EE312 Exercise 1

The purposes of this assignment are to

1. Acquaint you with the usage of Linux command line to edit, compile and run programs.
2. Acquaint you with the submission procedures for EE312.

You will log in to the EE machines mentioned in the syllabus page using some software such as PuTTY if your laptop runs Windows or a Terminal if you use Mac. For this project you are encouraged to edit the files and compile them using command-line tools. In the future we'll be using CLion.

If you are completely unfamiliar with command line tools

`cd <directory>` ← changes directory

`ls` ← lists current directory

`pwd` ← prints the name of the current directory (i.e., "where am I?")

`gcc Exercise1.c` ← compiles Exercise1.c and creates a.out

`mv a.out program` ←renames "a.out" to be "program"

`./a.out` ← runs the a.out executable file

`./program` ← runs the program executable file

FAQ: How do I run gedit on the LRC machine?

A: On a Mac, you would install XQuartz and then use 'ssh -Y …' to log in to the LRC machine. The -Y option (also called the -Y flag) allows you to cast your gedit window to your local display.

Instructions:

You should find an *Exercise1* folder on Canvas under files. You should find three files inside Exercise1. This document (Exercise1.pdf), a C- language program file (Exercise1.c) and a Makefile (Makefile).

Open Exercise1.c in an editor (e.g., `gedit Exercise1.c` if you are running with graphics or using `vi Exercise1.c`). Then edit the comments at the top of the file to correctly specify your name and to correctly describe the program's output. To compile program use `gcc Exercise1.c`. To run the program use `./a.out`. If you need to modify the program, do so, but add a comment where you make the modification.

Optional Steps:

Acquaint yourself with the various steps of the C compile. There are four steps, Pre-processing, Compiling, Assembling and Linking. You can do any or all of these steps using the gcc program. By default, gcc does all four steps. You can tell gcc to do just one step by providing command-line flags to gcc.

Preprocessor

Run "gcc –E Exercise1.c". The preprocessor handles all the #include and #define (and related) commands in your file. Many preprocessors also remove comments. This command sends its output directly to the screen. If you want to save the output to a file, you can use the –o command flag with gcc. Be careful not to overwrite your Exercise1.c file. Run "gcc –E Exercise1.c –o preproc.c". This will create preproc.c in the local directory. Note that preproc.c is much larger than the original Exercise1.c. I bet you didn't realize there was that much crap involved with just #include <stdio.h>, huh? Also note that you can compile preproc.c (it's a C program). Try "gcc preproc.c" and then run a.out.

OOPS!

OK, so you messed up somehow. Don't panic. To erase one file, do "rm <file>" for example, "rm Exercise1.c". To erase everything (don't do this!) do "rm *". Just be really careful when you try to erase everything, 'cause if you're accidentally in the wrong directory, it will erase EVERYTHING in that directory. It might be safest to make 1 working directory and another directory to store your best work, input file, etc.

Compiler

To run just the compiler, you use "gcc –S Exercise1.c", or if you want to use the already preprocessed file, "gcc –S preproc.c". This will create a .s file with the same name. The .s file is the assembly language produced from your c file. Open Exercise1.s with your editor and find _main. Note that the _ character is prepended to all function names in C. That's an old convention. Other than that, and the fact that the assembly language may be weird to you, it should look like an assembly listing with labels, instructions, etc. By the way, if you have a 64-bit Linux installation and you want 32-bit Intel assembly, use the –m32 flag, "gcc –S –m32 Exercise1.c"

Assembler

To run just the assembler, use the –c flag, for example, "gcc –c Exercise1.s". Or if you want to run the compiler and assembler, "gcc –c preproc.c", or if you want to run the preprocessor, the compiler and the assembler, use "gcc –c Exercise1.c". Believe it or not, most serious C developers compile their programs using this last option, where they run all the phases of the gcc compiler up to, but not including, the last. When you run gcc with the –c flag, you will create a .o file. This is a binary file containing the instructions (in binary) and the global variables used by your program. But, the .o file is not a complete program, it has to be linked first.

Linker

The final step in creating an executable program is to link it. The link step searches your program for any functions you didn't actually write (like printf), and then searches through the standard C library for those functions. The purpose of the linker is to combine several object files together and to create a single cohesive binary program. To run the linker you simply use gcc with no flags. If we already have the .o (known as an "object file") from the assembly output in the previous step, then we can specify that .o file on the command line, "gcc Exercise1.o"

Knowing the steps that the compiler performs when it builds your program is actually very important. Different sorts of errors occur at different points along the way. It's very unusual to encounter an error during the Assembler phase, but errors can occur at any other stage. A preprocessor error occurs when you have a #include statement and you have a mistake in the name of the file, e.g,. #include <stdIO.h> (upper case IO won't work). We'll make lots of compiler errors this term – syntax mistakes like leaving a semi- colon off, or too few -- in our programs. But we'll also end up making linker errors. A linker error occurs when we call a function that doesn't exist, or we use a global variable that the linker can find. It's important to recognize a linker error because the way to fix that mistake is to add the correct library to the project, or to make sure that we're including all the write .o files when we build. By the end of the semester, this should all become quite natural. Hopefully, seeing the individual steps helps lay the foundation.

Makefile

Look at the Makefile provided. It has 3 targets: a.out, Exercise1.o, and Exercise1.s. Ask your TA for assistance or look on the web for GNU make documentation.

Submission

Zip up your Exercise1.c and your Makefile and submit it to Canvas. Call your zip file

```
<Your EID>_Exercise1.zip.
```

If you make a mistake, you can resubmit it over the other one; ignore the resulting name change of your file, if any.