CS 455 Lab 13: C strings

Fall 2021 [Bono]

Goals and Background

To program in C++ you need to understand C strings too, because C is a subset of C++, and you will run into C language features in some C++ programs, and at times you will need to interface with C library code. For CSES students in this class an additional reason to practice a little bit more with C is to be better prepared for other courses in the curriculum that use C, such as the operating systems and networks courses.

We will still use the C++ compiler here (so we will use the usual compile command for a single-file C++ program), but because the point of the lab is to work more with C, you will be restricted on C++ specific features you can use. You may use *only* the following *C++* features in your code for this lab:

- call by reference
- ostream output (i.e., using <<)
- new / delete

(You should be ok if you *don't* use the C++ `string` class or the `istream` input methods, and if you *don't* `#include` any additional header files besides what's already there.)

Reading and reference material

- cplusplus.com tutorial on Character sequences and Dynamic memory.
- Thurs 12/2 lecture on Dynamic arrays and C strings
- <cstring> cplusplus.com reference pages for several functions to manipulate C strings

Review of C strings:

First, here is a review of some key features of C strings:

- A C string is not a class / object (C does not have classes). It's a null-terminated array of characters. That is, after the characters for the string contents, a `'\0'` char appears (this is called the null character). An empty string consists solely of the null char. So the array has to be at least one char larger than the number of chars in the string you want to store. Put another way, it's a kind of partially-filled array where instead of keeping track of the current length in a separate variable, it instead uses this null character sentinel to indicate the end of the part of the array that's in use. So, `<cstring>` functions, such as `strlen`, are implemented with a loop until it reaches the null char.

- C string variables can be defined using a regular array of characters or a pointer type (that would point to a dynamic array of characters), e.g.,

  ```
  char str[10];
  char *str2;
  ```

  So `str`, defined above, would be able to hold a 9-char string, such as "ideologue", but does not have room for "ideologues". Because those two types (array and pointer) are somewhat interchangeable, you can use either kind as a parameter to C-string functions. Unlike other functions that take an array as a parameter, the C-string functions generally do not need the size passed to them, because they assume there will be the terminating null char in the C string.

- The `char*` version (`str2`, above) has *no* memory allocated for it until you create a dynamic array. And C-string functions such as, `strcpy` do *not* allocate that memory for you, and do not check whether you try to write beyond the end of the destination C-string array. In fact, two common pitfalls of working with C strings are (1) forgetting to create the array if you are using char* variable, and (2) going beyond the array bounds. (`strncpy` is like strcpy, but it won't write more than `n` chars, even if it hasn't gotten to a null char yet.)

- Some of the C string library functions (documentation linked above) may come in handy for this lab, but you are not required to use them.

Exercise 0

Take a look at the starter code in `phone.cpp`: it reads from standard input line by line, until it reaches end of file, just echoing out each line to the screen. You can provide input from the keyboard, or you can use input redirection from a text file. Compile and run the code, with input from the keyboard, and again with input from one of the data files provided.

Exercise 1 (1 checkoff point)

You're now going to make some changes to `phone.cpp` so it can parse phone numbers. Assume the input on each line will be a single phone number with the format shown by example below:

```
213-740-4510
```

For each such line of input, parse the line into it's three constituent parts: area code (the first 3 digit chars), prefix (the second 3 digit chars), and line number (the last 4 digit chars). Each one of these will be a C-string with the names shown in the code below. I.e., the parts will be C strings, not ints. Print out the parts using the following lines of code:

```
cout << "area code: '"  << areaCode << "' " << endl;
cout << "prefix: '"  << prefix << "' " << endl;
cout << "line number: '"  << lineNumber << "'" << endl;
```

You may assume that the input will be in the correct format (i.e., you do not have to do any error-checking.) You may also leave in the "LINE READ" statement that echoes out the whole line of input. Note: there are a few useful named constants defined for you already in the code.

Test your code on input from the keyboard, and on the test data file `inPhone`. Do a run such that you save the output from `inPhone` into a file called `outPhone`.

Exercise 2 (1 checkoff point)

We might think of each of these parts of the phone number as a field (that we might store in some record). Refactor your code so that you don't have to repeat the part to read in a single field. Define a new function called `readField` whose parameters are the line buffer, a starting location in the line buffer, a field length, and the destination C string. It copies the indicated field into the destination C string. It will be more convenient if the start location is passed by reference so that it can be updated to where it ended up after reading the field. Your refactored program will have three calls to `readField`. It should generate the exact same output as your version from Exercise 1.

If you are a more experienced C programmer, and want to use a different interface that uses pointer arithmetic, you may do so.

Exercise 3 (1 checkoff point)

Make a copy of your program from Exercise 2 and call it `namePhone.cpp`

Enhance `namePhone.cpp` so that it now can read a name and phone number from each line, saving the name as one field, and the three parts of the phone number into the three other fields, like before. The format of a line will be as follows:

```
George H. W. Bush:713-255-9190
```

So, to generalize, the name is not a fixed width, can consist of 0 or more words, and will be delimited by the ':' character. The phone number will appear directly after that ':', and will have the same format as described for the last two exercises.

Print out the name using the following line of code

```
    cout << "name read: '" << name << "'" << endl;
```

So the complete output for this line of input would be:

```
LINE READ: George H. W. Bush:713-255-9190
name: 'George H. W. Bush'
area code: '713'
prefix: '255'
line number: '9190'
```

Test your code on input from the keyboard, and on the test data file `inNamePhone`. Do a run such that you save the output from `inNamePhone` into a file called `outNamePhone`.

Checkoff for DEN students

When you click the `Submit` button, it will be looking for and compiling (for source code) the files `phone.cpp` and `namePhone.cpp`. Make sure you put your name and loginid in all the files you submit.
