CSCE 451/851 Programming Assignment 0

Preparing for the Programming Assignments

Introduction

In this programming assignment you will be introduced to

- C/C++
- Linux/Unix POSIX library
- Manual pages or `man` pages
- Makefiles
- Development options and how to upload your code to the CSE servers.
- Understanding file streams

You will then develop two programs that emulate a few basic shell functions.

Submission procedure

Do the following:

1. Create a directory called `<UNL username>_pa0` (replace `<UNL username>` with your UNL username).
2. Place your solutions to Exercises 2, 3, and 4 below in the directory
3. Add a `Makefile` with targets of
   - "myecho" – should compile your `myecho` program into a binary called "myecho"
   - "mycat" – should compile your `mycat` program into a binary called "mycat"
   - "all" – should compile both programs
4. Zip the directory `<UNL username>_pa0` and submit.

Use CSE web handin to hand in your assignment. Submit a single zip file, `<UNL username>_pa0.zip` (e.g., `jdoe2_pa0.zip`).

1 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.

1.1 C/C++ Programming

There are lots of resources online to learn C. C and C++ are very similar, though C++ is a superset of C. You can use either C or C++ for any of the programming assignments. However, for most of them there is a choice that does make it a bit easier. Most students use C++.

1.1.1 Resources for C

- Beej's guide. Beej's guide also has lots of good information Unix interprocess communication (which you'll need)
- `https://www.tutorialspoint.com/cprogramming/index.htm`
- `https://www.w3schools.in/c-tutorial/`
- C for Java Programmers

1.1.2 Resources for C++

- cplusplus – good tutorial and reference
- learncpp – good tutorial
- Very extensive reference
- Learning C++ when you know Python

1.2 Main Differences Between C/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. If you're keen on C++, you can use Google to get many tutorials on using objects in C++.

The primary difference between C and C++ is the use of objects in C++. C has no concept of "objects" and therefore everything is procedural. I don't perceive that one is easier than the other in this course. The only slight advantage C++ may have is it's native handling of strings.

1.3 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. It's 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. **You will need to pay attention to this when you look at "man" pages to make sure you're not accidentally looking at a System V page.** The function should indicate whether or not it is POSIX compliant.

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

1.4 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 the Unix kernel, C programming, and even some abstract topics are available. A pretty good list exists here or here.

1.5 Makefile

`make` is a tool to build a software project. It's similar to Ant for Java (if you're familiar with it). It uses a file named `Makefile` or `makefile` to do this. You will use `make` in this class to build your software projects. You will need to write your own `Makefile` for some programming assignments, while for 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`
- Or just google "Makefile tutorial" and you will get lots of great hits.

If you plan to use your own machine for development, one way to make your life significantly easier is to setup Makefile targets that can automatically:

1. upload your code to the CSE servers
2. compile your code
3. run your code against test cases

This allows you to build that check into your development process.

1.6 Development Workflow and Uploading to CSE Servers

You can choose how you want to develop your programs. Generally, the two main options are:

1. Develop locally on your own machine, then check correctness by uploading your project to the CSE servers and recompiling and running tests there.

2. Develop and test directly on the CSE machines/servers. If you do this, please be very careful using VSCode. VSCode liberally uses its cache space and it quickly fills up your disk space allocation and you will get a "Disk Quota Exceeded" message from the sysadmins.

Most students choose the first option. In that case, there are numerous ways to get your code onto the CSE servers and compile and execute your code. Check out the UNL CSE FAQ for options.

Perhaps the most direct way is to:

1. use `scp` to "secure copy" your code to your home directory.
   - something like `scp <files> <CSEusername>@cse.unl.edu:<directory path on CSE server>`
2. use `ssh` to login to the CSE servers
   - something like `ssh <CSEusername>@cse.unl.edu`
3. navigate to wherever you put your code
4. compile and execute your code just as you did on your local machine

If you tire of writing your password every time you login to the CSE servers you can setup a private/public key pair. Instructions for doing that can be found by Googling "ssh key setup" or similar. If you do this **make sure you never share, show, or divulge in any way your private key**. You put the public key on the server, and the private key remains on your laptop.

You can (somewhat) automate this process by building the above commands into your Makefile and writing Make targets to test on the CSE servers.

1.7 Unix File Streams

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. This means 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 are 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.

2 Program Specifications

In these exercises we will develop simplified versions of some of the Unix tools (`cat`, `echo`) and use the program to introduce the POSIX API and some basic Unix concepts. For these, we are looking for simple implementations. You don't need to implement any switches or complex functionality (see Evaluation section below for more detail).

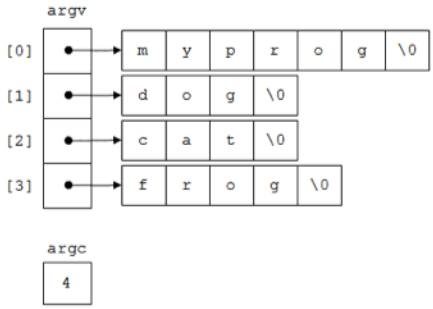
2.1 Program 1: `myecho`

In this exercise write a C/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 `main` in C/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

```
z123456@turing:~$ myprog dog cat frog
```



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.

2.2 Program 2: `mycat`

Here you will write a C/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 the file I/O:

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.

Testing

Finally, you should go through the process of uploading and testing your code on the CSE servers. Upload your code, any Makefiles, and any test cases to the CSE servers. Then login to the CSE servers and build and test the code 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 servers *you need to figure out why* since we will grade everything on the CSE servers. If you can't figure out why, you might need to do all your development directly on the CSE servers.

To test your code, examine the evaluation criteria below and test accordingly.

Evaluation

Your program will be graded according to the following rubric:

Command	Output Matches Shell Command
`./myecho my PA0 is working`	20
`./myecho "working with quotes"`	20
`./mycat <file1>`	20
`./mycat <file2>`	20
`./myecho 'echo and cat' > foo.txt;./mycat foo.txt`	20
Total = 100 points	

The output expected in each command is the same output as the built-in `echo` and `cat` shell commands. So when in doubt, check what they do to see what your program should do. `<file1>` and `<file2>` are simply random files that have **text** in them that we will use. So to test your code generate a few different files with only text in them and make sure the output of your program matches that of the CSE server `cat` command.