CSC 325 Lab 6

A String Library

Table of Contents

Due by 11:59 pm, **Tuesday, November 8 (Don't forget to vote!)**

1. Lab Goals

1. Practice using the C string library, both from the inside and out.
2. Read manual ("man") pages to understand a function's specification.
3. Systematically write test cases to help ensure correct program behavior.
4. Obtain additional experience with pointers, strings and memory layout, and dynamic memory allocation.

2. Lab Overview

For this lab, you will implement your own version of the core functions in C's string processing library.

In C, strings are just blocks of memory (e.g., static arrays, or a dynamically allocated block of bytes) that contain one character after another and are **terminated with a special null-termination marker**.

The functions in the string library abstract the details of memory and null-termination from the user to make strings less painful to use. Unfortunately, while they're much better than nothing, the functions are still quite low-level. For example, you cannot concatenate strings with a + operator like you can in many other languages, and using `strcat` requires that the user allocates and provides sufficient memory.

Since strings are everywhere, it's extremely likely that you will find yourself using these functions frequently. A great way to learn how they work is to implement them yourself, so that's what we will be doing in this lab!

You will be implementing the following functions (sorted alphabetically). Before you begin implementing these, spend some time considering the order in which you write them. Some of these are easier to write than others, and some of these functions may be needed to implement others.

- `strcat`
- `strchr`
- `strcmp`
- `strcpy`
- `strdup`
- `strlen`
- `strstr`

Once you've written your implementation of these functions, you'll write a small number of test cases for each to compare your version with that of the built-in C library. This should help you to convince yourself that your implementation is correct.

3. Handy References

- You can read up on some documentation on C libraries and header files.
- Use the `man` command on the terminal to view the manual page for a function that you're curious about using, e.g. `man strlen`. You can also use google by searching for "man strlen" to get the same information in a possibly-easy-to-read format.
  - Note: if you prefer to look at manual pages on the web, linux.die.net has an archive of documentation (e.g., strlen). In general though, it's better to get the manual from **your system**, since that better reflects the reality of the environment you're working within. Another system might implement a function slightly differently, and if you're reading the manual online, who knows which version you're looking at?
- Here is example of how to view the `strlen` manual page from the command-line:

```
$ man strlen
```

4. Lab Starting Point Code

4.1. Getting the Starting Point Code

Download the starter files from Canvas:

```
Makefile        my_strings.c    my_strings.h    stringtester.c
```

4.2. Starting Point Code

The files included in your repo are:

- `Makefile`: builds the `my_strings.o` object file and the `stringtester` executable file
- `my_strings.c`: starting point for your implementation of the string library. Your solution goes here.
- `my_strings.h`: the header files for your implementation of the string library. **You should not modify this file.**
- `stringtester.c`: code that demonstrates correctness of your implementation of the string library. You will need to write at least two test cases for each library function.

5. Lab Requirements

1. For each function, your implementation (in `my_strings.c`) should behave exactly as the C library's version does (when given valid inputs). **This includes matching the C library's argument types and ordering, return value, location of null-terminators, and final state of memory.** Read the manual pages to get a formal specification for each function.
2. For each function, **you should write two or more documented test cases in `stringtester.c`** to demonstrate that your version of the function is working as expected. To the extent that you can, you should vary the tests to make them cover different situations that may arise while using your library.
3. Your test cases **should** compare directly against the C string library. It's fine to make C string library calls inside of `stringtester.c`, but **you CAN NOT use any function from strings.h** in your implementation in `my_strings.c`
4. You should avoid copy-pasting code as much as possible. If you find yourself needing to perform the same operation in more than one function and it's more than a couple of lines long, **write a helper function**. The helper function need not be exported as part of the library, it can be a private function for your library to use.
5. For full credit, your solution should be well-commented, it should not use global variables, and it should be free of valgrind errors.
6. **Your library should not print any output except in cases where the standard C library also prints output**. (I'm not aware of any). It's fine if you want to include some printing to help you debug your program, but you must remove output from your library prior to submitting it.

6. Tips

- Test your code in small increments. It's much easier to localize a bug when you've only changed a few lines. You don't need to write every function before you start

working on test cases—you may find it beneficial to write them before moving to the next library function.

- In many functions, you may need to check for the special end-of-string null-terminator character. There are two ways to represent the null terminator:

    a. As a **character**, the null-terminator is `\0`. The `\` indicates that it's a special character, much like you use the `\n` character for a newline or `\t` to insert a tab.

    b. As a **numerical value** (recall that characters are one-byte numbers defined by ASCII), **the null-terminator is `0`**. Since characters are represented as 8-bit integers in hardware, an 8-bit numerical `0` value corresponds to the end of a string.

    c. Think about how you might use one function to implement another. For example, you might want to use `my_strlen` inside of `my_strcat` to tell you where the end of the first string is. Re-using code in this manner will cut down on the total amount of code that you have to write, and it'll make your code easier to reason about and debug.

7. Submitting your Lab

Please remove any debugging output prior to submitting.

Zip your Lab6 folder and submit it to Canvas.