Lab #1: Introduction

Overview

This lab is meant to serve as an introduction to the development environment that you will use for this course and to the C programming language through multiple small exercises. Parts of these exercises are intentionally repetitive to build your "muscle memory" for basic C syntax. In addition, you will use a debugger (`gdb`) to explore the basics of the memory layout of C programs (this is an important illustrative step in understanding pointers).

The Operating Environment

In this course you will develop programs in the Unix environment. You should be familiar with many Unix commands (such as `ls` and `cd`) from previous courses. If you are not, take the time now to read through a Unix tutorial (one is linked from the course website) and *ask questions*.

Editor

There are many different text editors available on the lab machines. Many people prefer to use either `vi` (`vim`) or `emacs`. You are welcome to use either of those (and there are good reasons for doing so), or you may use something else. My recommendation is that you select an editor that you can use through a remote connection (allowing you to work on the department servers from elsewhere) or one that provides file transfer capabilities (over scp), such as Sublime Text.

Some will stress the value of `vi`, noting that it is available on all Unix systems. This is true, but I leave the choice to you. More importantly, I recommend selecting and learning an editor with features that aid in programming.

Exercises

Download the `lab1.tar.gz`[1] file from the course website and extract its contents into a local directory (`tar xzvf lab1.tar.gz`). Within the `lab1` directory you will find a subdirectory for each of the following parts.

Part 1

Just to get started, change into the `lab1/part1` directory. In a new file named `part1.c`, write the infamous `Hello, World!` program. This will require the following:

- Including `stdio.h`.
- Properly defining the `main` function (it can be defined with a `void` argument list for this).
- Using `printf` to print the string.
- Returning 0 to signify normal termination.

Compile and run your program as follows.

```
% gcc -Wall -ansi -pedantic part1.c
% ./a.out
```

The flags (or switches) provided to `gcc` (those that begin with a dash) are used to increase the amount of feedback from the compiler regarding potential issues with the code. `-Wall` enables many warnings, `-ansi` checks the source against the C standard (ISO C90 on the department servers), and `-pedantic` forces warnings demanded by compliance with the standard. *All of your programs, except when otherwise noted, must compile cleanly (i.e., without warnings or errors) with these switches.* You might even consider adding `-Werror` (to turn warnings into errors) to avoid any temptation to let a warning slide.

The generated executable (assuming the source compiles) is stored as `a.out`. If the program fails to compile, then the `a.out`, if present, will remain unchanged from the previous successful compile.

[1]Why a gzipped tar file instead of a zip file? Why not? This is a systems programming course in Unix, so it's worth seeing multiple formats

Part 2

Change into the `lab1/part2` directory. For this part you will write a function that performs a simple calculation and test cases for that function.

This program is split over multiple files. You will notice that some files have a `.c` extension whereas some have a `.h` extension. Those with the `.h` extension are called header files. These files should only contain function prototypes (the name, return type, and argument types for the function), structure definitions (discussed later), symbolic constants, and `extern` declarations. They should not contain any "code" (i.e., they should not contain function definitions).

Perform the following steps.

- In `part2.c`, `#include math.h` in order to use `sqrt` below.
- In `part2.c`, `#include part2.h`. Though this is not strictly required in this case, it will be necessary when using structure definitions and it is common practice to allow the compiler to check function prototypes against their definitions.
- In `part2.c`, define the `calc` function (taking and returning a `double` value) to compute the following.

$$calc(x) = sqrt(x + 37) * 9 - 14$$

- In `part2_tests.c`, write test cases for this function using the `checkit` macros (you can read about them on the course webpage). One such test is already provided for you.
- Compile and run your program as follows. Verify that your tests pass. (Note that on some systems you will also need to explicitly link the math library into your executable. You can do so by adding `-lm` to the end of the `gcc` command line below (after the source file names)).

```
% gcc -Wall -ansi -pedantic part2.c part2_tests.c
% ./a.out
```

Some Details on Directives

You should examine the given header file. You will notice some lines that begin with `#`; these signify preprocessor directives. The C compiler begins with a preprocessing phase during which a number of directives are processed. These directives are used to guide some aspects of compilation. The most common of these is `#include`. This directive says to include the contents of the specified file (a header file) in the current file (almost like a copy-and-paste). Such a feature allows a programmer (you) to organize their program into multiple files where each represents a logical unit of functionality or development.

Alas, one must be careful with `#include` as including the same file multiple times (most often through a chain of one file including another which includes yet another where some ultimately include the same header file) may lead to the same declarations being made multiple times. The C compiler will then complain about redeclarations.

To prevent multiple declarations, the declarations in a header file are guarded with a `#ifndef` ("if not defined"). This directive acts similarly to the if statement in C, but, in this case, is checking to see if a preprocessor variable has been defined (in this case, PART2_H). If it has not been, then the variable is defined and the declarations are compiled. If the variable had been defined previously (i.e., this file was already included before), then the declarations are skipped.

Part 3

Change into the `lab1/part3` directory. For this part you will write a function that performs a summation of the elements in an array of integers. Following the structure of Part 2, define your program across the three files `part3.c`, `part3.h`, and `part3_tests.c`.

You must write a function, `sum`, that takes, as arguments, an array of integers and an integer denoting the length of the array. The function must return the sum of the integers in the array. The function definition be written in `part3.c` with the prototype added to `part3.h`.

In `part3_tests.c`, write test cases for the `sum` function. Be sure to test an "empty" array; you can do so by passing 0 as the length but with an array containing some non-zero number of elements (call it a little white lie; C won't mind).

Compile and run your program as in Part 2 (using the new file names, of course). Verify that your tests pass.

Part 4

Change into the `lab1/part4` directory. For this part you will write a function that creates a lowercase version of a given string. Again following the structure of Part 2, define your program across the three files `part4.c`, `part4.h`, and `part4_tests.c`.

In C, a string is really just an array of characters (`char`) terminated with the `\0` character (the null character).

1. Write the function, `str_lower`, to take two arguments: the original input string and a buffer to store the lowercase version of the input string[2]. Write the `str_lower` function to copy into the buffer the lowercase version of each character in the original string. You should use `tolower` from `ctype.h` to convert a character to lowercase.

2. Write the function, `str_lower_mutate`. This function takes a single string argument and changes the characters to lowercase by modifying the contents of the string itself. As such, the "result" is the original argument. This style is not uncommon in C since strings are mutable, but it can lead to unintended side-effects when there are multiple aliases to the same string.

Split this program over the three files as before and write a few test cases to verify correct functionality. Be sure to properly terminate the new string (in the buffer) with the null character.

Part 5

`make` is a program that is commonly used to automate compiling. More specifically, make executes commands based on a set of rules. These rules are defined as a set of dependencies and a target. If any of the dependencies is "fresher" (the timestamp is more recent) than the target, then the commands for the rule are executed. This is incredibly useful when a program is separated into many compilation units.

Change into the `lab1/part5` directory. Examine the contents of `Makefile` (you can do so by executing `more Makefile`). At the top of the file you will see multiple variable definitions (for, e.g., `CC`, `CFLAGS`, etc.). This is done, as is often the case, to reduce duplication and to simplify making modifications.

Next you will see a number of rules that specify, in essence, how to build a program. One rule, for instance, states that `$(MAIN)`, which was defined to be `example`, depends on `$(OBJS)` (i.e., `example.o` and `fact.o`) and `fact.h`. If `example` does not exist or if any of the dependencies was more recently changed, then the action code will be executed.

Type `make` in the directory that contains the `Makefile` and the source files. Notice which commands were executed. Now type `touch fact.c` (`touch` updates the timestamp on a file). Again, type `make`. By viewing the `Makefile`, you should notice that only those commands for rules that depended on `fact.c` were executed (i.e., `example.o` was not rebuilt).

You will work with `make` for the remainder of the quarter. This small example was only meant to get you started. Do not panic if some of this is unclear; model your future makefiles on this one and learn more as you become more comfortable.

Part 6

In the `lab1/part6/part6.h` header file you will find the definition of a structure to represent a point in two-dimensional space.

Perform the following steps.

- In `part6.h`, add the definition of a structure to represent rectangles in two-dimensional space. The rectangle should be stored as the top-left and bottom-right points.

- In `part6.c`, define the `is_a_square` function. This function must take a rectangle and return true (non-zero) if that rectangle is, in fact, a square. The function returns false, otherwise.

[2]The buffer is needed because an array (a string) declared locally to `str_lower` would be stored on the run-time stack and, thus, could not be safely returned. Alternately, one could dynamically allocate the buffer, but that comes later.

- In `part6.h`, add a function prototype for the `is_a_square` function.
- In `part6_tests.c`, add at least two test cases for your `is_a_square` function.

  Note that the testing macros cannot directly compare structures (i.e., there is no `checkit_struct` since structs are user-defined). Though not necessary for this exercise, the provided files demonstrate tests on a function (`create_point`) that returns a struct by comparing each component of the resulting struct against its expected value.
- Write a `Makefile` (modeled off of the one provided for the previous part) to build the program.
- `make` and run your tests.

Part 7

Write a simple program that takes command-line arguments and prints to the screen each such argument that begins with a '-' character. For instance, the following is the expected behavior.

```
% ./a.out here are -some command-line -arguments with some -dashes
-some
-arguments
-dashes
```

Part 8: Memory Layout Diagram

For this part of the lab you will explore some of the memory layout of the run-time stack by probing the running program in the GNU debugger (`gdb`). Though the goal at present is exploration, `gdb` will save you time later when you are trying to find and fix bugs (you will experiment with `gdb` in such a capacity in the next lab).

To use the debugger to its fullest extent, you will need to compile your program with the `-g` switch. In the `lab1/part8` directory, modify the given `Makefile` to add the `-g` switch to the `CFLAGS` variable and then build the program.

The simplest way to begin `gdb` is to specify the name of the program that you would like to debug. The following abbreviated transcript shows the commands that you should execute (it does not include their output).

Start `gdb` and execute this sequence of commands. Pay particular attention to the responses from `gdb` (the important points for this lab are denoted by a `<---` comment from me). You will be asked to draw a diagram of the run-time stack based on the addresses you see during this run.

```
% gdb layout
(gdb) break main
(gdb) run
(gdb) s
(gdb) s
(gdb) s                  <--- should be on line 19, just before call to function_one
(gdb) print &first       <--- print the address of the variable (i.e., where in memory
                              this variable is stored; on the stack)
$1 = (int *) 0xXXXXXXXX  <--- not actual output, keep track of the real number
                              if you prefer decimal format, use 'print /d  &first' above
(gdb) print &second
$2 = (int *) 0xXXXXXXXX  <--- keep track of the real number
                              Notice the difference in these two addresses.
(gdb) print &p
$3 = (int **) 0xXXXXXXXX <--- keep track of the real number
                              This is the location/address of variable p,
                              as above, not its value.
(gdb) print p            <--- this is p's value, the address it points to, which is?
(gdb) print *p           <--- this is the value at the address where p points
```

```
(gdb) s                  <--- step into function_one
(gdb) s                  <--- step through initialization
(gdb) print function_one_local
(gdb) print &function_one_local  <--- keep track of this result
(gdb) s
(gdb) s                  <--- step into function_two
(gdb) print &function_two_local  <--- what happened here?
(gdb) print function_two_local   <--- what happened here?
```

Draw the segment of the run-time stack that corresponds to this execution (just before the program terminates). In particular, label appropriate cells with their addresses and, if available, their names. For those variables with values, list the values in the cells. For the pointer, `p`, draw an arrow to the cell it "points to" (i.e., the cell with address equal to `p`'s stored value).

To get you started, `first` and `second` should label cells near each other (with the addresses given during the gdb run). `first` has the value 111 and `second` has the value 222.

Demonstration

After you have completed the above parts, call me over so that I can record your completion of this lab. You can (and should) continue with next part while waiting for or after your demonstration.

`man` Pages

You should learn to consult the man(ual) pages when you're looking for additional information about commands and standard C library functions. Getting used to reading man pages does take time (they can be quite dense), but they are incredibly useful.

At the shell prompt, execute the following commands (read as much about each as you are interested in, but do not skip the pages for `getchar` and `putchar`).

```
Basic Commands

% man ls
% man mkdir
% man cp
% man mv
% man rm

Programs used in this lab

% man tar
% man more
% man make
% man touch
% man gcc
% man gdb
% man man

A few C library functions

% man getchar
% man putchar
% man printf
% man strcpy
```