COSC 301: Operating Systems
Lab 1: C keeps me warm at night
Due: 11 September 2012

1 Overview

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.

As part of this lab, you will need to create and submit three files (`lab01.c`, `lab01.h`, and a `Makefile`) and create a `git` repository to store your files.

You are welcome to consult with someone else while developing the code for this lab, but you should work on and submit your own code. If you do collaborate with someone, please make a comment at the top of your code to say who it was.

Also, please do all lab work on Linux, not on MacOS (or another OS) in this course.

2 Detailed Description

First, create a new `git` repository to store your files for this (and possibly other) labs. Please create this on one of the publicly hosted servers (e.g., github.com or bitbucket.org). You can either create one repository for all your work in this class, or a separate repository for each lab. You will need to demo/show me your repo when you're done. Reminder: a handy tutorial on `git` can be found by typing `man gittutorial` at a Linux shell prompt.

2.1 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 do the removal *in place*, and not return anything. The string should "shrink" as whitespace characters are removed, so if you get an input string like `a b c` you should modify the (printable part of the) string to be `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 (unsigned) 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. You should not modify the input C string; return a newly allocated 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. Again, the input string should not be modified in any way.

"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 whitespace-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 the string `go`, the second array element should point to the string `patriots`, 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` (or the thread-safe `strtok_r`). I'd recommend using one of the library functions: `strtok` will do the "hard" work of finding each space-delimited word, leaving you with the task of putting 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 three files this lab: `lab01.c`, `lab01.h`, and a `Makefile` (described next). You `lab01.h` header file can just consist of four function prototypes corresponding to the functions you write in this lab. You are provided with a `main.c`, which can be used to perform some basic testing on your code (and you're welcome to add any additional tests as you write and debug your four functions).

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). We'll learn about `gdb` and `valgrind` soon.

2.2 Automated Tools for Building a C Program

You will need to create a `Makefile` to build (compile) the test program and the source file with the four functions you wrote. In this part of the lab description, we'll go through the basics of `make` and how to construct a `Makefile`.

2.2.1 A short Makefile tutorial

Makefiles are a simple way to organize code compilation[1]. Although there are tools besides `make` that do similar tasks, `make` is (by far) the most widely used tool for building software. This tutorial does not even scratch the surface of what is possible with `make`, but is intended as a starters guide so that you can quickly and easily create your own makefiles for small to medium-sized projects.

Let's start off with the following three files, `hellomake.c`, `hellofunc.c`, and `hellomake.h`, which would represent a typical main program, some functions stored in a separate file, and an include file, respectively. Here are example contents of these files:

Listing 1: hellomake.c

```
// hellomake.c
#include "hellomake.h"

int main() {
  // call a function in another file
```

[1] This tutorial is based on one developed at Colby College.

```
  myPrintHelloMake();

  return(0);
}
```

Listing 2: hellofunc.c

```
// hellofunc.c
#include <stdio.h>

void myPrintHelloMake(void) {
  printf("Hello makefiles!\n");
  return;
}
```

Listing 3: hellomake.h

```
// hellomake.h
void myPrintHelloMake(void);
```

Normally, you would compile this collection of code by executing the following command:
`gcc -o hellomake hellomake.c hellofunc.c -I. -Wall -g`

This compiles the two `.c` files and names the executable `hellomake`. The `-I.` is included so that `gcc` will look in the current directory (`.`) for the include file `hellomake.h`. The `-Wall` is used to turn on any (possibly helpful) compiler warnings. The `-g` is used to include debugging information into the compiled program in case we need to use a symbolic debugger like `gdb`. Without a `Makefile`, the typical approach to the test/modify/debug cycle is to use the up arrow in a terminal to go back to your last compile command so you don't have to type it each time, especially once you've added a few more `.c` files to the mix.

Unfortunately, this approach to compilation has two downfalls. First, if you lose the compile command or switch computers you have to retype it from scratch, which is inefficient at best. Second, if you are only making changes to one `.c` file, recompiling all of them every time is also time-consuming and inefficient. So, lets see how `make` can help address these problems.

The simplest makefile you could create would look something like:

Listing 4: Makefile 1

```
hellomake: hellomake.c hellofunc.c
	gcc -o hellomake hellomake.c hellofunc.c -I. -Wall
```

(Note that the second line starts with a TAB, and not 8 spaces. `Make` will complain if you don't use TABs.)

Makefiles consist of *rules*, which are composed of dependencies and actions (among other items). The first line in the above `makefile` is the start of a rule named `hellomake`, which depends on `hellomake.c` and `hellofunc.c`. As long as the `.c` files exist and haven't changed since the last time you ran `make`, (that's the "depends" part), the action (the second line) will be executed.

If you put this rule into a file called `Makefile` or `makefile` and then type `make` on the command line, the `make` program will read your `Makefile` and execute the compile command as you have written it. Note that `make` with no arguments executes the first rule in the file; typically there are multiple rules in a `Makefile`. Furthermore, by putting the list of files on which the command depends on the first line after the :, make knows that the rule `hellomake` needs to be executed if any of those files change. This is helpful, but we can do even better.

In order to be a bit more efficient, let's try the following:

Listing 5: Makefile 2

```
CC=gcc
CFLAGS=-I. -g -Wall

hellomake: hellomake.o hellofunc.o
	$(CC) -o hellomake hellomake.o hellofunc.o -I.
```

So now we've defined some constants `CC` and `CFLAGS`. It turns out these are special constants that communicate to `make` how we want to compile the files `hellomake.c` and `hellofunc.c`. In particular, the macro `CC` is the C compiler to use, and `CFLAGS` is the list of flags to pass to the compilation command. By putting the object files—`hellomake.o` and `hellofunc.o`—in the dependency list and in the rule, make knows it must first compile the .c

versions individually, and then build the executable `hellomake`. (There's a bit of magic here: since `CC` and `CFLAGS` are "understood" by `make`, it knows how we want to compile a `.c` file into a `.o` file.)

Using this form of `makefile` is sufficient for most small scale projects. However, there is one thing missing: dependency on the include files. If you were to make a change to `hellomake.h`, for example, `make` would not recompile the `.c` files, even though they needed to be. In order to fix this, we need to tell make that all `.c` files depend on certain `.h` files. We can do this by writing a simple rule and adding it to the `makefile`.

Listing 6: Makefile 3

```
CC=gcc
CFLAGS=-I. -g -Wall
DEPS=hellomake.h

hellomake: hellomake.o hellofunc.o
        gcc -o hellomake hellomake.o hellofunc.o -I.

%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)
```

This addition first creates the macro `DEPS`, which is the set of `.h` files on which the `.c` files depend. Then we define a rule that applies to all files ending in the `.o` suffix. The rule says that the `.o` file depends upon the `.c` version of the file and the `.h` files included in the `DEPS` macro. The rule then says that to generate the `.o` file, `make` needs to compile the `.c` file using the compiler defined in the `CC` macro. The `-c` flag says to generate the object file, the `-o $@` says to put the output of the compilation in the file named on the left side of the `:`, the `$<` is the first item in the dependencies list, and the `CFLAGS` macro is defined as above. (The `$`-prefixed variables are pre-defined by the `make` program.)

As a final simplification, let's use the special macros `$@` and `$^`, which are the left and right sides of the :, respectively, to make the overall compilation rule more general. In the example below, all of the include files should be listed as part of the macro `DEPS`, and all of the object files should be listed as part of the macro `OBJ`.

Listing 7: Makefile 4

```
.PHONY: clean
CC=gcc
CFLAGS=-I. -g -Wall
DEPS=hellomake.h
OBJ=hellomake.o hellofunc.o

hellomake: $(OBJ)
        gcc -o $@ $^ $(CFLAGS)

clean:
        rm -f $(OBJ) hellomake

%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)
```

We also added a `clean` rule that removes the object file and compiled executable. To invoke this rule, you need to type `make clean` on the command line. (We also need to add the `.PHONY` directive to tell `make` that the rule `clean` doesn't refer to an actual file; it's a "phony" target.)

Your job for this part of the lab is to create a `Makefile` for this lab program, adapting one of the (better) patterns above. For more information on makefiles and the make function, check out the GNU Make Manual, which will tell you more than you ever wanted to know about make: `http://www.gnu.org/software/make/manual/make.html`.

3 Submission

Submit your `lab01.c`, `lab01.h` and `Makefile` files to Moodle. (There's no need to upload your `main.c`.) Please create a single archive that includes all three files using the UNIX `tar` utility. The following command will create the file `lab01.tgz` that you can upload to Moodle:

```
$ tar cvzf lab01.tgz lab01.c lab01.h Makefile
```