may, and are encouraged to, write additional 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 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 the two Standard Library functions were not quite what I needed. These functions are described in more detail in the appendix Some Implementation Suggestions.

Supplied code

Download the supplied tar file and unpack it in a CentOS 7 directory. You will find the following files:

```
c04driver.c             test driver
StringBundle.h*         header file for required functions
StringBundle.c          C shell file for required functions and private helpers
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:

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

Invoke the driver as:

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

If invoked without -repeat, the program will choose a random set of GIS record strings from the specified GIS data file, and use those strings for testing. You should specify the supplied GIS record file. The results file will show the test strings that were used, information about anything that's wrong with your StringBundle objects, 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, 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.

What to submit

For this assignment, you must place your source code in the file `StringBundle.c`, and submit that file to the Curator. You will be allowed multiple submissions; the final one will be graded.

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 run multiple tests on your submission, using records with different mixtures of empty and nonempty fields.

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 the submitted file:

```
//  On my honor:
//
//  - I have not discussed the C language code in my program with
//    anyone other than my instructor or the teaching assistants
//    assigned to this course.
//
//  - I have not used C language code obtained from another student,
//    the Internet, or any other unauthorized source, either modified
//    or unmodified.
//
//  - If any C language code or documentation used in my program
//    was obtained from an authorized source, such as a text book or
//    course notes, that has been clearly noted with a proper citation
//    in the comments of my program.
//
//  - I have not designed this program in such a way as to defeat or
//    interfere with the normal operation of the 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:

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 empty fields 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, 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==
. . .
==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
2.00	Aug XX		Base document.