COSC 301: Operating Systems
Lab 1: Intro to C and Makefiles

1 Overview

Time to jump into some C! In this lab you'll write three functions to start honing your C skills. The functions all revolve around C string manipulation, which will require you to deal with pointers. You'll also learn about using Makefiles for compiling your code.

In all our labs this semester, you are welcome and encouraged to work with someone else. If you decide to do "pair programming", you **must** trade off between driver and observer — it is not acceptable for one person to drive or observe all the time. If you do work with someone else, 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). You can either work in the class VM, or on one of the lab machines.

2 Detailed Description

To get some practice with `git`, fork the repository at `https://github.com/jsommers/cosc301_lab01` to create a new repository in your own account. Once you've done that, clone the repo to your own local working space (either your virtual machine, or one of the lab machines). (I'd recommend using the https:// URL for cloning your git repo, unless you have prior experience with git, ssh, and ssh keys.) There is one new file to create and add to the repo (`lab01.c`). Once you add it, you'll commit changes to the file as you make them and as you test your code. Once you're done, you should push all your code up to github and demo your repo to me.

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 three 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.

You'll need to create a file named `lab01.c`, add it to your git repo, and add your three functions to this file. You should not need to modify `lab01.h` at all. The file `main.c` is included in the repo and contains some (very basic) tests for the three functions you should write. You're, of course, welcome to add any additional tests for ensuring your code works correctly. Once you are done writing your functions in `lab01.c`, you should commit any uncommitted changes to git, and push all your code up to github (i.e., `git push -u origin master`)

A `Makefile` is also included in your repository to help compile your code. A detailed tutorial on the `make` program and `Makefiles` is given below. Please be aware that you'll need to create your own `Makefile` for future labs and programming projects.

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, and the `valgrind` program for ferreting out memory corruption problems (I can also help with that). We'll learn more about `gdb` and `valgrind` in a later lab.

2.2 Automated Tools for Building a C Program

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

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

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

  return(0);
}
```

Listing 2: hellofunc.c

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

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

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

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.)

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

For this lab you'll need to (1) demo (show) your github repo to me with all your code (including the Makefile), and (2) submit the name of your repo to Moodle. The repo name you submit should be in the form `https://github.com/username/cosc301_lab01` or `git://github.com/username/cosc301_lab01.git`.