# CSE 451/851 Programming Assignment 0

Introduction to C, Linux/Unix, Makefile, CSE Linux Servers

## 1 Introduction

In this programming assignment you will be introduced to

- C
- Linux/Unix POSIX library
- manpages
- Makefiles
- how to upload your code to cse-linux-01.unl.edu.

## 2 Preliminary Information and References for Study

Before you embark on writing some code here are some places to begin learning about these topics. This will serve as a good reference to use when you need help or get stuck.

### 2.1 C Programming

There are lots of resources online to learn C. C and C++ are very similar, and C++ is a superset of C. You need to use C, not C++, for our programming assignments. Here are a few resources for C:

- Beej's guide
- `https://www.tutorialspoint.com/cprogramming/index.htm`
- `https://www.w3schools.in/c-tutorial/`
- C for Java Programmers

### 2.2 Linux/Unix and POSIX Library

There are two main methods for accessing Linux/Unix OS functions: System V and Posix. System V was developed in 1983 at AT&T and many descendants of that system are still based on it. Its competition was primarily the BSD variants of Unix with a competing standard for the underlying API. However, in 1988 the IEEE Computer Society decided to standardize this creating the Portable Operating System Interface (POSIX) standard for maintaining compatibility between Unix variants.

Most operating systems are now either POSIX compliant, or *mostly* POSIX compliant. This includes Mac, Windows, Android, VxWorks, iOS, and most distributions of Linux. In this class I strongly suggest you use the POSIX API as it will make your code interoperable between OSs. I also think many of the POSIX functions are easier to use.

Here is a list of POSIX C library functions. These are also available as man pages in your shell.

### 2.3 man (Manual) Pages

`man` (or manual) pages are key to your programming assignments. `man` pages can be accessed within a *nix shell via the command `man <topic>`. Many topics relating to the Unix kernel, C programming, and even some abstract topics are available. A pretty good list exists here or here.

### 2.4 Makefile

`make` is a tool to build a software project. It uses a file called `Makefile` or `makefile` to do this. You will use `make` in this class to build your software projects. You will need to build your own `Makefile` for some programming assignments, while in others it will be given. Either way you should have an idea of how to write a `Makefile`, how to use it, and do basic debugging on it. Some good resources are here:

- GNU Makefile Overview
- http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- https://opensource.com/article/18/8/what-how-makefile
- http://mrbook.org/blog/tutorials/make/
- Or just google "Makefile tutorial" and you will get lots of great hits.

### 2.5 Uploading to the CSE Linux Server

There are numerous ways to get your code onto the cse-linux-01@unl.edu server and compile and execute your code. The easiest way is to use a ssh software, like PuTTY, for Windows or use the terminal window if you are using Mac or Linux. Make sure to use your UNL credentials (Username and Password) to access the server.

To Upload your files from your local machine to the server, you can use software like WinSCP or FileZilla for Windows or FileZilla for Mac. There are other software's that you can use so choose the one you want.

## 3 Exercises

In these exercises we will develop versions of some of the Unix tools (cat, echo) and use the program to introduce the POSIX API's and Unix concepts. But let's start with doing a review of C.

### 3.1 Exercise 1: Review of C

Please review usage of pointers in C, dynamic memory allocation (`malloc` and `free`). Just google "pointers in C" to get lots of great hits.

### 3.2 Exercise 2: Hello World

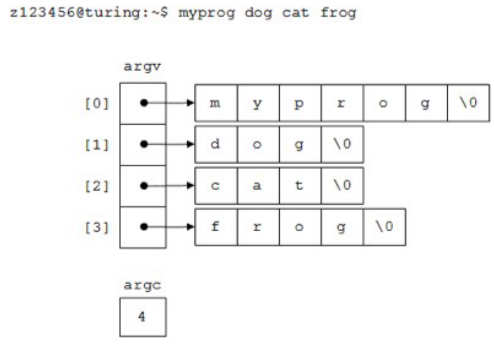
For this exercise write a program in C that prints "Hello World" to the terminal. Use a Makefile you wrote to compile it. If you're not working on the CSE Linux server, upload it to the server and compile it and run it there to ensure you can test your code there.

### 3.3 Exercise 3: Echo

In this exercise write a C program that will echo back to the terminal the input given to the program. You can learn more about `echo` by typing `man echo` into a terminal. This exercise will help you learn to work with command line arguments. Remember the function prototype for C is:

```
int main(int argc, char* argv[]); or int main(int argc, char** argv);
```

where `argc` is the number of arguments and `argv` is a pointer to an array of pointers (remember C has no notion of strings). Each member of the array points to the characters in each argument to the program. A representation of this can be seen here:



You can read more about it by googling "argc and argv in C." At this point you just need to parse the arguments and print them back to the terminal. A `for` loop should do the job nicely. Use a Makefile you wrote to compile your program.

### 3.4 Exercise 4: Cat

Here you will write a C program that will concatenate and print the contents of the files presented as command line arguments. Read the man page on `cat` to learn more. This will help familiarize you with opening, closing, and reading files.

The steps to do this exercise are:

1. read in and parse the arguments.
2. for each argument (let's assume no bad arguments)
    a. open the file.
    b. read the contents of the file.
    c. print contents of file to terminal
    d. close file

There are roughly two main strategies for doing File I/O here.

1. Use the C functions in `stdio.h`. These are C functions that utilize the underlying system libraries. As such they're perhaps a little cleaner and easier to use.
2. Use the underlying system POSIX API. These are very similar to the C library though perhaps a bit more "raw."

Typically, the C library functions look like `fopen`, `fclose`, etc. and work with "streams" represented by `FILE`. The POSIX system functions look like `open`, `close`, etc. and work with a file descriptor (which is really just an `int`). These types are nearly identical and can be converted to the other easily.

Feel free to use either one you want, but it is generally recognized that working with "streams" is easier. Use a Makefile you wrote to compile your code.

### 3.5 Exercise 5: Upload one Exercise to CSE Linux Server and Test

Finally, you should go through the process of uploading and testing your code on the CSE Linux server. Place each of your programs in a separate directory (along with its Makefile) Upload them and any test cases to the CSE Linux server. Then login to the CSE Linux server and build and test the programs you uploaded to ensure it performs the same as on your machine.

If your code doesn't compile or doesn't work the same on the CSE Linux server, *you need to figure out why* since we will grade everything on the CSE Linux server. If you can't figure out why, you might need to do all your development directly on the CSE Linux server.

## 4 What to submit

After testing your code on the CSE Linux server, make sure to compress the three directories into a zip file named Assignment0_lastNameFirstName. Please replace the last name and first name with your name. Then submit the zip file to Assignment 0 on the course canvas page.

## 5 Useful Notes to Increase Understanding

There are a couple important Unix concepts worth understanding in preparation for our programming assignments. In the man pages, some of them use the term "stream," to discuss I/O while others use "descriptor." Read this for an explanation.

This also brings us to another important concept. One of the core philosophies in Unix, is that everything is a file (or treated like a file to be more specific). Files, directories, devices (e.g., terminal, storage, processor) are all treated as files. What this tie back to is that, we can open, read and write to the devices, console, directory etc. the same way as a file. Given this abstraction, it is often easier to visualize the data as just a stream of bytes. This article talks more about it.

This concept extends to the default inputs and outputs. Whenever Unix creates a process, it opens three files (or streams) by default.

1. stdin (file descriptor value = `STDIN_FILENO` = 0)
2. stdout (file descriptor value = `STDOUT_FILENO` = 1)
3. stderr (file descriptor value = `STDERR_FILENO` = 2)

These streams handles to default input and output. stdout and stderr are usually mapped to the terminal (or console). This is the reason any print statement appears on the terminal. One of the nice parts about treating things as "streams" is that we don't need to worry about where the data is going. So, if for example, we decide that we don't want the output to appear on the terminal, we can remap the output descriptor to a file instead. Similarly, we could read from a file instead of terminal by remapping stdin to something else.

From the perspective of the process, it just needs to know the descriptor it should read and write to. The pipe command `|`, and the file redirector commands `>`, `<` work in this way. You should get used to using those on the command line and understand how they work.