Strings and Decision-making in C — Finding Occurrences of Characters

The C language does not have a formal string type; instead, a string is stored in an array of **char** values, with a special **char** value, denoted by **'\0'**, stored right after the last character of the string. This character is called the *string terminator*. If we declare and initialize a string this way:

```
char Str[7] = "string";
```

We get an array holding 7 **char** values like this:

's'	't'	'r'	'i'	'n'	'g'	'\0'
0	1	2	3	4	5	6

An array is simply a collection of elements, all of the same type, which can be accessed by specifying the position of the element within the collection by writing an expression of the form `A[index]`, where `A` is the name of the array and `index` is a valid position within the collection.

In the example above, `Str[2]` would be the element at position 2 (which has the value `'r'`).

The maximum number of elements the array can store is called the *dimension*; in the example above, `Str` has dimension 7. When creating a C string, we must be careful to make sure the dimension is large enough to allow room for the special terminating character, `'\0'`. (Fortunately, that will not be an issue for you to handle in this assignment.)

Valid positions in an array range from 0 to *dimension* - 1. You might wonder what happens if you try to access an invalid position; that is, one that is outside the bounds of the array. It is sufficient for now to say that the results are seldom good, and always logically incorrect. (You will have to deal with that issue in this assignment.)

The C language provides a fairly rich set of Library functions for handling strings; however, you will not need any of these for the assignment. In fact, you are not allowed to use any of those functions; specifically, you must not include the Standard header file `string.h` in your solution. Violating this restriction will result in a score of 0, which will be assigned administratively after the assignment has closed.

For this assignment, you will implement a C function that will take a C string and a single character as input, determine the locations of all occurrences of that character in the give string, and return the sum of those locations. For example, suppose you were given the character `'l'` and the following string:

```
          1111111111222222222233333333334444444444555555555566666666667777777777888888888899999 9
012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345

The penalty good men pay for indifference to public affairs is to be ruled by evil men. -- Plato
        ^                                       ^                      ^         ^          ^
```

Your function should find the occurrences of `'l'` flagged above, and compute the return value:

```
8 + 48 + 71 + 81 + 92 == 300
```

If the given character does not occur in the given string, your function should return the value -1.

The required interface for your function is:

```
/** Computes sum of locations of given character in a given C string.
 *
 *  Pre:  Str is a valid C string
 *        Target is set to the desired character
 *
 *  Returns:  sum of indices at which Target occurs in Str;
 *            -1 if Target does not occur in Str
 *
 *  Restrictions:
 *    You may not use any Standard Library string functions; if you
 *    do not include string.h, you will satisfy this requirement.
 *    You may not write output within your submitted solution; but,
 *    you may write output during your testing.  However, if so, you
 *    must delete any such code, and any include for stdio.h, before
 *    submitting your solution.
 *
 *    Violations of these restrictions will result in a score of 0.
 */
int sumIndicesOf(const char Str[], const char Target);
```

The use of **const** in the function interface deserves some commentary. When a parameter to a function is declared with the type qualifier **const**, that indicates that the function is not supposed to modify the value of the parameter. To quote from section 6.7.3 of the C11 Standard:

> If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.
>
> If the specification of an array type includes any type qualifiers, the element type is so qualified, not the array type. If the specification of a function type includes any type qualifiers, the behavior is undefined.

Perfectly clear, right? Probably not… It's worth understanding some things about the C Standard right from the beginning. The Standard reads like a legal document, with the intention of being fully precise except when it is intended to leave things undefined. Notice the quoted paragraphs do not say that attempting to modify a const-qualified type is illegal. Rather, they say if an attempt is made to do so, *the behavior is undefined*.

So, how should you interpret that? First, the Standard does not say that a C compiler must (or even should) treat a violation of const as an error (or even generate a warning). That said, gcc will at least generate a warning message. Second, if the result of a certain operation is undefined, you should avoid doing it.

Finally, this tells you some important things about the C language. Some people would say this tells you terrible things. I disagree with that, but that's purely personal preference. Things in C are not necessarily what they seem, unless you are paying very close attention to the Standard. The use of const in C is best viewed as a statement of the designer's intent. That intent can be violated without breaking the rules, but it is probably unwise to do so. On the other hand, in some circumstance, it might be necessary to do so. It might even be brilliant to do so. The Standard provides explicit warning that violating const may be dangerous. If you choose to do so, it's at your own risk, and anything that goes wrong is your own fault, not the fault of the language.

I suggest you look at it this way: using a hammer to drive a nail into a board might appear to work, at least at first, but there's a really good chance the result will be less than ideal.

Getting Started

A tar file is available, containing the testing/grading code that will be used to evaluate your solution:

`c01driver.c`	test driver... read the comments!
`sumIndicesOf.h`	header file declaring the specified function... do not modify!
`sumIndicesOf.c`	C source file for implementing the specified function
`c01Grader.h`	header file declaring the test case generator... do not modify!
`c01Grader.o`	64-bit object file containing the test case generator

Download the tar file `c01Files.tar` from the course website and save it on your CentOS 7 installation (or on rlogin), in a directory created for this assignment. Unpack the tar file there. You can do this by executing the command:

```
CentOS> tar xf c01Files.tar
```

The file `sumIndicesOf.c` contains a trivial, nonworking implementation of the specified function. You must edit this file to complete the function. That is the only file you will be submitting, so if you make changes to any of the other supplied files, there is an excellent chance we will be unable to compile your submission, or that it will run differently for us than for you. We will not make any accommodations for such errors. Therefore, do not modify any of the other supplied files.

Implementation and Testing

You can compile the given code by executing the command:

```
CentOS> gcc -o c01 -std=c11 -Wall c01driver.c sumIndicesOf.c c01Grader.o
```

You can run the test/grading code by executing the command:

```
./c01 K
```

where K is the number of test cases you want to be analyzed. Read the comments in `c01driver.c` for more information. `c01driver` will use the supplied grading module to create test data, run your function on that test data, check your answers, and create a file, `c01Log.txt`, showing the results, which might be something like this once you've completed your solution:

```
Running case 1:
   Looking for occurrences of 'l' in the string below:

            1111111111222222222233333333334444444445555555555...
   0123456789012345678901234567890123456789012345678901234567890...

   The penalty good men pay for indifference to public affairs ...
         ^                                      ^           ...
   You reported the sum was: 300.
   Excellent!

Running case 2:
   Looking for occurrences of ' ' in the string below:

            1111111111222222222233333333334444444445555555555...
   0123456789012345678901234567890123456789012345678901234567890...

   To announce that there must be no criticism of the President...
     ^        ^    ^     ^    ^  ^  ^         ^  ^   ^         ...
   You reported the sum was: 4351.
   Excellent!
   . . .

Score: 10 / 10
```

Note that the `Target` character may not be a letter, or even a digit. Also, note that the search is case-sensitive (which actually makes things simpler.

If you try this with the original version of `sumIndicesOf.c`, the code will compile, but the results will be (mostly) incorrect. Each execution of driver will produce a different set of test data, if you use the `-repeat` switch (with the same number of test cases as before):

```
./c01 K -repeat
```

Now, the test data file created by the most recent run without the switch will be reused. That allows you to focus on a fixed set of test cases while you are debugging.

You should test your solution thoroughly; the given testing code generates random test data, and there is no guarantee that it will cover all cases unless you run it a sufficient number of times. You might want to begin by running with K == 1.

Submission and Grading

You should not submit your solution to the Curator until you can correctly pass tests with the given testing/grading code.

Submit your completed version of `sumIndicesOf.c`, after making changes and testing. Your submission will be compiled, tested and graded by using the supplied code, but that will be done manually after the due date. A TA will also check to see if your solution violates any of the restrictions given in the header comment for the function; if so, your submission will be assigned a score of zero (0), regardless of how many tests it passes.

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/

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 grading code.
//
//    <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

Any changes or corrections to the specification will be documented here.

Version	Posted	Pg	Change
1.00	Aug 26		Base document.