Lab 2: String manipulation

CS 429: Fall 2019

Assigned: September 26th, 2019
Due: October 11th, 2019, 11:59 PM
Last updated: September 26, 2019 at 18:54

1 Introduction

Your task is to re-implement a few of the functions of *string.h*. *string.h* is the library used to manipulate array of bytes (strings) in C. You will be given a small amount of code that tests your implementation as some guidance and a header file. The idea here is that you will be given tasks like in the real world: you are given what are the inputs and outputs (called the signature of a function) of your functions and what they should do; you are responsible for writing the code that actually does it.

START EARLY!

2 Logistics

The lead TA for this lab is Henrique Fingler.

This lab should be tested on a UTCS lab machine (linux_64). There might be differences in Mac and PC architectures that might give different answers, that's why we ask that you test on a lab machine.

This is an *individual* project. You may not share your work on lab assignments with other students, but feel free to ask instructors for help (e.g., during office hours or discussion section). Unless it's an implementation-specific question (i.e., private to instructors), please post it on Piazza publicly so that students with similar questions can benefit as well.

Please do the assignment as you're supposed to; there are ways to write code in C and get the same code in assembly, don't do that. Don't copy code from anywhere, do it yourself, it's very important for this class and next classes.

You will turn in a tarball (a `.tar.gz` file) containing the source code for your program through Canvas. The file given to you is also a tar file, see the Submission section for instructions on how to unpack it.

Any clarifications or corrections for this lab will be posted on Piazza.

Since this is the second time we are giving this lab out, we will be a little lenient. You are expected to do every function since you will have plenty of time and support from TA's. Some corner cases won't take points off, like concatenating into a string that does not have enough space to hold all characters (in fact, you will exploit this in the next two labs), so you should assume that the inputs are correct and aren't maliciously developed, like overlapping pointers and buffer overflowing.

Always remember that strings are null-terminated, a.k.a., the last character is a zero. Assume all string inputs will be like this. In some functions you'll be required to put a zero at the end, when concatenating two strings, for example.

3 How to Start

Read and understand the code skeleton and the Makefile that was given to you, some details below:

- **my_string.h**: this file contains the signatures of the functions you have to implement and links to their documentation. The only difference between these signatures and the ones from string.h is that there's a prepended "my_" on each. This is done so that you can use the system's string functions and yours at the same time.
- **my_string.s**: the assembly file where you will write the code for a subset of functions. The

  ```
  .globl <function_name>
  ```

  is there to say that external code can call that function.
- **my_string.c**: the C file where you will write the code for a subset of functions.
- **Makefile**: the Makefile. Use this to compile your code. Typing "make" will compile your code and link it with the main.c file. **HIGHLY** recommended that you understand what Makefiles are and do: `http://mrbook.org/blog/tutorials/make/`. You can also type "make clean" to remove the .o and the binary files. So if you are typing "make" but your main file is not being updated, do "make clean ; make", this will rebuild all files from scratch.
- **grade.c**: This is the file that will use your string.h implementation, you can use this to test your code and compare it against the system's. This file is not important for submission since the TA's will use their own tests.
- **main.c**: A main function given to you in case you want to write tests in a different way and have full control over it. You don't need to use or submit this file, but it might be useful.

Read subsection Testing below now, too. The output of running the testing script will sort of explain what you're supposed to do.

4 First task

4.1 Strings and C

Familiarize yourself with how strings work in C, here are some pointers:

- `https://en.wikipedia.org/wiki/ASCII`

- `https://www.tutorialspoint.com/cprogramming/c_strings.htm`
- `https://en.wikibooks.org/wiki/C_Programming/String_manipulation`
- `http://www.cs.virginia.edu/~evans/cs216/guides/x86.html`

This part is not required, you don't have to submit any solution to this part, but it will help a lot.. Try doing the following, while either reading the C reference (`http://www.cplusplus.com/reference/cstring/`) or googling "how to do X in C" (because we know you're going to do it anyway). You should also print the string after every step:

1. Read a string (let's call it A) from the user (standard input, aka the keyboard)
2. Find the length of the string, both by hand (using a for or while loop) and by using a function
3. Find a way to remove the last character of the string, so if you read "hello", modify the string such as it's "hell" now.
4. Allocate a new string B (using malloc because you don't how during compile time how long the string is) and copy A into B.
5. Append the letter 'a' to A.
6. Compare A to B (as a whole), print yes if they are equal, no if not.
7. Compare the first 5 or the length of B, whichever is bigger, characters of A and B, print yes if they are equal, no if not.
8. Find a way to copy the last 3 characters of A into the beginning of B. if A was 'hella', B would be 'lla'.

4.2 Assembly

Next, you should understand enough assembly to code the main task. The amount of assembly you need to know is very small; you pretty much need to use addition/subtraction (add/sub instructions), moving registers to and from memory (mov instruction) and compare/jump to do loops (cmp, jmp, je and maybe some other you'd like to use).

You should also understand how parameters are passed in assembly. This will be useful for this and the next two labs. It's not complicated, there's just a convention that says the first six parameters of a function (in this lab we only use 1, 2 and 3 parameters) are passed in a specific register order.

Here's some reading you should do:

- http://cs.lmu.edu/r̃ay/notes/nasmtutorial/ : you don't need to understand everything here, for example, you won't be using any *call* or *syscall*, neither defining a main function, since it's in C already. You can stop reading it at Mixing C and Assembly Language. And we won't need the "nasm" compiler, this is just taken from their compiler guides.
- ../docs/guides/x64_cheatsheet.pdf A nice cheatsheet that you can use. You should read the entire thing, although some won't be required, such as: Special Arithmetic Operations, Conditional Move Instructions and Dynamic stack allocation. You're encouraged to read and understand those, though.

Understand what is caller-save and callee-save. Prefer caller save registers to you don't have to save them (push into the queue and pop before returning). Here's a good list: https://stackoverflow.com/questic registers-are-preserved-through-a-linux-x86-64-function-call.

Here's a small list of the ones you should prefer to use: rcx, rdx, rsi, rdi, r8, r9, r11. If you need more, you can save r12 through r15 and use them, there's an example in the assembly file given.

5 Main task

Your task is simple: to implement a subset of string.h in either C or assembly, what functions and in what language are defined below.

And again, to check what each function's input, what it must output and do, check this url: `http://www.cplusplus.com/reference/cstring/`. Each function has an example of its usage, you can copy and paste it into a C file and run to see what it does.

The functions required are listed below, the number in parenthesis is a difficulty number (1-3); you should start with the easier ones and also start with C, it's easier to do assembly after figuring out the patterns.

C:

- strchr (1)
- strncpy (2)
- memmove (2, read below)
- strncat (2)
- strncmp (2)
- strcmp (2)
- strrchr (3)
- strstr (3)

Assembly:

- strlen (1)
- memset (1)
- strcpy (2)
- strcat (2)
- strcspn (3)

Most of these have a very similar pattern. For example, memmove contains pretty much a copy of strcpy's code. strncat and strncmp require the same loop, from 0 to n. strstr might use something similar to str(n)cmp. strcat requires finding the end of the string, which is pretty much what strlen does.

Another hint; before actually writing code, write the algorithm for each using a pseudo-language so you understand the structure of the code you will have to write and to take corner cases into consideration, here's an example:

```
my_strcat(string1, string2):
  pointer = string1 //start of string1
  move pointer to end of string1 //pretty much strlen's code
  source = string2 //start of string2

  while (source) is not end of string (0x00, byte zero):
    copy (source) into (pointer)  //copying the content (one character
                                  // from memory) of the string, not the
                                  //pointer itself, hence the parenthesis
    advance source pointer //add one to it (advancing to the next character)
    advance source pointer //add one to it (advancing to the next character)
```

Now that you have the structure, it's much easier to write code.

Also, most functions require just one for loop, a few require two, and strstr is the only one that requires two nested loops.

For memmove, if you read the documentation you will note that it is very similar to memcpy with one difference: it accepts two arrays that overlap. It's important because you cant copy character by character because you might overwrite the source array. The hint here is to create an intermediate array (using malloc), copy the source array into it, then move data from the array you created into the destination.

6 Hints

Here are a few hints from frequently asked questions last semester.

6.1 Assembly

- You can't copy memory-to-memory. For example, the following instruction is illegal: *movq (%rax), (%rbx).*
- The last letter in a *mov* instruction means the amount of bytes it will copy. *movb* will move a byte, while *movq* will move a quad (8 bytes).
- in a *mov* instruction, both parameters have to match the size. For example, if you want to read one byte from memory (the size of a character, so this is what you will use in this assignment pretty much) into the *a* register, you would do: *movb (some_addr), %al. al* is the lowest byte of the register, use this cheatsheet to see all registers. Another example: *movb (some_addr), %ax* is illegal because it's trying to move one byte into a 2 byte register (size mismatch).
- Pointers are always 64 bits. So if the parameter of a function says *char**, this is a 64 bit pointer that points to one byte characters. For example, for *strlen*, the first parameter is a pointer to a string and, because it is the first parameter, it will be (by convention) in *%rdi.* If you want to see what is the first character, you would put it in a one-byte long register, eg: *movb (%rdi), %r11b*, and now %r11b has the first character.
- Read section 4.2 of the cheatsheet linked above to see how to easily access positions of an array. For example *mov %cl, (%esi,%eax,4)* would move cl into $(esi + (eax * 4))$. Here's another nice reference.

6.2 Testing

There is an automated python script to test your code, it is in *grade.py.* Open it and you will see some example tests. It works by calling the function in *grade.c*, which tests one function call when it is executed using the parameters.

First thing you should do is just straight up test the code without implementing anything to see the output. You do so by running *make*, then *python grade.py.*

Feel free to share test cases on Piazza.

7 Submission

You will submit a file called "lab2.tar.gz".

To create the `.tar.gz` file you need to submit, make sure all the files of the lab are inside a folder called lab2. While one directory above lab2 (as in, your shell is not inside the lab2 directory) use the following commands.

```
$ tar -cvzf lab2.tar.gz ./lab2
```

This should be done on a UTCS Linux Machine. You should search what every letter passed as parameter does, but here's a tldr: c for create, v for verbose, z as bzip compression and f for saying the next parameter is the file output. The next parameter is the directory you are compressing (like zipping a directory, just a different format).

To ensure that the tarball submission was created properly, move the tar file to a different folder (one idea: *mkdir test ; cp lab2.tar.gz test/ ; cd test*) and try untarring it with the following command:

```
$ tar -xvzf lab2.tar.gz
```

Almost same thing as before, except you have an x for extract instead of a c.

This should create a folder `lab2` with the files you want to submit. Please follow this part, if you don't the grading script might break and give you a lower grade.

Also, after submitting to Canvas, download your submission to check if your files are really there. If you submit empty files (this is common, unfortunately) you will not get a grade.