Lab 1 (10 points)  
 August 30, 2017

Laboratory Assignment #1 —  
Getting Started



**CS-2303, System Programming Concepts, A-term 2017**

Due: at 11:59 pm on the day of your lab session

### Abstract

Write two *C* programs and one *C++* program, and then compile and run them in a Linux shell on CCC systems. Learn about the make utility and makefiles.

### Outcomes

After successfully completing this assignment, you should be able to:–

* Develop a single-file C or C++ program on a Linux platform using a command shell and editor
* Read in and print out numbers using formatting strings in C, and do ordinary calculations using floating point arithmetic.
* Understand enough about makefiles to carry out Programming Assignment #2.

### Before Starting

Read Chapters 1 and 2 of *The C Programming Language*, 2nd edition, by Kernighan and Ritchie.[[1]](#footnote-1)

Part 1 of this assignment is to compile and run the “Hello, world!” program of page 7 of K&R and also to compile and run the *C++* version of the same program from §1.1 of *Absolute C++*, by Walter Savitch.[[2]](#footnote-2)

Part 2 of this assignment is a simple arithmetic computation in *C*. *Before coming to the lab, write out your proposed solution to this part!*

Part 3 of this assignment will introduce the make utility and makefiles.

Note: If at all possible, you should carry out this lab on your course virtual machine. If your virtual machine is not ready or not usable, you may carry out this lab assignment using the CCC Linux system via a remote connection.

### Getting Started

1. Sign the attendance sheet.
2. *If you have the course virtual machine installed on your laptop computer, and if you have it with you today,* start the virtual machine and log in.

When the desktop appears, bring up a Linux *command shell*. You can do this by typing CTRL-ALT-T, or by clicking on the icon on the taskbar, or using the menu icon in the upper-left corner of the screen.

If successful, skip forward to Part 1: Hello, World in C and C++ below.

1. *If you are unable to use your course virtual machine today*, you will use XWin32 on the lab computer. If XWin32 is already running, there will be a blue *X-monitor* icon in the system tray at the lower right hand corner of your screen — similar to this:– . If XWin32 is not running you must start it on your computer by clicking:–

Start → All Programs → Utilities → XWin32 9.0 → X-Win32

XWin32 is a terminal application for Windows computers; it allows Windows users to connect to Linux servers on a local network or via the Internet. X applications running on those servers will be displayed onto the Windows desktop. There are two reasons why we run XWin32 when connecting to Linux:–

* to enable copy/paste between Linux windows and other windows, and
* to allow an incoming connection to be accepted. It is this feature that allows you to run emacs, kwrite, and other programs on the CCC servers so that their windows show up on the desktop of the lab computer.

1. Log onto the Linux system using your WPI username and password:–

* Double-click the PuTTY icon (or start PuTTY from the Start menu)
* Type ccc.wpi.edu in the connection window and click OK
* Enter your CCC Linux username followed by return, followed by your password

### Part 1: Hello, World in *C* and *C++*

1. Create a directory and subdirectory called cs2303/lab1; make lab1 your working directory.

Note: If you have never used a command shell or a command prompt before — or if you are unsure of what it means to do this step — please refer to the Appendix of this document.

1. Using emacs, gedit, **atom**, **vim**, or your favorite editor,[[3]](#footnote-3) type out the “Hello, world” program exactly as it appears in K&R. Call your resulting program helloWorld.c.[[4]](#footnote-4) You invoke the editor on a file by typing the name of the editor, followed by a space, then the name of the file you want to create or modify.
2. Rather than continually exiting and restarting the editor, you can edit in one window and type commands in another. An easy way to do this is to open a second copy of the shell window, then move to the subdirectory where you are putting the files for this assignment.
3. In the Linux shell, compile the program with the command

gcc -Wall helloWorld.c

The -Wall “switch” means “Warnings: all”; it instructs the gcc compiler to display all warning messages. If you find errors or warnings, make changes and recompile. If you have difficulty, consult a classmate or one of the TAs. When you can compile with no errors or warnings, run the program with the Linux command

./a.out

1. Experiment with inducing compilation errors. For example, remove the semicolon at the end of line starting with printf and recompile. Note that error messages in *C* usually appear one or more lines *after* the actual error. Moreover, a single error can cause the appearance of many false errors after it.

Also experiment with warnings. Remove the zero after the return. Compile using the –Wall switch as in Step 2 and then again *without* the –Wall the switch. In this course, all programs *must* compile free of warnings (i.e., using the –Wall the switch) unless an exception is allowed by the Professor.

1. Repeat the same steps for a *C++* program called helloWorld.cpp. A copy of this program is also found at the end of this document. This time, compile the program with the command

g++ -Wall helloWorld.cpp

1. Submit both programs to *Canvas*. From a browser, visit the following URL,

<https://canvas.wpi.edu>

login in, navigate to the course “CS2303-A17” and to the assignment *Lab1*. Submit the files helloWorld.c and helloworld.cpp as part of assignment *Lab1*:–

Be sure to submit your program by 11:59 PM on the date of your lab session in order to get credit for this part of the lab.

### Part 2: A simple numerical calculation

Write a *C* program called triangle.c to prompt the user for the *x*- and *y-*coordinates of the three corners of a triangle, calculate and print the lengths of the three sides, calculate and print the circumference of the triangle, and calculate and print the area of the triangle. The following mathematical formulas may be used in these calculations:–

The formula for determining the length *lAB* of the line between points (*xA*, *yA*) and (*xB*, *yB*) is

.

If *lAB*, *lBC*, and *lCA* are the lengths of the three sides of a triangle, then the area of the triangle is



where *s* is one-half the circumference of the triangle – i.e., *s* = ½ × (*lAB* + *lBC* + *lCA*).

The following illustrates a sample of the program output and input. The program prompts are shown as bold and do *not* end with new-line characters, while the user responses are shown as nonbold and *do* end with new-line characters. All result lines end with new-line characters.

Enter the x- and y-coordinates of point A:- 1.05 -2  
Enter the x- and y-coordinates of B:- 1.115e+2 21.1  
Enter the x- and y-coordinates of point C:- -25 -3.14159

Length of AB is 112.840  
Length of BC is 138.636  
Length of CA is 26.075  
Circumference is 277.551

Area is 237.833

The function sqrt() is declared in the interface <math.h>. It returns the square root of its argument. The argument of sqrt() must any non-negative numerical value, and the result of sqrt() is of type double. If the argument is negative, sqrt() fails with an error and the result is undefined. (The program may or may not crash.)

Because you are invoking the sqrt() function, you will need to link the math library to your code. Therefore, the gcc command needs the –lm switch, like this:–

gcc -Wall triangle.c –lm [[5]](#footnote-5)

The printf() function is declared in the interface <stdio.h>. It takes the following form:–

printf("string in double quotes", arg1, arg2, arg3, …)

The string, which is enclosed in double quotes, contains zero or more *conversion specifiers*. Each conversion specifier begins with a '%' character and specifies how to convert the corresponding argument to printed characters. The *ith* conversion specifier corresponds to the *ith* argument following the string. For example, a conversion specifier of '%f' says to treat the corresponding argument as a floating point number and print it with the default precision – e.g., *3.14159*. See Table 7-1 of K&R for more details.

The scanf() function is declared in the interface <stdio.h>, and takes the following form:–

scanf("string in double quotes", &arg1, &arg2, …)

The string, enclosed in double quotes, contains one or more *conversion specifiers*. For this assignment, two conversion specifiers are sufficient for each input request. As with printf(), each conversion specifier begins with a '%' character and specifies how to convert the input characters into a numerical or other value. The *ith* conversion specifier corresponds to the *ith* argument following the string, which *must* be the name of a variable and which *must be preceded* by a '&'character. For example, a conversion specifier of '%f' says to treat the input string as a floating point number with optional sign and optional exponent and store it in the variable named by the corresponding argument. More details about scanf() can be found in §7.4 of K&R.

There are no restrictions on the input data; that is, the user may enter any real or integer values for the *x-* and ­*y­-*coordinates of each of the three points of the triangle. You should not define any functions in the solution for this program other than the main() function. You should *not* use any loops or conditional statements.

Submit this assignment using *Canvas* system as above. From a browser, submit the file triangle.c as part of assignment *Lab1*.

*If you have trouble with this lab, ask for help right away. You may ask your classmates, other students, the TAs, the Professor, or anyone else who can help you.*

### Part 3: Introducing Makefiles

The remainder of this lab introduces an important tool — the make command. *If you do not have time to finish this part, please finish it on your own time prior to next week.*

The make command in Linux and Unix allows you to easily maintain and keep your files up to date when you are working on a project that is split across several .c files and their associated .h files. You will find your multi-file programs easier to manage if you keep all of the files for one project in a separate directory, along with a makefile (as described below), to manage the compilation of those files.[[6]](#footnote-6)

A file named makefile or Makefile (note: no extension!) is the usual input file for a tool called make, which can be invoked from the command line of a Linux shell. The purpose of make is to help you build, update, and maintain a collection of related program files. It does this by figuring out which files have changed since you last built or rebuilt your programs. The makefile specifies dependencies among files, so that make can figure out which files have to be recompiled as a result of any of your edits or other changes.

*A properly designed* makefile *is a normal part of every non-trivial programming project in Linux or Unix, both at WPI and in the professional world.* It allows someone else — a grader, a boss, a colleague, or a user — to rebuild your system on another platform by simply typing the command make. Using the instructions and dependencies encoded in the makefile, make takes care of it from there, and the result should be an up-to-date, executable copy of the program, system, or application.

Moreover, the *Eclipse* integrated development environment — which will be introduced next week in the lab assignment — uses make as its tool for compiling and linking programs. Therefore, this part of this lab is essential for getting started with next week’s lab.

#### Getting started with makefiles

Download the file Lab1Files.zip file from Canvas (it is under Assignments / Lab 1) and unzip it into your directory.

You should now have a folder containing the files intarray.c, intarray.h, sinewave.c, and makefile. You can use command **cd Lab1Files** to move into that directory.

#### What’s in a Makefile?

The collection of files needed to build a system or application usually includes header files (.h files), source files (.c files, .cpp files, or source code files for the language you are using), compiled files (.o files), and sometimes other kinds of files. Many of these depend on each other. For example, consider the program downloaded for this week’s lab. The final product is an executable file named sinewave, which is created by linking together two other compiled files, sinewave.o and intarray.o. These two compiled files are created by compiling sinewave.c and intarray.c , respectively. Moreover, both of these .c files include the header file named intarray.h. So, one of the dependencies expressed in the makefile is:–

if sinewave.c or intarray.h changes, then sinewave.o must be regenerated by the compiler command:–

gcc –g –Wall –c sinewave.c

To see this dependency, open up the file named makefile in a text editor.

Near the bottom of the file you’ll find these two lines:–

sinewave.o: intarray.h sinewave.c  
 gcc $(CFLAGS) -Wall -c sinewave.c

The first of these is called a *target line*, which begins with a file name followed by a colon. The file is called the *target*. After the colon is a list of zero or more names of files that the target depends on. If the target is missing or older than any of its dependencies, then it is considered “out of date” and must be rebuilt. The make tool uses target lines and last modification dates to determine which targets must be rebuilt. It does this recursively, so that if one of the files upon which the target depends is itself out of date (or missing), make causes that file to be rebuilt first, etc.

Note: **make** depends upon a consistent and reliable time-of-day and date clock in order to determine whether one file is “newer” than another. If you are using more than one computer and their clocks are not in sync, modification dates on files could be inconsistent with each other, and **make** could produce unexpected results.

After the target line, there is a series of commands that tell exactly how to rebuild the target. For the case of sinewave.o, only one gcc command is needed to compile the file. Note that this command includes the –c flag to tell gcc that it should only compile the specified file(s) and not attempt to link them into an executable program yet.

The gcc command also includes the notation $(CFLAGS). This is not a compiler switch but rather a variable to make. It lets you control from the command line which compiler switches you wish to apply to all of the files compiled by make. The makefile also contains the following line near its beginning:–

CFLAGS = –g -Wall

This defines the variable CFLAGS and causes it to default to the text string "–g -Wall" (without the quotes). This means that the programs should be compiled for debugging (–g) and that all warnings are displayed. We will see below how to override this default.

The –Wall flag conforms to a requirement of this course that all warnings must be displayed; this flags is also shown in the makefile as part of CFLAGS.

Note: **makefiles** have a peculiar requirement — command lines, such as the **gcc** command, *must* each begin with exactly one *tab* character, not with a sequence of blanks!

Continuing with the theme of dependencies among files, the executable file sinewave is created by linking together the object files intarray.o and sinewave.o. If either of these two object files should change, then sinewave also needs to be recreated. Here is the appropriate target line and command from our makefile:–

sinewave: intarray.o sinewave.o  
 gcc $(CFLAGS) -Wall sinewave.o intarray.o -o sinewave -lm

This target line says that if either sinewave.o or intarray.o is newer than the target sinewave (or if the target sinewave does not exist), then make must cause the sinewave program to be regenerated with the gcc command shown. The –o switch specifies that the output program is to be named sinewave, and that the two input files (sinewave.o and intarray.o) are to be linked with the “math” library, which happens to be named libm.a. The reason is that this program uses a precompiled mathematical function to evaluate the *sine* function sin().[[7]](#footnote-7)

#### Practice Using Make

There are two simple ways to use the make facility to automatically regenerate your files. The first approach builds a specific file. For example, suppose you want to regenerate intarray.o. Then you can use the make command shown here. Type this command in now:–

make intarray.o

The make command will find the dependency information in the makefile. It sees that intarray.o depends on two other files, so it will first ensure that those files are present and regenerate them if necessary. In this example, the two files intarray.h and intarray.c are necessary for generating intarray.o. These two files are present, so the make command proceeds to generate intarray.o, using the gcc command that is specified in the makefile. When the gcc command is executed, it is displayed in your command shell or on your display screen. It this case, it would be:–

gcc –g –Wall –c intarray.c

After this make command finishes, you should list the files in your directory, where you will find the object file intarray.o is now present.

Note: You can specify as many targets as you wish to the make command.

Now trying using the make command without specifying a file, like this:–

make

Without a specified file, the make command will regenerate the first target that it finds in makefile. Try this now, and you will see that the executable file sinewave is regenerated, since sinewave is the first target in makefile.[[8]](#footnote-8) During the process of regenerating the sinewave file, make discovers that sinewave depends on intarray.o and on sinewave.o. But the file sinewave.o is not present. This means that before make can build sinewave, it must first regenerate sinewave.o. Only then it can proceed to regenerate the executable file sinewave. On the screen you will see the two steps displayed:–

gcc –g –Wall –c sinewave.c  
gcc –g –Wall sinewave.o intarray.o –o sinewave -lm

Next, edit the file intarray.h in some trivial way — for example, by adding or changing a comment. Now type the command

make

again and you will see that it rebuilds everything. Can you explain why?

Finally, if you wish to turn off the –g switch, you may use the following command:–

make CFLAGS=

This simply sets the variable CFLAGS to the null string, thereby disabling compilation for the debugger. You may also change the compiler flags, for example, by

make CFLAGS=–O or make CFLAGS=–O2

This tells the compiler to optimize the compiled code for speed. You can consult the man page for the gcc command for other compiler switches, which may also be specified in the CFLAGS variable.

To get a better feeling about how dependencies work, edit your makefile to remove the dependency that sinewave.o depends upon intarray.h. Again make a trivial edit to intarray.h, and then type make. You will notice that it did *not* rebuild sinewave.o. This, of course, is an error, so change it back!

#### Cleaning up

Notice the last two lines of the makefile. These say

clean:  
 rm -f sinewave.o intarray.o sinewave

This is a target line to “make” the target called clean. *Clean* is not a file but rather an artificial target, sometimes called a “phony” target. You can see that clean does not have any dependencies, but it does have one command line, namely a command to remove the .o files and the original target file sinewave. This is common practice in system programming — to provide a way of cleaning up intermediate files (in this case the .o files) needed to build a system, leaving only the original files.

Type the command

make clean

and then list your directory. You will see that sinewave and the .o files have disappeared. Now type

make

and you will see that they have been rebuilt.

### “Hello, World” programs

In case you forgot, here is a copy of the “Hello, World” program in *C*:–

/\* HelloWorld.c -- type your name here (and on every program   
 you write!) \*/

#include <stdio.h>

int main(void) {  
 printf("Hello, World!\n");  
 return 0;  
} // int main(void)

Likewise, here is the “Hello, World” program in *C++* —

// helloworld.cpp -- type your name here (and on every  
// program you write!)

#include <iostream>

int main(void) {  
 std::cout << "Hello World from C++!"<< std::endl;  
 return 0;

} // int main(void)

## Getting credit for this Lab

To earn credit for this lab, you must submit the following to *Canvas* under the assignment *Lab1*:–

* HelloWorld.c
* HelloWorld.cpp
* Triangle.c
* The output (i.e., the file a.out) from running make clean followed by make CFLAGS=–O2.

Appendix — Command Shell

A *command shell[[9]](#footnote-9)* (or *command prompt* in Windows terminology) is a simple text-oriented interface for interacting with many kinds of computers. When the shell is ready to receive a command, it types a *command prompt* in the command window, and then it waits for typed input from the user. The language of the command shell is a simple imperative language, and commands typically respond by printing one or more lines of textual information before allowing the shell to issue its next prompt.

The following illustrates the general form of a command:–

% commandName operand1 operand2 operand3 ...

In this line, the '%' character is the command prompt, which is printed by the shell.[[10]](#footnote-10) The string of text typed by the user begins with the name of the command and is followed by zero or more operands. The user terminates the command by typing a newline character (i.e., the RETURN or ENTER key). Most command shells are intelligent enough to recognize backspace as meaning “undo the previous keystrokes.”

Some useful commands are

% ls

List the contents of the current (i.e., working) directory. (*Directory* is equivalent to *folder* in graphical environments.)

% mkdir cs2303

Make a new directory and name it cs2303.

% cd cs2303

Change the working directory (i.e., current directory) to be cs2303.

% cd

Change the working directory to be the user’s home directory.

% ls cs2303

List the contents of the directory cs2303.

% ls –l cs2303

List the contents of the directory cs2303 in “long form” — i.e., showing not only the names of files and directories within cs2303 but also their sizes, modification dates, and permissions. Note the –l (the letter ell), which is a *switch* of the command. Switches are generally indicated by a minus sign '– ' and they usually precede the principal operands of the command; they control the detailed behavior of the command.

In systems like Linux, Windows, or Macintosh, there are literally thousands of commands for all sorts of purposes. A particularly useful command is

% man commandName

Display the page or pages of the manual describing the command named commandName. This display may be many pages, and the man command allows you to scroll up and down through them.

#### System commands and user commands

A very important concept is that each command is actually a program stored in a file whose name is the name of the command. The program is a fully-formed program written in *C*, *C++*, *Java*, or some other language. It has a main() function and (probably) many other functions. The command shell picks apart the operands of the command line and stores them in the arguments of the main() function before invoking that function.[[11]](#footnote-11)

The command shell has a list of places in which it looks to find the files for the commands. That list of places is called the PATH, and it can be displayed in Linux or Macintosh by the command

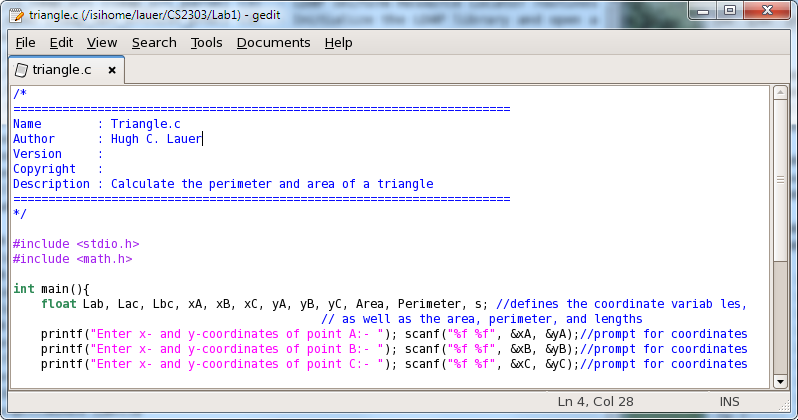
% echo $PATH

As a result of many years of experience, the user’s current directory is *not* in the PATH. This helps to avoid the accidental, unintended execution of programs in the current directory. Therefore, to execute a specific command in the current directory — for example, a.out or helloWorld from part 1 — you need to explicitly prefix the name of the current directory in front of the command. An alias for the current directory is simply '.', and an alias for the parent or containing directory of the current directory is '..'. Therefore, to run the helloWorld or a.out program, you need to type one of the following to the command shell:–

% ./helloWorld  
% ./a.out

#### Creating and editing a program

The choice of text editor for programming is a highly personal matter, not unlike the choice of one’s clothing style. There are dozens of text editors available to command shell users, and all have their advantages and disadvantages. If you have never used one or do not have a preference, the Professor recommends gedit. This has a familiar looking point, click, and type interface, and it can be configured to be useful for *C* and *C++* editing. Here is a sample screen:–



You can start this editor with the command

% gedit filename.c &

where filename.c is replaced with the name of the file you wish to edit. This will bring up a new window, independent of the command shell. The '&' symbol at the end of the line means “run this command separately from the shell, and let the shell continue to accept and run other commands.”[[12]](#footnote-12)

#### Learning the command shell language

There are many, many resources for learning the commands of *Linux*, *Macintosh*, or *Windows*. Here are a few:–

<http://community.linuxmint.com/tutorial/view/100>  
<http://www.ee.surrey.ac.uk/Teaching/Unix/>  
<http://www.linfo.org/command_line_lesson_1.html>  
<http://www.youtube.com/watch?v=imDrXVVHowU> (a video)

Although we will be using the command line interface only a little in this course, you will use it intensely in the Operating System course and probably in a number of other contexts.

1. Throughout the course, this book will be referred to as K&R. [↑](#footnote-ref-1)
2. Copies of both programs may be found at the end of this document; feel free to copy and paste them. [↑](#footnote-ref-2)
3. If you don’t have a favorite editor, see the section entitled *Creating and editing a program* at the end of this document. [↑](#footnote-ref-3)
4. A copy of the “Hello World” program appears at the end of this document; you may copy and paste it. [↑](#footnote-ref-4)
5. More information about –lm and the math library can be found in the section *Practice Using Make* on page 6 below. [↑](#footnote-ref-5)
6. Very large projects such operating system kernels, compilers, and sophisticated web browsers typically comprise a hierarchy of directories, along with a hierarchy of **makefiles** to support them. [↑](#footnote-ref-6)
7. In general, the **gcc** or **g++** switch **–lxyz** means “link with the file named **libxyz.a**. [↑](#footnote-ref-7)
8. Note that at the end of the **makefile**, there is a target **all** with a dependency on **sinewave**. This is needed for *Lab 2*. You may ignore it for this lab. [↑](#footnote-ref-8)
9. The etymology of the word *shell* in this context is obscure. The best that the Professor can come up with is that the *shell* is a program that surrounds the commands that you run with a “hard” protective environment. The protection is such that even if the command or program goes haywire, the shell continues to work and provides the user a way of running other commands to eventually recover. [↑](#footnote-ref-9)
10. Some command shells will prefix the user’s name and/or the name of the working directory as part of the command prompt. For example, **[lauer@CCCWORK4 ~/CodingExamples]$** in the CCC Linux systems. [↑](#footnote-ref-10)
11. We will learn more about command line arguments in Programming Assignment #2. [↑](#footnote-ref-11)
12. To configure **gedit** for editing *C* programs, select the **Preferences** subcommand from the **Edit** menu, and select the **Syntax Highlighting** option. In this window, select **C** in the **Highlight mode** box. Later in the course, you should change this selection to **C++**. [↑](#footnote-ref-12)