EECS 211 Lab 3

Strings

Winter 2019

Today we are going to practice manipulating "strings".

Getting Started

Let's get started by logging into a remote Northwestern server. We did this last week, but if you need help remembering the steps, they are included below.

The list of remote Northwestern servers can be found here: http://www.mccormick.northwestern.edu/eecs/documents/current-students/student-lab-hostnames.pdf

Windows

Open PuTTY. You'll need to enter a hostname to login to. The link on the right will take you to a list of student lab hostnames (such as *tlab-03.eecs.northwestern.edu* or *batman.eecs.northwestern.edu*). Ensure SSH is selected, then press Open. When prompted, enter your EECS username and password (not necessarily the same as your NetID password) and you're good to go.

Mac/Linux

Open up your terminal. At the prompt, use the ssh command of the form

```
$ ssh [eecs-id]@[eecs-host].eecs.northwestern.edu
```

where *[eecs-id]* is your EECS username (probably your NetID) and *[eecs-host]* is replaced by one of the EECS hostnames from the list of student lab hostnames (such as *tlab-03.eecs.northwestern.edu* or *batman.eecs.northwestern.edu*). When prompted, type in your EECS username and password (not necessarily your NetID password), press Enter again, and you should be logged in remotely!

Getting the code

Recall our basic Unix commands: *cd*, *ls*, *mkdir*, and *pwd*. What do they stand for and what do they do? Ask your TA if you don't remember.

Or ask Google.

Use the following *curl*-and-*tar* pipeline to download and extract the code into your directory of choice. We suggest that you keep your EECS 211 files in an `eecs211/` subdirectory of your home directory, but it's up to you.

```
$ curl $URL211/lab/eecs211-lab03.tgz | tar zxv
```

You should now have a directory called eecs211-lab03.

Setting up the build system

Type the `$ dev` command into the shell to ensure that you are using the correct developer toolset. You must do this every time you open a remote connection and plan on compiling C code.

Writing the code

Navigate to your eecs211-lab03 directory, and open up src/lab2.c in Emacs using

```
$ emacs -nw src/lab3.c
```

Notice that there is already some skeletons of functions and some code in *main()* here.

*const char *str_chr(const char *s, char c)*

First, find the function called *str_chr*. We are going to use this function to determine if the character `c` exists in the string `s`, and if so, where. If you remember from class, we have a few ways of iterating, most notably `while` which is what you will use for this function.

Notice that *str_chr* is going to return a `const char *`.

While loops

As we learned in class, a `while` loop has the following syntax:

```
while (<expr>) {
    // Looping through code here
    // Until <expr> is false
}
```

Note that in while loops we usually will use a boolean expression for <expr> (an expression which returns `true` or `false`.)

Use a while loop inside our *str_chr* in order to see if `c` is every equal to any one of the charcters in `s`. Make sure to use a `return` statement to return the `char *` if we find it (or a `NULL` if nothing is found).

Remember that we have the ++ operator to help us.

Once you think that your function works as intended, save and and try compiling and running it. If you remember from last week, we used the *make* command in order to turn our C file into machine code. Run:

C-x C-s to save

```
$ make build/lab3
```

Remember, make works as follows: `$ make [target]`. Target is usually the name of the executable file that will be built by the make command.

If everything works, if we list the files in `build`, we should now see a file called lab3. Enter the command

```
$ build/lab3
```

See if your value looks right! If it doesn't, don't worry, Rome wasn't built in a day. Try and see what went wrong. Play around with the value of `s` and `c` to see how it affects the result.

Error messages may look scary, but in reality, they're there to help you! Not intimidate you!

*bool is_prefix_of(const char *haystack, const char *needle)*

Once we have everything working with our *str_chr*, let's move on to a similar function called *is_prefix_of*. This function is similar to *str_chr* in that it loops through a string to find something, but the difference here is that we are looking for a substring - not just a character. Since both of the inputs are "strings" (`char *`), you will need to check that not only one character matches in the substring (needle), but that every character matches. Return true if the needle is fully contained by the haystack.

Notice that *is_prefix_of* is going to return a `bool`.

Once you are done, make and run your file. See if your function properly identifies prefixes. If not, no worries, go back and try again!

*const char *str_str(const char* haystack, const char *needle)*

Once the function *is_prefix_of* is working, write a new function *str_str* that uses *is_prefix_of* to determine if a word exists anywhere in another word. To check if the search word (needle) is in the haystack, first check to see if it is a prefix of haystack. If needle is not the prefix to haystack, try to see if needle is the prefix of everything but the first letter of haystack. This loop will effectively check for the subword needle in every possible position inside haystack. Make sure to return haystack if you find the subword and NULL if you don't.

Make and run lab3, and see if *str_str* works the way that you intended. Hopefully everything works! If not, as usual, go back and try and find what went wrong and update your code.

char interpolate(const char *format, const char *args[], char *buffer)*

Now using what we have learned about how to manipulate strings we are going to write our own version of *sprintf*(3) (a relative of *printf*(3)) called *interpolate*. Interpolate will return a `char *`, and takes as input a `const char *`, an array of `const char *`, and a final `char *`. The first input (`const char *`) format will contain our format string. This string is what our program will work through to try and come up with an output string. The second input is args and this holds the elements that we will be placing into the new string. The final input is simply our buffer, where we will build everything to return. The rules for our format string are going to be that you will fill in text any time you see `{}`. So, a string that looks like `"Hello {}!"` with and argument of `"Jason"` would return `"Hello Jason"`. As well, we want to allow our format strings to have modifiers. If you just see `{}`, then return exactly what you got as input but if you see `{^}` then make the input uppercase and if you see `{v}` then make the input lowercase. This is a complex problem so it might be useful to break it down into the component parts: identifying format string and modifiers, filling the string in (modified).

Once you have this done, make and run lab3, and see if this feature is working!