CSC209 Notes

Software Tools and Systems Programming

https://github.com/icprplshelp/

Last updated January 30, 2023

These are the notes I've taken for the course CSC209. Notes from PCRS will not show up here for now. As this class is inverted, this document might not contain as much content as you hope it would.

1 Introduction

This course is about:

- Software tools
  - Using the command line and not clicking on UIs (apparently linux addicts hate them)
- Systems programming
  - No, this course isn't learning to program
  - It's about the pieces of C we haven't seen yet
  - And the systems part of programming in C (the file system, the notion of processes, and communications over the network)

1.1 Unix Principles

- Unix is different fundamentally than an IDE or an application with a UI.
  - An IDE gives you a button for everything.
- Unix has simple tiny tools that can be combined to do interesting tasks.
  - A lot of them are programs that do one things, maybe with a few variations, but all the same thing
  - We have a way to connect these tools together in different combinations to do more interesting tasks
- To do everything together, **they work with plain text files.**

- **None of the UNIX tools (should) require human-interactive input** (e.g. `input()` in python)
  - Why? So we can automate things and put commands into scripts, so they can run without having us involved
  - We want a simple output format, so the output format is the simple format that the next tool can take the input
  - I/O streams
    * In Java, we have `System.out.println()` (standard output) and `System.err.print()`
    * For every process, we have a `stdin` (most of the time, it's a keyboard).

1.2 Commands

UNIX is short on vowels and you'll rarely see them. Commands are short.

- `cd` for change directory
- `ls` is for list all the commands on the path or the current directory
  - a.k.a. all your files... probably I'm wrong on this
  - `ls -F` lists them with an extra slash, listing all files with a `/` at the end if it is a directory. For example: `file1.txt file2.txt folder/`
  - `-lF` gives us the long listing (and combines the effects of `-F` – more verbose and looks like file explorer but with text)
- `cat <file>` (stands for concatenate) shows me the files content...
  - It takes the argument `<file>` and uses it as the filename it should open and read for standard input.
  - `cat` takes what comes in standard input and push it to standard output
  - Then the shell displays the standard output in the window for us.
  - Multiple arguments? `cat` stands for concatenate, so it concats all the inputs and sends the output to standard output.
    `cat <file> <file> <file>` puts file three times into the standard output.

  - `cat document document > double_document` – you know what this does. That's exactly how you concat stuff into a new file.

- `sort` sorts each line in lexical order and puts it in standard output.
- `wc document` (gives me the word count. It gives me `LINE_COUNT WORD_COUNT CHARACTERS` (words are tokens?)
- `sed "<regex>"` (stream editor)
  - Applies the regex to ALL lines and sends it into the standard output.
  - Example: `sed 's/, /XXX/' document`
    * Search for `,` and replace it with `XXX` (similar to find and replace)
  - Let's do this again: `sed 's/\(.*\), \(.*\)/\2 \1/' document`... good luck figuring that out (basically transforms `LastName, FirstName` to `FirstName LastName`
- Piping: the pipe symbol – whatever process is on the LEFT, the standard output from THE LEFT becomes the standard input on the RIGHT. For example:
  - `sed 's/\(.*\), \(.*\)/\2 \1/' document | sort`
    * We don't sort by first name, so maybe sort first?
  - `sort document | sed 's/\(.*\), \(.*\)/\2 \1/'`
  - Now, this command, if `document` were a text file filled with `LastName, FirstNames` on each line, it sorts by last name first THEN does the swap.
- The `>` symbol redirects the output of a command (what is printed / put in system out) to a file, if I could rewrite it.
- `man <command>` gives us the manual page for the command. For example, `man sort` gives us a help page.
- `cut -d " " -f 1` has the same effect as `line.split(' ')[0]` for every line in the file.
- `unique`... you know this... **NOT**.
  - Filters out **ADJACENT** matching lines.
  - If you really want to get rid of duplicates, SORT FIRST

> UNIX is user-friendly; it's just choosy about its friends.
>
> Become a friend of UNIX.

1.2.1 The Grep Command

> Grep: search for (a string of characters) using grep.

It's in the dictionary. It means **global regular expression and print.** The idea of gripping is you're searching an RE on a file and you're sending in standard output the lines that match.

For example, find all the lines that match this RE and print it.

And do you want to grep but get the numbers of occurrences? Pipe it into `wc`.

1.2.2 Shell

The interface between me and the OS. When we type in the shell, the `$` is a shell prompt (we can redefine that if we want) and will probably already be there.

```
$ wc hello.c
```

- the text `wc hello.c` is a command for the cell
- `wc` is the name of an executable file or program to run
  - `wc` is in the `PATH` variables
- The remaining text `hello.c` is an argument to the program

The shell

- targets the exectuable
- passes in the arguments

Know the tools!! Memorize their name, but not the options. You can look them up using the `man` page. Experts underestimate how much they use them, so it will boost your speed if you memorize them.

- `head`, `tail`, `cd`, `mkdir`, `ls`, `cp` (copy), `mv` (move/rename), `rm`, `diff`, `comm`, `cut`, `cat`, `wc`, `grep`.

Get used to them, because they will replace the use of UIs (I will never get used to them).

2 How To Work With Git

2.1 Adding

When making a file that you'll need to submit:

1. Create the new file. Anywhere you want, but in your working directory.
2. Run `git add <file_name>`. This adds it to git.

2.2 Committing and Pushing

When you make changes to a file:

1. Run `git add <file_name>`, or run `git add .` to add everything in the directory except for `gitignored` files (because the csc209 file structure is a bit shaky, don't do this).
2. Run `git commit -m ""`
    3. If you don't include `-m ""`, git will force you to input a commit message in vim. And vim is a nightmare to navigate.
3. Run `git push`.

2.3 Removing a file

Added a file by accident? Here's how to remove it

1. Run `git rm <file_name>`
2. Commit and push like normal.

3 File System

File systems are trees... or are they? The `/` is the root directory, and inside the `/` directory, there are a bunch of other directories, and so on.

3.1 File System Hierarchy

- Everything starts in the root directory, name `/`
- A directory is a file that contiains directory entires
- A directory entry maps a file name to an `inode`
  - A data structure that contains information about the file (size, owner, access/modified/creation time, perms, and so on).
  - Includes direct pointers to file blocks.

3.1.1 LS Outputs

- The leftmost from the table generated from `ls -lF` is the permissions, except for the first character, which is the type. hence, the table is:
- type, permissions
- owner
- group
- size
- date last modified
- file name, ends with `*` if executable and `/` if directory. If you see these symbols, they aren't part of the file name

3.1.2 Permissions

```
-rwxr-xr-x
```

- (ignore the first char, that's for something else. afterwards)
- first 3 chars are for the owner
- next three are for the user group associated with the owner, but not necessarily the owner
- last three is for everyone else

File permissions:

- read, write, executes
- For directories:
  - Read:
    * Can run `ls` on dir
  - write:
    * Can create or delete files in dir
  - execute:
    * Can pass through directory even without the read perms

3.2 chmod

```
chmod <mode> <path>
```

Change permissions. Look at the slides for how to run the commands.

Two approaches:

- Using octals (more concise but harder to learn; check the slides). Learn octal to binary, and the other way around.
  - This completely overwrites the permissions and does not preserve anything.
- Or the more readable approach
  - `chmod <u/g/o><+/-><r/w/x> ...`
  - Adds or takes away permissions

  - `chmod go-x ...` takes away `x` perms from both `g` and `o` – these categories of users

Use `*` to target ALL files (except for `-`) (run for all files). Similar but not exactly REs; they are called `globbing`

You need to know both, because you'll have to accommodate people who are addicted with either approach

3.3 Globbing

A little like regular expressions but different

- `*` matches any no. of any character (equivalent to `.*` in RE)
- `?` matches any one character (equiv to `.?` in RE)
- `[list of chars]`
- `[1-5]` or `[a-z]` or `[a-xz]`

That's the basic stuff. You should probably memorize that

To be used if you want to mention a file… but not targeting all files. And you can also mention multiple files at once:

```
chmod o-r day.txt e1.pdf emptydir
```

3.4 Running a Program

In Python, we would write a program in a plain text file normally named ending in `.py`. We run it using the command `python3 hello.py` (normally) from the command line.

- program being run is `python3` and takes the argument `hello.py`

For a C program: we have a file `hello.c`. We compile it with

```
$ gcc -Wall -g -std=gnu99 -o hello hello.c
```

See the other arguments? You MUST use them. Get a macro or something

The arguments:

- `-Wall` (show me all the warnings – if you have warnings, usually something is wrong and also you'll lose marks. It is not a style warning, and it is on something you've done that is likely wrong)
- `-g` (when I build the executable, leave in the information inside the executable so that we can run the debugger. otherwise the file will be kinda obsfucated)
- `-st=gnu99` tells us the version
- `-o` (The next argument is the name you should store the executable. If you do not put that in, the executable defaults and goes out to `a.out`. Apparently, lots of people had trouble with it)
- `hello.c` (The source file you want to target)

Turns it into an executable. We'll be using `gcc` (make sure it works on `teach.cs` but I can use another C compiler if I want when practicing).

Run `./hello` to execute the file. The `./` states the directory: `.` means the current directory I'm in (cwd), and `/` is "IN THE DIRECTORY".

Some users can do this without `./` – it may due to a configuration in your path. Probably not a good idea, as you don't want a file named `ls` in any of your directories. I wouldn't.

3.5 Paths: Absolute vs. relative

Absolute vs. relative paths

ABSOLUTE: all the way from the root to the path: `/u/.../.../hello.c`

RELATIVE: relative to `cwd` or `pwd` or `..`. For instance, `hello.c` if that file is in my present working directory.

4 Arrays

The takeaway: don’t use pointers for array access, and the other way around. It gets confusing.

When an array of size 4 is declared, it sets aside space, saying “I can’t put anything else here.” All we know is that `A` is an array that starts there. It is up to you, the programmer, to stay in the space you allocated.

Beware that this is an array of pointers:

```c
char *result[2];
// result[0] is type char*
// result[1] is type char* as well
```

5 Strings in C

Strings in C are character arrays with a special character at the end to denote the end of a string.

5.1 Copying and Concatting

PATTERN TO COPY STRINGS

```c
strncpy(to_copy_to, to_copy, strlen(to_copy) + 1);
```

PATTERN FOR CONCATING STRINGS

```c
strncat(s1, s3, sizeof(s1) - strlen(s1) - 1);
```

5.2 String Variables vs. String Literals

String **variables** are defined using `char str[] = "hello world";`. The string data is stored in the stack. I could also use `malloc` with this, and it will be considered a string variable.

String **literals** are defined using `char *str = "hello world";`. The memory address is put in somewhere that is read-only.

- You may reassign what `str` points to afterwards. You **do NOT need to free them, so do not worry about memory leaks. These are managed by the system.**

> In C, a string variable is a variable that holds a reference to an array of characters, whereas a string literal is a sequence of characters enclosed in double quotes, such as "hello world". When a string literal is used in a program, it is stored in a read-only memory location, and a pointer to that location is used to refer to the string. Attempting to modify a string literal will result in a runtime error, as the memory location is not writable.

In C, a string variable is defined as an array of characters, with the last element being a null character ('\0'). Here's an example of how to define a string variable:

```c
char str[11];  // Defines a string variable of size 11
```

A string literal, on the other hand, is a sequence of characters enclosed in double quotes. Here's an example of how to define a string literal:

```c
char *str = "hello world"; // Defines a string literal
```

In above example a pointer is pointing to the literal and the pointer can be used to refer to the string.

It's also possible to define a string literal as a constant, like this:

```c
const char *str = "hello world"; // Defines a string literal and pointer as a constant
```

It will also prevent the pointer to point to any other memory location, but the memory location still be a read-only.

5.3 len

Don't use `sizeof(string)`. This is determined in compile time and is based on the bytes this string takes up. Also, concating strings won't work using `+` as you're adding their pointers.

To get around this, put `#include <string.h>`. Then

- `strlen` returns the number of characters in the string not including `\0`. You can treat the return value as an integer.

5.4 Copying strings

When we copy a string, we overwrite what was previously there. When we concat strings, we add one string to the end of what was previously there in the other.

`char *strcpy(char *s1, const char *s2);`

Overwrites what was at the start of `s1` with `s2`. Note that `*s2` must be a string **(either a string variable or a char array that includes a null terminator)**.

DO NOT COPY TO ANYTHING THAT IS A STRING LITERAL – THIS WILL RESULT IN UNDEFINED BEHAVIOR

Beware: `strcpy` is an unsafe function. Don't copy a large string into a char array that is too small. An error may be raised, or no error is raised and the program gets a bug.

For many unsafe functions in the C library, there is a safe counterpart. We have a safe function: `char *strncpy(char *s1, const char *s2, int n);`. Here, `n` is the max. chars that can be copied into `*s1`. It shouldn't be larger than the length of `s1`. It is **not guaranteed to add a null terminator (this occurs if `s2` is "cut off"),** and if that is the case, you will have to add the null terminator yourself, explicitly.

Copying pattern:

```c
char to_this[99];
char *temp = "12345";
strncpy(to_this, temp, 5);
to_this[5] = '\0';
// argument 2 in strncpy does not
// need to be a variable, it can just
// be "12345"
```

Alternatively, if the `n` argument to `strncpy` is larger than the string given in the second argument, then the `'\0'` will be added automatically. However, there will be cases where you won't know the size of `temp`, so it's safer to just add `'\0'` to the very end of `to_this`. If `temp` is smaller, then the `'\0'` will be added earlier and all will be fine.

I might prefer this:

```c
char to_this[99]; char *temp = "12345"; strncpy(to_this,
   temp, sizeof(to_this) - 1); to_this[sizeof(to_this) - 1]
    = '\0';
```

5.5 Concating strings

Adds to the end of what is previously there. Appends to it: `strncat`. `n` indicates the max. no of chars, not including null terminator, that should be copied from `s2` to the end of `s1`. `strncat` always adds `'\0'` to the end of `s1`.

Pattern:

```c
strncat(s1, s3, sizeof(s1) - strlen(s1) - 1);
// the -1 makes room for the null terminator.
// sizeof(s1) - strlen(s1) -1
// gives us the unoccupied length
```

This pattern prevents the edge case of `s1` being overcrowded by limiting how much of `s3` can be copied in there.

5.6 Searching characters

`char *strchr(const char *s, char c);`

- String to search, the character to search for
- Returns the pointer to the character that was found (first instance), and returns `NULL` if it can't find a character.
- If you want an index, use pointer arithmetic to determine the index: `p - s1` where `p` is what was returned by `*strchr` and `s1` is the string.

5.7 Searching substrings

`char *strstr(const char *s1, const char *s2);` returns the pointer to the character of `s1` that begins the first substring that matches `s2`.

`strstr(s1, s2)- s1` is similar to `s1.find(s2)` in python

6 Reading inputs and IO

We still need `#include <stdio.h>`, and both `printf` and `scanf` use format specifiers.

Before calling `scanf`, we print the prompt first by convention.

`scanf("%lf", &cm)` asks us to input a long float. The number of parameters after the string must be equal to the number of format specifies after the string. The reason why `&` is here, because in order for `scanf` to change the value of `cm`, it is necessary to tell `scanf` the location of the `cm` variable. `&` is the symbol that gets the location of the variable. `scanf` places the input number to the location `&cm` so we can use it. `&` is related to pointers, which we will look at later – but for now, `scanf` requires `&`.

For character arrays, you do NOT need the & symbol.

Analogous to `cm = float(input("Type a number of centimeters: "))` in Python, where we added the prompt.

Here's a cheat-sheet for string formatting:

- `%c` for single char, a pointer to an individual character
- `%d` for decimal, base 10. Works with `int` and `long`
- `%e` for exponential floating point
- `%i` for integer, base 10
- `%o` for octal, base 8
- `%s` for a string
- `%u` for an unsigned decimal
- `%x` for hex
- `%%` and `\%` should print a literal percent sign

Any program you run has standard input to read from your keyboard input. When you use `printf`, your data is written to standard output, and it defaults to refer to your screen.

Two streams are available when a program runs. We also have standard error. It is an output stream. Standard error also refers to your screen.

- Standard output is for normal program output
- Standard error is for errors

You might want to change where your outputs are placed. You might want standard output to be saved to a file, while standard error be printed to a screen.

`scanf` returns EOF if there's nothing to scan / standard input is empty.

6.1 Reading Files

- use `fopen` to open the file
- use `fgets` (or `fscanf`) to read its contents
- close it afterwards (the `with` keyword does not exist in C.)

Example:

```
#include <stdio.h>

```

```c
int main(int argc, char *argv[]) {
  FILE *fp;
  char buffer[100];

  fp = fopen("file.txt", "r");
  if (fp == NULL) {
    printf("Unable to open file\n");
    return 1;
  }

  while (fgets(buffer, 100, fp) != NULL) {
    printf("%s", buffer);
  }

  fclose(fp);
  return 0;
}
```

6.1.1 `fgets` vs `fscanf`

`fgets` is for reading from files and `fscanf` is for reading structured data from files (similar to `scanf`, which requires users to input something that matches a pattern).

```c
while (fscanf(fp, "%d", &value) == 1) {         printf("%d\n", value);
}
```

`fgets` and `fscanf` will only read one line of the file at a time. Each time you call it, the next time it is called, it will read the next line, and it should return a flag (depends on which function you use) if the end of file is reached.

If you want to read the entire contents of a file in one go, use `fread`. It reads a specified number of bytes into a buffer.

A **buffer** is just a temporary storage area. Nothing special; it's not a special type, and it can be as simple as a string. Sometimes, you need it if you want to pass it into the `printf` function.

6.2 Redirecting streams

You can change streams while a program is executed.

6.2.1 Input Redirection

```
./a.out < number.txt
```

Here, `number.txt` goes into standard input, which is immediately read by the first `scanf`.

6.2.2 Output redirection

`./a.out > result.txt`

Everything that was printed gets saved in `results.txt`. Beware of file overwrites!

I/O directions are not C features but rather OS features.

Limitation: only one file can be used for I/O redirection. You need to do something else.

6.3 CLI and Type Conversions

Firstly, you should know that strings are `char` arrays, so you declare them like this:

```c
char *s = "bruh";
```

Now, I might want to perform Python's `int()` operation on it. It is: `strtol(s, ...)`, which stands for string to long. The API is `long int strtol(const char *str, char **endptr, int base);`

- `**endptr` is a pointer to a character array. When entered into the argument, the character the numbering cuts off, `*endptr` will point there (as if reassignment caused by a side effect in a function).

6.4 Passing information to your program using the CLI

We can write `main` as this (please name them like this):

```c
int main(int argc, char **argv){
    // argc is no. of arguments
    // argv is argument vector: array of strings
    // which contains all the arguments you put in,
    // in order.

    // this means argv[0] is the name of the executable, ./
       the_executable
}
```

And if I input this into the command line, in the same directory as the executable: `./the_executable arg1 arg2 arg3`, then `argc = 4` and `argv = ["./the_executable", "arg1", "arg2", "arg3"]`

6.4.1 Enforcing correctness and structure in arguments

- Use `argc` to check the number of required arguments.

```c
if (argc < 3) {
    printf("you stupid");
    return 1;
}

if (argv[1][0] != 'a') {
    printf("you also stupid");
    return 1;
}
```

You can have as many arguments – that's what `argc` is for. You also need `stdlib.h` included to do this.

Cast the numerator to a double before dividing if you want a decimal.

7 Memory allocation and Calling

I use `malloc` if I'm not sure exactly how large something is going to be.

Or we might want to encapsulate the creation of memory in a function, and the size is variable. Also, I might not want that memory to be gone after the function returns. That case, you `malloc` and return a pointer to that piece of memory.

Even if you're only using the piece of memory in the function, if you statically allocate memory, you can't use it again.

Nothing on the heap gets a label. It doesn't matter where you put it on the heap.

The name of an array evaluates to a pointer to the 0th element. The compiler does all of that for you, and it is stored on the symbol table, the same table that links variables to addresses. When you declare an array, the array's location is **not saved** in the stack. Imagine there is a table of symbols and address. That doesn't use memory, and I'm not going to think about it right now.

```
*my_int_array_size_42069 = malloc(sizeof(int) * 42069);
```

Try not to cause memory leaks. Don't reassign to a pointer that is returned from a `malloc` use without freeing it, and look out for pointers declared inside for loops, because they may be wiped after each loop.

7.1 Pushing onto the stack frame

During a function call, the higher it is on the stack, the later it is declared.

When a function is called:

1. We create a stack frame on it
2. For every parameter in the function, we allocate the right amount of space and give it a label. We allocate bottom up, so the first argument goes on the lowest part of the stack. This is always how parameter passing works, though things may be different for default parameters.

3. AFTERWARDS, they get their values from the corresponding arguments, from left to right (and thus bottom to top).

7.2 Passing an array into the argument from the function

When you use the name of an array in an expression, it evaluates to the address of the 0th element. If you pass it into the function, you are passing in exactly that, with (hopefully) no strings attached.

8 Structs

Arrays are useful for aggregating multiple values of one type into a structure.

Structs are used to aggregate data if the values of the data are not all the same type.

8.1 Using Structs in Functions

For arrays, you can't pass them into a function. Instead, you pass in its pointer.

For structs, if you pass in a struct, **you are passing in a copy.** The function gets a copy of the entire struct, including arrays. Any array inside of a struct is copied to. What if we want to retain changes to a struct by a function?

1. Return the struct back to the caller. This is ugly as you copy the struct twice. This is wasteful and is noticeable if the struct is large
2. **Pass a pointer to the struct as a parameter**

```
(*s).parameter = new_value;
```

We prefer the second case.

When a struct is defined, the compiler reserves a block of contiguous memory large enough to hold all of its members. The individual members of the struct are then laid out within this block of memory in the order that they were declared. Each member is given a unique memory address within the struct, which can be used to access it.

When you use a pointer to a struct, the pointer holds the memory address of the first byte of the struct. This means that you can use pointer arithmetic to access the members of the struct directly. For example, the expression `ptr->x` is equivalent to `(*ptr).x`.

An instance of a function that takes a pointer to a struct as an argument:

```c
void  printPoint(struct Point* p) {
    printf("(%d, %d)\n", p->x, p->y);
}
```

8.2 Typedef

```c
typedef struct node {
    // ...
} Node;

// then we can use Node in place of struct node
// when saying the type of something
```

Just always use `typedef` from now on to avoid the hassle of having to type `struct` every time you would've typed its type.