COSC 301: Operating Systems
Lab 2: C keeps me warm at night
Due: 13 September 2011

Fall 2011

Contents

1 Synopsis **1**

2 Updates **1**

3 Detailed Description **1**

3.1 C functions 1

3.2 Automated Tools for Building a C Program 2

3.2.1 scons 3

4 Submission **3**

1 Synopsis

Time to jump into some C! In this lab you'll write a few functions to start honing your C skills. The functions all revolve around C string manipulation, which will require you to deal with pointers.

Note: For those who have installed their own Linux virtual machine, you may need to install the C compiler, debugger, and other programs. On Ubuntu, the following command will work to grab everything you need:

```
sudo apt-get -y install gcc gdb scons valgrind manpages-dev
```

2 Updates

None yet.

3 Detailed Description

3.1 C functions

You'll need to write four C functions for this lab, as follows:

Remove white space from a C string: This one is pretty simple. The function should take a C string as a parameter and remove any whitespace characters from the string (spaces, tabs, and newlines). It should return the string without the whitespace characters. (Note: it's ok to do the whitespace removal "in place", so you might end up returning the same pointer as was passed to the function.) The string should "shrink" as whitespace characters are removed, so if you get an input string like `a b c` you should return the string `abc`.

You can either directly compare characters to test whether it's a space, tab, or newline, or you can use the C library function `isspace` to do the test for you. `man isspace` if you want to use the built-in function.

Convert a C string to a Pascal string: The second function should take a C string on input and return a string formatted for the Pascal language. The difference between the two is that C strings have a NULL (`'\0'`) character termination to indicate the end of a string, and in Pascal the first byte of a string holds the length of the string, but there is no termination character.

This function should just convert an input C string to a Pascal-formatted string, returning the Pascal string. Just like the whitespace function, it's ok to do the conversion in-place. (That is, you don't need to necessarily allocate any new memory. You can just use the existing memory allocated to the C string.)

Note that because the length of a string is encoded in a single byte, there's a hard limit to the maximum length of a Pascal string. If the input C string can't be encoded as a Pascal string, your function should return `NULL`.

An example: if we have the C string `"ABC"`, it will be encoded in the following four bytes (recall that the integer value for ASCII `'A'` is 65):

65 66 67 0

The same string would be encoded in Pascal as:

3 65 66 67

Convert a Pascal string to a C string: Similar to the above, except the reverse direction: take a Pascal string on input and return a C formatted string.

"Tokenify" a string into an array of strings: Now for a little tougher function. `tokenify` should take a C string and split it up into space-delimited words. The words should be returned as an array of C strings, with the last element of the array explicitly set to NULL. For this problem, you will need to allocate new chunks of memory using `malloc`. *The array as well as each C string referred to in the array should be newly allocated from the heap.*

For example, if you get the string `go patriots` on input, you should return an array of three elements. The first array element should be a pointer to a C string containing the first word, same for the second element, and the third element should be NULL. (Since there's no built-in way to detect the length of an array, we have to explicitly include a stopping point in the form of NULL as the last element of the array.)

You can either do the tokenization yourself (*i.e.*, find each whitespace delimited word using primitive comparisons), or you can use the C library function `strtok` (which I would recommend doing). `strtok` will do the "hard" work of finding each space-delimited word; it's up to you to put them into an array. The downside to the convenience of using `strtok` is that it can be a bit tricky; you'll want to carefully read the `man` page.

You'll need to create/edit one file this lab: `lab02.c`. Three other files, `lab02.h`, `main.c`, and `SConstruct` are posted on Moodle to help get you started. `lab02.h` is complete: you shouldn't need to make any changes to it. The only changes you'll need (or want) to make are to the `lab02.c` and `main.c` files. In `lab02.c` you'll put all your code to implement the four functions described above. In `main.c` you'll put any additional testing code to call your function in order to make sure that they work as intended. The `SConstruct` file is given to help compile your program; some description of this file is given below.

Since the above functions are mostly straightforward, you can probably do most debugging by employing `printf` statements in strategic locations. You can also use the `gdb` program to step through your program line-by-line (I can help to get you started with that), and the `valgrind` program for ferreting out memory corruption problems (I can also help with that).

3.2 Automated Tools for Building a C Program

There's no grade associated with this part of the lab, but you'll probably want/need it to get your program compiled and running.

You should know by now that `gcc` is the compiler we're using for creating executable binaries from our C source code. For very simple programs, it's easy enough to just type `gcc -o junk -g -Wall junk.c` on the command line to compile a program. When programs get to be larger than a single source file, compiling one-by-one becomes a serious pain.

Fortunately, there are tools out there that can help. All Integrated Development Environments (IDEs) like Eclipse and Apple's XCode have capabilities to build a complex multi-source file project. There are also command-line oriented tools such as `make` [1]. You'll likely encounter `make` at some point in your CS career. It's powerful and widely used, but the configuration files can be a pain since everything must be manually spelled out. In this class, we'll instead use a powerful yet fairly easy to use command-line tool called `scons`.

3.2.1 scons

Instead of using `make`, we're going to use a much more user-friendly program called `scons`. The configuration syntax for `scons` is much simpler than for `make`, and it's pretty easy to specify everything necessary to build a complex program. We'll really just scratch the surface of capabilities in scons in this class. (Note: Google Chrome, among other major software systems, is built using `scons`, so this is not just a "toy" tool we'll be using.)

There is one required configuration file for `scons`, which must be named `SConstruct`. In the very simplest case of a source file named `test.c` from which we want to build a program named `test`, the contents of `SConstruct` can simply be:

```
# just a one-liner!
Program('test', 'test.c')
```

To compile the program, you can say `scons` on the command line (in the same directory in which your `SConstruct` file and `test.c` file are located). If you modify the source code and want to rebuild the program, you just type `scons` again. Example:

```
$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
gcc -o test.o -c test.c
gcc -o test test.o
scons: done building targets.
$
```

If you want to clean things up and remove `.o` (object) files and the executable, you can type `scons -c`.

For a slightly more complex setup to build a program consisting of two source files, and one in which we want to specify different compiler flags, the `SConstruct` file is still quite simple:

```
# builds a program named 'lab02' from two source files, just like
# this lab...
env = Environment()
env.Append(CPPFLAGS=['-g','-Wall'])
env.Program('lab02', ['lab02.c', 'main.c'])
```

(Note that `scons` is written in Python, so some of syntactical elements of `SConstruct` files are built on Python syntax. You can pretty much write any valid Python program code in a `SConstruct` file if you want, too.)

Most of the `SConstruct` files we'll need in this class will be on the level of complexity of the second example: not too hard.

When compiling your code, I recommend that you **always** set your `CPPFLAGS` to include `-g` and `-Wall`. `-g` will instruct the compiler to include debugging symbols (and you'll want these if things go south), and `-Wall` instructs the compiler to emit any warnings if it encounters dubious code. *Pay attention to these warnings!* They can be cryptic to read and understand (I can help), but they usually indicate something unsavory going on in your code. When I compile your programs to grade them, I always look at any compiler warnings first...

4 Submission

Just submit your `lab02.c` file to Moodle. There's no need to upload your `main.c`, `lab02.h` or `SConstruct` files.

[1] See `http://www.gnu.org/software/make/manual/make.html`.