# CPE 460 Laboratory 1: A Crash Course in C

Department of Computer Engineering
Yarmouk University

Spring 2017

## 1 Purpose

Welcome to CPE 460! This lab will give you a brief introduction to and Linux operating system and its development tools. You will be asked to perform a few exercises and read various manuals about some function calls. There are lots of great C and Linux resources available online. Manual pages, Google[1] and StackOverFlow[2] are your best friends in this lab. In addition, you can always ask your TA and instructor for clarifications.

## 2 Submission

There is **no** lab report for this lab, but you are expected to review, understand, and ask questions about this material during the lab period. Future labs will build on this material.

## 3 Logging In

Linux is a multi-user operating system – it allows multiple users to have accounts and to be logged in at one time. TA will assign you a username and a default password to login. Linux presents a desktop environment that enables users to easily use and configure their computers. Once you feel marginally comfortable with the desktop, open a terminal window by going to the top menu `Applications` then `Accessories` and finally `Terminal`. You can add the `Terminal` program to the Desktop by right-clicking on the `Terminal` item in the previous menu and selecting `Add Launcher to Desktop`.

## 4 Command Line

The command line in Linux is just another program, usually called a *shell*. The shell allows you to run other programs or scripts, and is generally useful for managing your computer once you know which commands to use. Linux distributions ship with several shell types (e.g., `bash`, `korn`, `bourne`, etc.)[3]. Ubuntu comes with `bash` as the default shell. When you start a terminal, the first thing you will see is the prompt (often called the command line). As its name would suggest, it is prompting you to enter a command. It should look something like this:

```
[username@host currentdirectory]$
```

We type the commands after the prompt (`$`).

[1] `http://www.google.com`
[2] `http://www.stackoverflow.com`
[3] You can find more on these different type of shells and their differences on:`https://en.wikipedia.org/wiki/Unix_shell`

## 5 `gedit` Text Editor

If you aren't already proficient with a Unix editor, you should pick one to become proficient with. The editor `gedit` is a useful editor. There are several other editors available, many with more features. More seasoned programmers tend to use `emacs` or `vi`, which are console-based editors.
To run the `gedit` editor:

```
$ gedit somefile &
```

## 6 GCC Compiler

A compiler converts *source code* into *object code* or *executable code*. The GNU Compiler Collection (gcc) is included with Linux. Several versions of the compiler (C, C++, Objective C, Fortran, Java and CHILL) are integrated; this is why we use the name "GNU Compiler Collection." GCC can also refer to the "GNU C Compiler," which is the `gcc` program on the command line.

## 7 Experiments

Execute the C programs given in the following experiments. Observe and interpret the results.

### 7.1 Files, Directories, Listing, and Paths

#### 7.1.1 File & Directory Creation

Let us make a new directory named `460-lab1` to contain your work for this lab:

```
$ mkdir 460-lab1
```

Navigate/go to the directory:

```
$ cd 460-lab1
```

Edit/create a new file named `practice.txt`:

```
$ gedit practice.txt
```

A window should open for you to type in. You can create files without an editor using the command `touch filename`. You can refer to `man touch` for more info.
It is always helpful to display the line number along with lines of code so you can fix compilation errors and debug your code more easily. To display line numbers: go to `Edit` menu, then `Preferences`, and finally check the checkbox for `Display Line Number`, then hit `Close`.

#### 7.1.2 Foreground and Background Processes/Jobs

Go back to the command line. Notice anything odd? The prompt didn't reappear. That's because it's waiting for `gedit` to finish. We'd like to continue typing commands at the same time, so let's fix this:

`gedit` is currently in the *foreground*. On the command line, type `Ctrl-Z`. You've just suspended `gedit` and placed it in the `background`. You should have a prompt again. However, because `gedit` is suspended, we can't use it until it returns to the *foreground* or we start running it in the *background*.

Type `bg gedit`, `gedit` will now be running in the *background* while we continue to type commands. If we wanted to return it to the *foreground*, we could have typed `fg gedit`. To list a list of jobs in the background, use the `jobs` command.

*Note*: you can avoid this whole mess in the future by starting `gedit` in the background. Just type an `&` after the command:

```
$ gedit somefile &
```

Now that we have an editor running, type something into the window and save it.

### 7.1.3 Directory Listing

Back on the command line, type the following:

```
$ ls
```

The file `practice.txt` should be listed. Now try:

```
$ ls -a
```

You should see two additional entries – "`.`" and "`..`". `.` is a reference to the current directory, and `..` is a reference to the parent directory. Type:

```
$ cd ..
$ ls
```

You should see the `460-lab1` directory in the listing, since you're now in the parent directory.

### 7.1.4 Absolute & Relative Paths

You need to know two more things about directories: *absolute* and *relative* paths.

An *absolute path* is a path that starts with "`/`". Because it starts with the root of the file system, it will always refer to the same file no matter what your current directory is.

A *relative path* is a path that starts with any other character. It is relative to the current directory, and will refer to different files depending on the current directory.

Type `pwd` to see the current absolute path. You should see something like `/home/username`, which indicates you're in the `username` directory under the `home` directory. This is where to find your account in Linux. It's the default current directory when you login. You can always go back to it by typing:

```
$ cd
```

Now try using an absolute path to switch directories. Type (replacing username with your own username):

```
$ cd /home/username/460-lab1
```

You should now be in your `460-lab1` directory. To confirm this, type

```
$ pwd
```

To return to your home directory, type:

```
$ cd
```

Now switch to the directory again using a relative path:

```
$ cd 460-lab1
```

Again, use `pwd` to verify that you're in the subdirectory. So far there's no difference, right? There will be occasions when you will want to specify the file or directory you want to work with unambiguously by using an absolute path. You can use relative and absolute paths to describe files and directories in many commands. For example, if you're in your home directory, the following two commands should be equivalent:

```
$ ls /
$ ls ../..
```

**Why does this work?** Since the first command uses an absolute path (it starts with `/`), then it will always return the same result no matter where you are. If you're in the directory `/home/username`, then `../..` resolves to the parent of the parent of the current directory, or `/`. If you were in `/home/username/460-lab1` and typed `ls ../..`, you would see the directory listing for `/home` instead.

### 7.1.5 More Commands

Here's a list of commands you might find helpful. Try a few out, but be careful with the ones that delete files! You don't have permission to delete anything other than your own files, but there isn't a recycling bin to recover your work from!

- `mkdir` – make a directory
- `rmdir` – remove a directory
- `cd` – change current directory
- `ls` – lists files and directories
- `mv` – move/rename a file/directory
- `rm` – remove a file
- `cp` – copy a file
- `less` – view a file (use arrow keys or space to scroll). Use `q` to quit.
- `man` – display the manual for a subject or command. Use `q` to quit.

**Can you find the proper command line for performing each of the following operations?**

1. Copy all files from directory `dir1` into an existing directory `dir2`.

   ______________________________

2. Move files from one directory into an existing directory.

   ______________________________

3. Delete all files that ends with `.txt` in the current directory.

   ______________________________

### 7.1.6 General Linux Tips

- Linux has two clipboards to allow more convenient copy/paste while keeping the features of standard cut/copy/paste. When you select text in the terminal or any application, it is copied to the "selection" clipboard; use the middle mouse button (wheel button) to paste into an application. The second clipboard is usually accessed through `Ctrl-C` and similar shortcuts; however, these do not work in the terminal as they are sent to the program in the terminal.
- You can use the tab key to finish commands and paths. Hit tab twice to see a list of potential matches.
- You can use the arrow keys to look through recently typed commands. In bash, `Ctrl-R` will search your command history for the text you type.

## 7.2 `printf`, `scanf`, and Command-Line Options

In this exercise, we will write a program to read user input from keyboard and output to screen. We will also know how to intepreta the user arguments passed to the program from command line.

### 7.2.1 Example Program

First, create a directory `exercise1` under directory `460-lab1`, by typing:

```
$ mkdir exercise1
```

Second, create a source file named `test.c`, by typing:

```
$ gedit test.c &
```

Third, write the following program:

```
#include <stdio.h>

int main(int argc, char** argv) {
    if(argc == 1){
        printf("Enter your name:");
        char name[100];
        scanf("%s", name);
        printf("Welcome [%s] to CPE 460!\n", name);
    } else{
        printf("User has sent me %d arguments, they are:\n", argc);
        for(int i = 0; i < argc; i++){
            printf("Argument[%d]: %s\n", i, argv[i]);
        }
    }
    return 0;
}
```

If your code was typed in correctly, you can compile and link `test.c` into an executable by typing this sequence of commands:

```
$ gcc -c test.c
$ gcc -o test test.o
```

When you use the `-c` option in gcc, the C files are compiled into object files (`.o`) that can be linked together into an executable. If you got an error similar to the following: `error: 'for' loop initial declarations are only allowed in C99 mode`. That means that you are using an old C compiler and you need to compile your code using the flag `-std=c99` as follows:

```
$ gcc -c -std=c99 test.c
$ gcc -o test test.o
```

Remember to save your work and always re-compile and build your work after each edit to the source files.

It is also possible to compile `test.c` file at one time with the command

```
$ gcc -o test test.c
```

Compiling the C files to object files is less memory intensive when you have hundreds of files in a project; in addition, since compiling from C to object is much more expensive than linking object files, compiling to object files will allow you to only recompile files that have changed, rather than an entire project. The `make` utility automates this process. `make` will be discussed in the next Lab.

### 7.2.2 Running without Options/Arguments

To run the program without any arguments, type:

```
$ ./test
```

*Note:* why "`./`"? Recall the discussion of directory navigation in the previous section? `.` is a reference to the current directory. Since the shell needs to find the file you want to run, you have to specify the complete path (or have the file in a directory stored in the `PATH` environment variable). You can think of `./` as a shortcut for the absolute path up to the current directory. Don't forget this – you'll need it to run programs for the rest of the semester.

The previous program when run it produces the following output:

```
Enter your name: Mansaf
Welcome [Mansaf] to CPE 460!
```

**Observations**

- Function `printf` and `scanf` is equivalent to `std::cout` and `std::cin` in `C++`, repectively.
- Function `printf` is used to print to the "standard output," which, for a PC, is typically the screen.
- Function `scanf` is used for a formatted read from the "standard input," which is typically the keyboard.

**`printf` Function Call**

```
int printf(const char *Format, ...);
```

`printf` takes a formatting string `Format` and a variable number of extra arguments, determined by the formatting string, as indicated by the `...` notation. The keyword `const` means that `printf` cannot change the string `Format`.
The formatting strings consist of plain text, the special character `\n` that prints a newline, and directives of the form `%s` and `%d` indicates that `printf` will be looking for a corresponding variable in the argument list to insert into the output. A non-exhaustive list of directives is given here:

- `%d` Print an int. Corresponding argument should be an int.
- `%f` Print a float.
- `%c` Print a character according to the ASCII table. Argument should be char.
- `%s` Print a string. Argument should be a pointer to a char (first element of a string).
- `%X` Print an unsigned int as a hex number.

Please refer to `printf` man page for further info through the command `man printf`.

**`scanf` Function Call**

```
int scanf(const char *Format, ...);
```

Arguments to `scanf` consist of a formatting string and pointers to variables where the input should be stored. Typically the formatting string consists of directives like `%d`, `%f`, etc., separated by whitespace. The directives are similar to those for `printf`. For each directive, `scanf` expects to see a **pointer** to a variable of that type in the argument list. A very common mistake is the following:

```
int i;
scanf("%d",i); // WRONG! We need a pointer to the variable.
scanf("%d",&i); // RIGHT.

char message[20];
scanf("%s", &message); // WRONG! message is already a pointer to the start of array.
scanf("%s", message); // RIGHT.
```

The pointer allows `scanf` to put the input into the right place in memory. Please refer to `scanf` man page for further info.

### 7.2.3 Exercises

Answer the following questions:

- Can you find what would be the output string if you enter a space-separated input to the program instead of `Mansaf` and why?
- Can you modify the program to accept space-separated string of size 250 and print it out fully? (Hine: `man gets`)

### 7.2.4 Running with Command Line Options/Arguments

Command line options are options that are passed to a program when it is initiated. For example, to get `ls` to display the size of files as well as their names the `-l` option is passed to it. (e.g. `ls {l /home/me`). In this example, `/home/me` is also a command line option which specifies the desired directory for which the contents should be listed. From within a C program, command line options are handled using the parameters `argc` and `argv` which are passed into main.

```
int main(int argc, char **argv)
```

To run the program with arguments, type:

```
$ ./test mansaf maqloba makmoora
```

The previous program when run it produces the following output:

```
User has sent me 4 arguments, they are:
Argument[0]: ./test
Argument[1]: mansaf
Argument[2]: maqloba
Argument[3]: makmoora
```

### 7.2.5 Exercises

Answer the following questions:

- What does `argc` refer to?
- What does `argv` contains?
- Write a C program `sum.c` that sum of the user-provided numbers through the command line arguments to the program. *Hint:* You can convert a string representing integer to its integer value using the `atoi` function call. Refer to `man atoi` for details.