CSC209 Notes

Operating System (OS): System software that manages hardware, other software, and common computer services (eg. Windows, macOS, Linux, Android, iOS)
Unix: A family of OSs based on the original Unix OS developed by AT&T. Or, anything following Unix principles:

1) Do one basic thing well *with basic variations*
2) Simple input formats *plain-text, no interactive inputs*
3) Simple output formats *plain-text, can be passed as input to another tool*

Linux: An open-source family of Unix-like OS systems (eg. Ubuntu, Debian, Fedora Linux)

Graphical User Interfaces (GUI): Use of icons, menus, and mouses to manage interaction with a program
Command-Line Interfaces (CLI): Use of input commands and arguments to manage interaction with a program

Shell: Program acting as interface between human/other programs and the OS, providing basic OS services.

- **Unix Shell:** A shell for Unix-like OS's, which have CLIs.
- **Bash:** A Unix shell and command language used in most Linux distributions

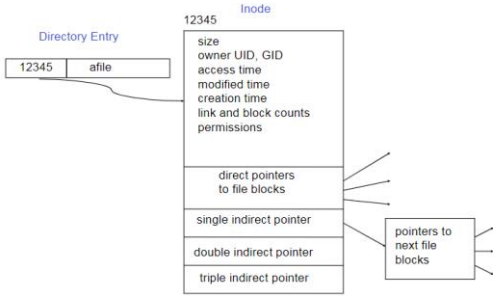
Command	*Symbol*	*Description*
`$ [A] [B] [C] [B]`	`$`	Shell prompter. Like Python's "`>>>`"
	`[A]`	A built-in command.
	`[B]`	Optional settings/flags(s), which are space-separated and start with "`-`" or "`--`"
	`[C]`	Argument(s), which are space-separated. Extra arguments are ignored.

File Descriptor: An integer assigned by the OS as a key for a specific open file/communication channel.
Standard Input: Default place where programs read text (ie. user input). → file descriptor 0
Standard Output: Default place where programs print output text (ie. the CLI). → file descriptor 1
Standard Error: Default place where programs print error text (ie. the CLI). → file descriptor 2
Piping (`<`, `>`): Redirecting a program's output or input to something other than standard input/output

Inode: A data structure in Unix-style file systems containing file information (ie. metadata, content location on disk)
Directory Entry: A mapping of file names to inodes
Directory/Folder: Contains directory entries and other directories

- "`.`" refers to the current directory
- "`..`" refers to the parent directory
- "`*`" refers to everything in the current directory

Root Directory (`/`): The top of directory hierarchy. Contains all files.



Permissions

In Linux, multiple users can share the same filesystem, so varying levels of user access and access scope is needed.
User (`u`): Only accessible by owner
Group (`g`): Only accessible by a defined group of users
Other (`o`): Accessible by everyone not in user/group
Read (`r`): View contents of file/directory
Write (`w`): Edit/delete file, add/remove files in directory
Execute (`x`): Run a file or access subdirectories

Permissions Display	*Symbol*	*Description*
`[A][BCD][EFG][HIJ]`	`[A]`	File type. "`-`" for regular/plain file and "`d`" for directory.
	`[BCD]`	User permissions. B, C, D, are respectively "`r`", "`w`", "`x`" if the user has that scope of access. Otherwise, the letter is replaced with "`-`"
	`[EFG]`	Group permissions. Same letter conventions as user permissions.
	`[HIJ]`	Other permissions. Same letter conventions as user permissions.

Basic Commands

Shell Command	*Description*
`echo [A]`	Print [A] and a new line
`pwd`	See what directory you're in
`ls [A]`	See directory [A]'s files
`ls -l [A]`	See directory [A]'s files, long-format (includes permissions, file size)
`cd [A]`	Go to directory [A]
`mv [A] [B]`	Move [A] to file path [B], or rename [A] to [B]
`mkdir [A]`	Create directory in file path [A]
`rmdir [A]`	Delete directory in file path [A]
`rm [A]`	Delete file [A]
`cp [A] [B]`	Copy file path [A] to file path [B]. Overwrites [B] if it exists
`chmod [A] [B]`	Change permissions of file [B], where [A] accepts... • `(a/u/g/o)(+/-/=)(r/w/x)`, ... *eg.* `a+x, u=rw` means all can execute, user is read/write • (0-7)(0-7)(0-7) *eg.* `741` is (4 + 2 + 1)(4 + 0 + 0)(0 + 0 + 1), so user is `rwx`, group is `r--`, other is `--x`
`head [A]`	Output first 10 lines of file [A]
`tail [A]`	Output last 10 lines of file [A]
`diff [A] [B]`	Output line-by-line differences of files [A] and [B], what to edit to make A = B
`comm [A], [B]`	Output line-by-line differences of files [A] and [B]
`cut [A]`	Delete something in file [A] and output it. Flags are mandatory to specify what to delete.
`cat [A]`	Print the contents of file [A]
`cat [A] [B]`	Print the concatenation of [A] and [B].
`wc [A]`	Output number of characters in file [A]. Can also find line count, word count, etc.
`od [A]`	Write a file [A] that is in octal to standard input. Used for reading binary files.
`grep "[A]" [B]`	Output all lines in file [B] containing regex [A]. Flags are useful here.
`gcc [A]`	Turn C or C++ file [A] into executable filename of default name `a.out`
`cpp [A]`	Run the C preprocessor on C file [A]
`as [A] [B]`	Assemble file [A] (in assembly code) to executable file [B]
`make`	Run a Makefile inside the current directory
`kill –[A] [B]`	Send signal [A] to process with ID [B] (see here for signals)
`ps`	View currently-running processes and their process IDs
`fg`	Resume the most recently suspended process
`man [A]`	Get more info on command [A]
`*.[A]`	Every file with the extension [A]. `*` is called a wildcard character. This is called "globbing"
`[A]; [B]`	Execute command [A], then command [B].
`[A] > [B]`	Pipe/redirect command [A]'s output to file path [B]. Creates/overwrites files.
`[A] >> [B]`	Append command [A]'s output to file path [B].
`[A] 2> [B]`	Pipe/redirect error of command [A]'s output to file path [B]. Creates/overwrites files.
`[A] >& [B]`	Pipe/redirect output of command [A]'s output to the stream with file descriptor [B].
`[A] &> [B]`	Equivalent to the above command.
`[A] < [B]`	Pipe/redirect file path [B] to command [A]'s input.

`[A] << [B]`	Pipe/redirect string [B] to command [A]'s input.
	[B] is of format "`EOF`
	`Blah`
	`EOF`". `with the new lines!`

`[A] | [B] | …`	Redirect command [A]'s output to command [B]'s input. Left to right. Can be chained.
`./[A] [B]`	Run executable file [A] with arguments [B]. `./` is not necessary if piping or using flags.
`../[A] [B]`	Run executable file [A] located in the parent directory with arguments [B]
`../../[A] [B]`	Run executable file [A] located in the grandparent directory with arguments [B]

- Piping can be chained – `[A] > [B] < [C] 2> [D]` redirects [A]'s output, input, and error into [B], [C], [D]
- Consider `sed` for finding and replacing, or `sort`. Look stuff up online!

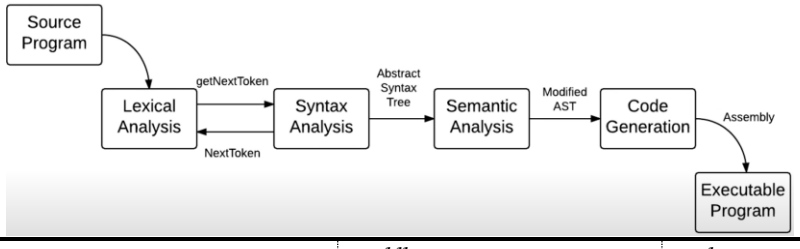
Running Code

Language	*Description*	*Code*
Interpreted	Code files are passed on as arguments to an interpreter in command prompt.	`> python3 hewwo.py`
Compiled	Code files are compiled into executable files in Unix. Then you run executable.	`$ gcc -Wall -g -std=gnu99 -o hewwo hewwo.c` `$ ./hewwo`

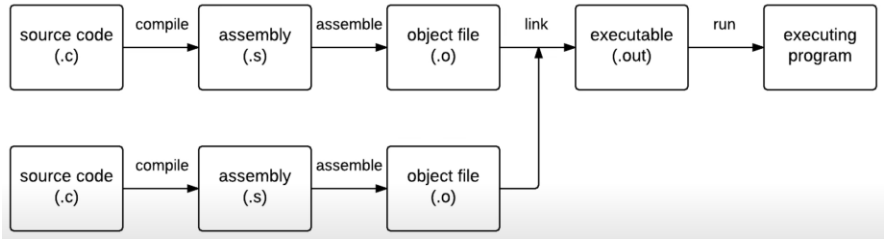
Argument	*Description*
`-Wall`	Report compiler warnings
`-g`	Generate debugger information
`-std=gnu99`	Specify language standard *(version of C)*
`-o hewwo`	Specify name of output executable file *(default is* `a.out`*)*

From Code to Executable

Compiling: In general, the translation of a programming language, usually from higher-level to lower-level



Beginning	*Middle*	*End*
Lexical & syntax analysis *Turn code to abstract syntax trees, intermediate languages. GCC converts to Gimple & Generic. C's preprocessor processes macros, compilation, etc. before code is compiled.*	Semantic analysis *Tries to optimize for speed in code. This process is often blended into other processes*	Code generation, Executable *Turns intermediate code into assembly language, which can be turned into an executable.*



Assembly Language/Assembly: A low-level programming language (depends on the computer) with very strong correspondences between its code and machine code instructions.

- **Assemblers:** Software that converts assembly files (.s) into object files (.o).
- `gcc` accepts .o and .s files!

Linker: Combines assembled object files into one executable file – a package of all instructions and all data)

- Executables not guaranteed to run on all computers – assembly and linking is machine-specific

Command	*Description*
`gcc -D A1 A2=5`	Run a file, first defining the macros A1 *(set to 1 by default)* and A2 *(set to 5)*
`gcc -S`	Stop after compiling to .s (assembly) file; don't assemble
`gcc -c`	Stop after assembling to .o (object) file; don't link
`gcc file1.c file2.c`	Link multiple files in the compiling process

Makefile: A file that manages dependencies between files and linking. Has the general format specified below:

```
Target: Prerequisites/Dependencies
    0+ shell commands separated by lines that compile Target from Dependencies
```

Makefile Command	*Shell Command*
`A.o: A.c A.h` `  gcc -c A.c -o A.o`	`$ make A.o`
`B.o: B.c B.h` `  gcc -c B.c -o B.o`	`$ make B.o`
`C: A.o B.o` `  gcc A.o B.o -o C`	`$ make C` → this will know to call `make A.o` and `make B.o` if not called before!

Symbol	*Description*
`%`	Placeholder for a name
`$<`	Name of first dependency (including the final .c/.o)
`$^`	Name of all dependencies
`$@`	Name of target
`$?`	Name of all out-of-date prerequisites
`X = A.o B.o C`	Set variable `X` to the files `A.o, B.o, C`
`$(X)`	Refer to variable `X`
`.PHONY`	Defines the following word as a keyword instead of a file

```
FILES = A.o B.o C

D: $(FILES)  (Shortcut to A.o B.o C)
    gcc $(FILES) -o D

%.o: %.c  (.o files built from similarly-named .c)
   gcc -c $< -o $@

.PHONY: clean
clean: (creates a command with no dependencies)
    rm *.o
```

- You can run make commands with custom variable arguments, like `make D FILES="asdasd.o"`

Wav Files

Wav File: A binary file that stores music. Consists of:

- One 44-byte header – information about wav-file, how to play file
- Many 2-byte values – alternating, for left and right channels. An integer representing frequency

```
$ od -A d -j 44 -t d2 cool_music.wav
// First column is byte number, other columns are the values in each byte
0000044         2       2       2       2       2       8       8       8
0000060         8       8      16      16      16      16      16       4
0000076         4       4       4       4
0000084
```

Argument	*Description*
`-A d`	Write output in base 10 (d, decimal)
`-j 44`	Skip first 44 bytes of file (the header)
`-t 2d`	Indicates the file consists of two-byte values

Shell Programming

Shell Interaction	*Print Output*	*Description*
`$ i=5` `$ i="5"`		Set variable `i` to 5. Everything is a string in shell, including commands! Use ' or "" or have \ for arguments with spaces.

`$ 'echo' $i`
`$ echo $i`
`$ echo ${i}wow`
	`5`
	`5`
	`5wow`
		`$` reads what follows as a variable.
		
		`${}` is used for combining variables with text, no spaces between

`$ x="$i wow"` `$ echo x`	`5 wow`	`" "` allows variable substitution. It also allows \ and " to keep their special meanings.
`$ x='$i wow'` `$ echo x`	`$i wow`	`' '` prints the string exactly as it is.
`$ expr $i + 1`	`6`	`expr` evaluates what follows as math, prints it, and returns it.
`$ expr 5.1 + 1`	`Error`	Math operations only work on integers.
`$ expr $i '*' 2`	`10`	`'*'` or `\*` is used for multiplication, as `*` already exists in syntax.
`$ echo expr $i + 1`	`expr 5 + 1`	`echo` prints everything after it, literally.
`$ echo` `` `expr $i + 1` ``	`6`	`` ` ` `` evaluates everything inside it first.
`$` `` `expr $i + 1` ``	`error`	`` ` ` `` turns the input to 6, which is not a command.
`$` `` v=`ls -l` ``		Sets variable `v` to the output of `ls -l`
`$ ./a_program "$v"`		`" "` is used to not interpret white spaces in `v` as other arguments.
`$ read x y`		`read` 2 space-separated string inputs from user. Store in `x` and `y`. ➢ Extra arguments are put into the last variable. ➢ If not enough inputs, empty variables set to empty string
`$ echo $PATH`	`/bin:/usr/…`	`PATH` is a special variable, a list of file paths separated by `:`. To parse commands without `/`, Unix looks in `PATH` directories to find the item.
`$ echo $?`	`0`	`$?` is the previous command's return. 0 is success, $\neq 0$ is failure.
`$ test 03 -eq 3` `$ echo $?`	`0`	`test` can compare integers. Use flags `-eq` ($=$), `-ne` ($\neq$), `-gt` ($>$), `-ge` ($\geq$), `-lt` ($<$), and `-le` ($\leq$).
`$ [03 -eq 3]`		`[…]` is alternative notation for `test`.
`$ test 03 = 3` `$ echo $?`	`1`	`test` can compare strings. Use flags `=` ($=$), `!=` ($\neq$).
`$ test -z $v`		Test if a variable is a length-0 string.

`$ test 3 -le 4 -a 3 -ge 4`
`$ test 3 -le 4 &&`
`  test 3 -ge 4`
		`test` allows booleans with `-a` (and) and `-o` (or).
		You can also booleans `&&` and `||`, which auto-stop (eg. for `x && y`, if `x` is false, `y` is not evaluated)

`$ test –[A] [B]`		`test` can check file [B]'s existence/type, where [A] can be `-f` (is regular file), `-d` (is directory), `-s` (is non-empty regular file), etc.

`$ if [A]`
`  then [B]`
`  elif [C]`
`  then [D]`
`  else [E]`
`  fi`
		If statement that executes [A], and if successful (return is 0), executes [B], otherwise executing [C], and if successful, executes [D], otherwise executing [E].
		
		Empty branch bodies are not accepted; use `:` (like Python's `pass`)
		
		Each keyword needs its separate line to be parseable.

`$i = 0` `$ while test $i -lt 6` `  do` `  i=`expr $i + 1` ` echo $i` ` done`	`1` `2` `3` `4` `5`	While loop, basic format. If statement [A] in a while loop prints and you want to ignore the printing, redirect the printing to a special file: `[A] > /dev/null`.

`$ for i in *.c`
`  do`
`  echo $i`
`  done`
	`file_a.c`
	`file_b.c`
	`…`
		For loop, basic format. Example code iterates over .c files.
		Files: Iterate through files
		Strings: Iterate through space-separated substrings
		Integers: Iterate w/ `` `seq [start] [increment] [end]` `` *(inclusive)*

`$ for i`
`  do`
`  echo $i`
`  done`
	`arg1`
	`arg2`
	`…`
		If this code is in a .sh (shell) file, it will iterate over every command-line argument given when executing this file.

`$ case $i in`
`  hewwo)`
`  echo hi`
`  ;;`
`  cool*)`
`  echo wow`
`  ;;`
`  *)`
`  echo okay`
`  ;;`
`  esac`
	`okay`	Switch statement that does exact comparisons (despite the "in" keyword) in top-to-bottom order.
		`*` matches any string, so it's a good default/final case.
		`cool*` is any string starting with `cool`

`$ echo $#`	`…`	`$#` is the number of input arguments.
`$ echo $0`	`file.sh`	`$0` is the name of the .sh file/program.
`$ echo $i`	`argi`	`$i` is the *i*-th input argument.
`$ echo $*`	`arg1 arg2 …`	`$*` is all input arguments, separated by spaces
`$ echo $@`	`arg1 arg2 …`	`$@` is same as `@*`, but if it is being iterating over, `"$@"` makes sure words with spaces are processed as a single argument unlike `"$*"`.
`$ shift`		`shift` sets `$[i]` to `$[i - 1]`, discarding `$1`.

`$ while $# -gt 0`
`  do`
`  …`
`  done`
		Typical argument processing loop that uses `shift`, looping until the number of arguments is 0.

`$ # hewwo`		`#` is a comment. Same syntax as Python
`$ #! /usr/bin/sh`		`#!` is a shebang line. Indicates what interpreter the file should be run with – Bourne shell, Bash shell, PowerShell, Python, etc.