Stanley Wei

# COM SCI 35L

Paul Eggert

Spring 2023

**Table of Contents**

**Topics**

- **File systems** - organized/organizing collection of data on [persistent] secondary storage
- **Scripting** - the practice of writing "quick and dirty" code, as opposed to more rigorous traditional programming
    - More maintainable, faster to write, but possibly slower, less robust
        - A script will generally be geared towards a single specific task, whereas a program may support many different tasks
    - Scripting languages: Shell, Python, JavaScript (, Lisp)
- **Building & distribution**
- **Version control**/configuration management: maintaining separate versions of a program
    - May distinguish between dev versions vs user versions, or even dev versions vs other dev versions (in the case of collaborative development)
    - Version control typically done using *Git*
        - Users' view of a repository may differ from the actual internals
- **Low-level debugging** (for OS, using C)
- **Client-server programs** - a form of *distributed application* (operating on >1 computer)
- *CS130:*
    - Programming, data design, integration (i.e. combining different software components), configuration (taking a program for a general purpose, and modifying it to perform a specific task successfully), testing, forensics (retrospective attitude, as opposed to proactive debugging)
        - Configuration - LAUSD payroll system failed due to improper configuration

**Programs & Applications**

- Categories of programs:
  - Standalone applications (e.g. machine code) - IoT
  - Applications atop an OS (e.g. terminal scripts, e.g. Emacs)
  - Web applications/PWAs operate atop a browser, atop an OS
    - Utilize a *client-server approach*
- Layers of a computer (lowest to highest):
  - The ***hardware*** is commanded by instructions from the *kernel* & *applications*
  - The ***kernel*** (operating space) runs in the kernel space, invisible to *applications*
    - Walls off applications from executing certain instructions on the hardware, allowing only basic safe processes + a limited set of proper system calls
      - Most operations used by applications are just basic load/add/mul
  - ***Applications*** (*processes*) run atop the kernel, each in their own application space
    - May be built upon the C library, with additional libraries (and eventually the application itself) stacked on top
    - Each application is partly isolated from other applications - can generally only communicate indirectly (e.g. through the kernel (via file system), network, etc.) or through the use of pipes
- Programs vs processes
  - *Program* - a relatively static object (downloaded at some point, doesn't change)
  - *Process* - a program being run by the machine, taking input and producing output
    - Program + internal data + connections to the external world (e.g. IO, file inputs/outputs, network connections, etc.)
    - Can also start/stop its own *subsidiary processes*, pass its own data as input/take the subsidiary's output as input via pipes
    - Output need not be to the screen as text - can also be to a windowing system, which can itself pass input back
      - Cloud systems may forego changes beyond terminal text changes
  - Ex: an Emacs *program* is just a binary file/directory somewhere on the system; an Emacs *process* is an actual running instance of the program.

**Emacs**

- Written in C at the bottom level; higher levels at Lisp [Emacs Lisp]
    - C handles low-level IO, signals, Lisp interpreter [~15%]
        - Kept relatively simple; handles any tasks that might take up lots of system resources, CPU instructions, low-level objects
            - Is not properly object-oriented, but mimics the structure
    - Lisp [~85%] as a scripting language - handles most complex decisions, control
        - Does not handle most CPU-intensive tasks (passes them off to the C layer), but makes most decisions
    - Browser low/high-level differentiation: C++ vs JavaScript/Python
- Notes on Emacs
    - Emacs is modeful, i.e. the actions that it takes to a specific character command depend on its mode (the current context)
    - Is capable of reflection-introspection: can modify Emacs from within itself
    - Can be reprogrammed with Lisp code (defun)
- Emacs refcard: [https://www.gnu.org/software/emacs/refcards/pdf/refcard.pdf](https://www.gnu.org/software/emacs/refcards/pdf/refcard.pdf)
- Commands:
    - Shell: M-X shell RET [M=Meta=Edit/Alt]
        - Quote next character (write to text [translating to plaintext-representable format if needed], e.g. Ctrl-C -> ^C [representation of Ctrl-C as ^C]): C-q
        - Emacs still controls character inputs - simply passes all inputs to shell
    - Enter dired (directory edit) mode: C-x d *directory name*
        - Rename: [dired] R
    - Disable [buffer] read-only: C-x C-q (Does not modify system read-only protections)
    - Display character information: C-u C-x =
    - Insert by hexadecimal encoding: C-x 8 RET
    - Replace string: M-x replace-string
- Set key: M-x global-set-key \n *key* *function*
- Message: message, concat
- Time: current-time-string, format-time-string
- Load file: M-x load-file

**Emacs Internals**

- Emacs internal data has two notable types: *buffer* and *string*
- **Buffer** - an Emacs structure (byte sequence) in RAM representing the contents of a file
    - Are easy to edit - cheap insertion
        - Is represented as an array of bytes with a gap [garbage] in the middle, i.e. divided into two halves; modifying fills in the gap with content
            - Is resized as needed when the buffer fills - average O(1) [amortized] performance for single-character insertions
        - Cursor is always kept at boundary between the first half and gap (*the point*); first/second halves are rebalanced as needed to maintain this (inefficient to move cursor, but cheap to insert)
        - Important location - the mark (user-set)
            - Region of interest (similar to a highlighted area) is the area between the point and the mark (be it an area to be deleted, copied, etc.)
    - Used to read/write to files; given a file, will read the whole file, no matter its size
    - All contents on screen usually represented in data by a buffer/buffers
- **String** [Emacs] - an Emacs structure representing a byte sequence, similar to a C string
    - Are usually not modified (harder to modify, relative to buffers), smaller than buffers

**Character Bit Representation**

- C characters represented by 8 bits
    - ASCII characters contained within a 7-bit set (+1 bit for verifying correct format)
        - Contains 127 characters: first 32 control characters (first control character - null), all printable characters, final control character [del]
    - Neither Ctrl nor Meta are actual characters themselves, just modifiers a la Shift
        - Bit representation of a Ctrl-*char* is identical to the bit representation of *char*, with the exception of one bit (second) turned off relative to *char*
            - Meta-*char* turns one bit (first) on relative to *char*
- Typing Ctrl-A maps to hexadecimal code 1 [Ctrl-A], Ctrl-C to code 3 [Ctrl + *char* is a single distinct 7-bit character, not separate Ctrl + *char* codes]
    - C-m interpreted as Enter (carriage return = \r), C-i as [horizontal] Tab [=\t]

**Regex**

- Regular expressions - used in grep [+ other places], incl:
    - Simple text strings [patterns]
    - Simple text strings + * [include all]

| Regular Expressions | |
|---|---|
| **Pattern** | **Matches** |
| abc | Matches “abc” |
| . | Matches any single character |
| E* | Zero or more Es |
| \(bcd\)* | Any number of “bcd”s |
| \(bcd\)\{5,10\} | Between 5 and 10 “bcd”s |
| \(bcd\)\{5,\} | 5 or more “bcd”s |
| \bcd\)\{,10\} | Up to 10 “bcd”s |
| \C (for any special character C, e.g. *) | Matches just C |
| \ + ordinary character | **Behavior undefined** |
| \ + number | Matches pattern at the regular expression of that number in the command |
| ^E + *text* | E at line start |
| *text* + E$ | E at line end |
| [abcxy] | Matches any character a, b, c, x, y |
| [A-Za-z0-9] | Equivalent to [ABC…Zabc…z012…9] |
| [*start char*-*end char*] | Matches any character between start, end chars (via ASCII encodings) |
| [spc-~] | Matches all ASCII characters |
| [^ABC] | Matches any character not A, B, C |
| [A^BC] | Matches A, B, C, or ^ |

| | |
|---|---|
| [][] | Matches [, ] (cannot be other way around) |
| [*\.] | Matches *, \, or . (not special in [] exps) |
| [[:alpha:]] | Matches character class alpha |
| [^[:ascii:]] | Matches all non-ASCII characters |
| [A-Za-z_][A-Za-z0-9_]* | Special: Matches C identifiers |

- Regular expressions: enclose in “” [grep]
    - * (asterisk), \ (backslash), . (period) not special within bracket expressions
    - Character classes - alpha (alphabetic), digit (English UTF-8 locale), ascii
        - Emacs supports character class ascii; grep does not
            - Ascii in grep: [NULL-DEL] matches bit patterns 0000000 to 0111111
        - Same as in <ctype.h>
- Many other programming languages have their own regex versions (with potentially different syntax)
    - grep -p switches to Perl regex
- Option -E (extended regular exp. notation = ERE) prior to regex allows for omitting \
    - ERE adds certain additional operators: | (or), ? (=\{0, 1\}), + (=\{1,\})
        - GNU provides \| in BRE
        - ab|cd == (ab)|(cd) [precedence: counting operators > concatenation > or]
    - Tradeoffs - operator| requires backtracking capability (more expensive)
    - Vs basic regular expressions (BRE)
- Regex tools
    - sed - stream editor
    - awk - a more robust version of sed [Perl[/Python] - more robust versions of awk]

**Unicode**

- Non-ASCII characters (e.g. Unicode)
    - ASCII only uses 7 of 8 bits in a byte - allows for an additional 128 characters, + ASCII, to be supported in a byte
- Historically - different character sets sprang up adding an additional 128 characters based on the [primarily European] language in question (e.g. French set, Greek set, etc.)
    - ISO-8859-x [x indicating specific char set]
    - Flaws - didn't work for international languages, interpretation of bit patterns relied on knowing which character set was in use
- **Unicode** - international standard assigning character codes to all characters in all supported languages internationally
    - Contains approx. 2^24 characters
    - Originally stored in 16-bit chunks (general standard for char data types)
        - 8-bits only for C unsigned short
        - Issues: not backward-compatible with pure ASCII files, eventually not big enough to fit entirety of Unicode
    - Next moved to 32-bit chunks
        - Issues: still not backwards-compatible, memory-inefficient
- **UTF-8** - international standard for representing Unicode
    - Designed to be backwards-compatible with ASCII, space efficient
    - Also contains unambiguous character boundaries
    - Encode character string into byte string; decode byte string to character string
    - Has 3 kinds of bytes:
        - 0 + [7-bit sequence] - represents an ASCII character
        - 10 + [6-bit sequence] - trailing (2nd or later) byte in a multi-byte character
        - [Sequence of 1s] + 0 + [sequence] - beginning sequence of 1s encodes # of bytes in character
            - Length encoded in base 1 - max 4 bytes
        - Allows for a character to only use up the minimum number of bytes required to represent that character
- Invalid UTF-8 bytes:
    - Indicated length is incorrect or >4

        - Not all 4-byte sequences have defined character representations [Unicode not entirely filled-in - has room for expansion]
    - Character is truncated [partial character] (e.g. streaming data from a network)
    - An expected trailing character is not a trailing character
    - Unexpected [isolated] trailing character
    - Specific: Bit pattern: 11000000 (byte 1) + 10111111 (U+003F) == 00111111 (ASCII)
        - ASCII characters can be represented in longer-byte sequences [overlong encoding] - evade regular ASCII regex patterns, e.g.
        - Declared invalid, albeit only discovered 10 years after launch of UTF-8

**The Shell**

**Overview**

- Emacs IDE/shell/files: persistence, getting work done, writing programs
- Roles of the **shell** (sh/bash):
    - Can be used as a ***command language*** for running other commands
        - Can be used as a ***scripting language***, reading shell commands from a script
        - Is *interactive* - a command is written and immediately executed, all on the fly, without having specified the next line
    - Can be used as a ***configuration/setup language***, i.e. as a means of getting other programs configured, set-up, and running
        - In contrast with application-specific languages (ASLs) - the shell eschews performance/efficiency for ease-of-use/integration
        - Shell commands may spawn off other applications (e.g. nano, gedit), which can be both controlled by and controlling the shell
- The shell designed as a small, simple form of getting more robust applications running
    - Designed as a simple way for the user to access/run programs or the OS (i.e. as an interactive layer between programs/OS, and the user)
        - Modularity - programs as modules
- Given a line representing a command: determines what the words (tokens) in the command are, invokes the command given (if found in PATH) and passes it an argument vector representing each word in the command (e.g. grep word1 word2 -> [“grep”, “word1”, “word2”])\
    - Words identified by separation by spaces (number of spaces not considered)
    - [command name] + [arguments]
    - Backslash escapes - Backslash can be used to insert special characters (i.e. \spc interpreted as part of a word, not a separator)
        - Two backslashes becomes one backslash
            - Represented by two backslashes in C
        - Backslash + newline eats up the newline
    - Quoting:
        - Single quoting: Any characters inside the single quotes interpreted as a single word

        - Any special character can be put in as a normal character (incl. newlines), except for a single quote [apostrophe]
        - Putting a single quote inside a word (e.g. a’b c’d) works and is interpreted as one word (can be used to insert apostrophe)
    - Double quoting: Similar to single quotes, except a few characters are interpreted as special
        - Note: single quote (apostrophe) not special
        - Special characters:
            - $ replaces a value with the value of the shell variable with the corresponding name (e.g. x = “hi” -> “$x” becomes “hi”)
            - Backslash used to escape
                - Two backslashes produces a single backslash
    - Large source of security vulnerabilities - not knowing language quoting rules

**Shell Commands**

| Command | Function |
|---|---|
| cat (concatenate) | Echoes all input given in (repeats typed-in text) |
| ps (process status) | Lists running processes on the current terminal associated with the current user |
| less | Provides control over display data (similar to nano) |
| grep | Given a pattern and a directory path, matches all files on path with lines matching the pattern |
| od | Given a file, outputs any non-ASCII characters as their Unicode character encoding |
| test | Performs various tests (e.g. checking file existence, string comparisons) |
| : [single colon] | Always succeeds |
| kill | Given a process ID, kills said process |
| wait | Given a process ID, waits for said process to finish |
| touch | Creates new file |
| mkdir | Creates a new directory |
| rm | Given a file, removes said file |
| ln | Given a file and a link name, creates hard link |
| ls | Given a directory path, lists contents of directory |
| find | Search for files recursively within a directory |
| shuf | Given a file, shuffles and prints the lines of the file |
| seq | Prints a sequence of numbers |

| | |
|---|---|
| sed | Reads standard input and outputs to standard output, but edits each line before outputting it |
| awk | A more robust version of sed |
| chmod | Change file mode bits (notably read/write/execute permissions) |
| which | Given a shell command, prints binary file associated with said command |
| tr | Translate or delete characters (tr [SET1] [SET2]) |
| sort | Given a file, sorts lines in file in alphabetical order |
| comm | Given two files (with lines in sorted order), compares file contents |

- cat
  - If given file name(s) as argument, copies file data (in sequence) and output to standard output [terminal]
  - Reads from standard input if no arguments are provided, or an argument is "-"
    - "-" = stdin is convention
  - Mechanism: reads incrementally, e.g. in 1 MB blocks (stores 1 MB in RAM)
- ps
  - -e argument also includes system processes
  - Timestamp indicates amount of CPU time used
- Backslash indicates command continues to next line
- less - control over display of data (similar to nano)
  - /*search term* searches
  - Akin to a read-only text editor
  - Shell command | less passes the output of the shell command as input for less
    - Vertical line indicates a pipe (cmd1 | cmd2 -> cmd2 takes cmd1's output as input)
- Grep
  - Grep [pattern] [file] - outputs every line in file matching the pattern [regex]
  - grep -v: print only names of matching files
  - Notably, has two different failure exit statuses:

        - 1: No match found, no notable errors
        - 2: Serious error (e.g. file not readable)
- Od [file] - outputs any non-ASCII characters as their Unicode character encoding
    - -c: octal, -t: hexadecimal
    - X1: 1-byte
- test
    - test -r *file name*: test if file is readable file
    - test -f *file name*: test if file exists
    - test *string 1* -eq *string 2*: test if strings equal
        - Can be used to compare command exit status ($?) to exit codes, e.g.
        - Equal to [str1 -eq str2]
- sed "s/*search*/*replace*/2" replaces 2nd instance of search with replace in every line
    - /g for all; /3g for 3 and up; "1,3 s/…" for lines 1,3; -n only replaced lines
    - '5,$d' deletes 5th, last line; '*pattern*/d' deletes pattern matches
- Chmod bits
    - *file type*rwxrwx…; file type can be 'f', 'd', 'l', etc.
- Tr [SET1] [SET2]; -c = "complement"
- true, false
- set a b c: sets $1, $2, $3 to a, b, c
- shift: Shifts $1, $2, etc. (arguments) right by 1
- Shortcuts
    - Control-D/^D/C-D indicates EOF if at start of line, from a terminal
    - Control-C interrupts the current process
- Features
    - Write multiple commands on one line - separate with semicolons
    - Appending & to the end of a command will run it in the background as a separate process (as a descendant of the current shell)
- /dev/null: always-empty file (will always return nothing when read from)

**Shell Keywords**

- Some command names [words] reserved by the shell (designated keywords) for compound commands

            - Keywords only considered special [reserved] at the start of a new command
- *Shell keywords:*
    - ***exit*** - exits the shell
    - ***if, then, else, fi*** - parts of a compound command for conditional statements
        - if, then, else, fi (closing word); also: elif
        - if, then, else, fi only keywords at start of command
        - Ex: if test -r foo \n then cat foo \n else echo "foo missing" \n fi
            - Can also put sequences of commands, nest commands
                - Nesting conditionals - use else, fi to exit a nest
        - If a; then \n b \n else \n c \n fi
        - Exit status is the exit status of the last command of the then or the else (whichever was executed)
            - Omitting else - evaluates as true
    - ***case, in, esac*** - parts of a compound command for case statements
        - esac - closing word for a case chain
        - Ex: case $? in \n 0) echo str;; \n 1) echo str2;; \n 2) echo str33;; \n esac
            - Default case: *)
            - Cases can also be patterns
                - Use globbing patterns (same as used by the shell for file names)
                    - Ex: *.c);;, *.[ao])
            - Multiple cases: separate with vertical bar (e.g. 0|1)
            - Would work if all written on one line
        - Exit status is the exit status of the last command of the then or the else (whichever was executed)
            - No case match - evaluates as true
    - Keyword **{ + }** (group a series of commands; run in sequence)
        - Exit status = exit status of last command executed
            - If an earlier command fails, shell will continue to run subsequent commands [ignores failure]
    - Keyword **!** (negates command['s exit status])
        - Is considered its own word [need space between words]
            - Is a command, not an operator

        - Common extension (present in Bash; not universal):
            - !! = text of previous command
                - !! *arguments* add the arguments to the end of previous command
            - !* = arguments of previous command
            - !$ = last argument of previous command
        - Exit status: 0 or 1
    - ***while, do, done*** - parts of a compound command for [while] loops
        - Also: until (alternative), break, continue
            - No do-while loop defined
                - Alternative: while *cmds* \n do : \n done
        - Ex: while *cmd* \n do *cmd* \n done
    - ***for, in, do, done*** - parts of a compound command for [for] loops
        - Ex: for i in a b 27 c \n do *body*  \n done
            - Runs body 4 times, where $i is equal to a, b, c, 27, and c for the 1st, 2nd, 3rd, 4th iteration, respectively
            - Similar to python for loops
            - Alt: for i in *.c
- *Extra shell tokens:*
    - **&&** - if the left-hand command succeeds, run the right-hand command
        - Returns exit status of last command run
    - **||** - if the left-hand command fails, run the right-hand command
        - Returns exit status of last command run
    - **()** - act similar to {}, but runs contained commands as a subprocess
        - Subprocess - akin to running commands in separate shell process [a descendant of the original]
            - Any changes to the shell itself (e.g. changes to shell variables) done in a subprocess do not affect the main shell [are erased when the subprocess exits]
        - Curly braces do not guarantee commands are run as a subprocess
- *Shell variables:* ***name*=*value***
        - Can only contain strings (incl. strings of digits)
            - Using special characters (e.g. spc) necessitates quoting

        - Can have multi-character names
        - Expands right-hand side (e.g. references to shell variables with $ are replaced by the value of the respective variable)
        - Spaces around the = break the assignment
        - Can perform multiple assignments on one line
    - Used for two purposes - internal use (local to shell), export to subsidiary processes
        - Exporting:
            - *assignment* *command*
                - Exports assigned variable as environment variable of descendant process, for descendant process only
                    - Parent shell unaffected; variables unchanged
            - export *assignment*
                - Affects both shell, subsidiary processes
                    - Changes to exported environment variables also affect the exported values [even if not explicitly re-exported]
    - Built-in variables
        - **$?** - exit status of most recent command
            - Convention: 0 = success, nonzero = failure
            - Max: 128 or 256
        - **$!** - process id of [most recent] background command
        - **$0**, **$1**, **$2**, … correspond to the arguments of the process
            - **$0** corresponds to current process's name
        - **$*** concatenates all arguments $1, $2, etc. (except $0)
        - **$@** acts like $*, except when quoted
            - When quoted ("$@"), separates each argument into its own word
        - **$(***command name***)** runs the command as if it were a shell command (run as subshell), then saves the output
            - Is not a proper variable
            - Can replace $(, ) with ` [not common]
- ***Stages of shell command interpretation (done strictly in order):***
    - Delimit the command (default delimiting tokens: ; | \n)

        - Any commands not properly delimited (e.g. within a $() expression) may result in unintended output
    - Read in command as a series of words, and expand:
        - ***Tilde expansion***: ~ expanded into $HOME path
            - ~, ~/* expanded into $HOME
            - ~*user* expanded into user's $HOME
            - Only expanded if at the start of a word
        - ***Variable expansion***: $x evaluates to the value of shell variable x
        - ***Command substitution***: $(*cmd*) is expanded into output of the command
        - ***Arithmetic expansion***: $((x + 5)) evaluates to value of shell variable $x, plus 5
            - Arithmetic expansion takes priority over $((*cmd*))
                - Must add spaces between parentheses for latter expansion
        - ***Field splitting***: some words in a command may split into multiple words once interpreted
            - Ex: x='a b'; cat $x splits x into two words: 'a' and 'b'
            - $IFS [inter-field separator] = tab, space, newline
                - Can be modified by modifying $IFS
    - ***Pathname expansion/globbing***:
        - Characters:
            - *: matches any sequence of characters [incl. empty]
                - *.c returns all files ending with .c, e.g.
            - ?: matches one char
            - [*chars*]"": matches one char in a set
                - Negation: [!*chars*]""
        - *, ? do not match file names starting with . [hidden files]
    - ***I/O redirection***: some words are treated as directives to the shell in regards to what to do with input/output [rather than simple arguments]
        - Characters:
            - >*file*: create/truncate file [discarding all existing data], write standard output to the file
                - >>*file* appends to a preexisting file, rather than truncating
            - <*file*: write file to standard input

    - Will only be effective if the command has been told to read from standard input
    - <<EOF [+other commands] RET *lines*\n EOF: effectively creates a temporary file to pass as stdin
      - EOF can be any word - << will keep reading until it finds a match
      - Contents in EOF expressions are expanded unless the EOF at start is quoted
- Done left to right, independently
  - cat <x <y evaluates to just cat <y, e.g.
- Can be done without expressions
  - >x >y just creates two empty files, e.g.
- Note: can read from and output to the same file, possibly causing infinite loop
- (**#>**/**#<**): 0=stdin, 1=stdout, 2=stderr
  - & after >/<: 2>&1 changes the contents of 2 to be written to 1, e.g.
  - 2>&-: closes 2 [writing to stderr causes error]
- **<>** opens a file for both input, output
- Expansions do not occur in single-quoted strings
  - Only variable, command, arithmetic [$] expansions occur in double-quoted strings

**Shell Scripting**

- Shell scripting
  - A shell script is essentially a sequence of terminal commands written within a file
  - A shell script "expanded", command-by-command, into a simpler language during execution
- Traditionally, a shell script is split into multiple files (as needed) rather than combined into single big scripts with functions (unlike higher-level scripting languages, e.g. Lisp/Python)
  - Pros of split - forgoes | [pipe] functionality, split files do not all need to be shell scripts [can be C++ scripts, e.g.], more modular
  - Pros of all in one - ease of integration, viewability, package/distribute, more [space/memory] efficient [less overhead - 1 rather than 2 processes]

**The File System**

- Linux file systems structured as rooted trees, starting at the root
    - Directories as nodes, non-directory files as leaves
        - Non-directory files: regular files, symbolic links, special files
        - Files, directories, symlinks distinguished by first bit: '-' vs 'd' vs 'l'
- A file system is akin to a simple hash table: a [non-null] string (e.g. /usr/local/bin) maps to a directory/file
    - Consequence: a single string cannot map to multiple files (files cannot share the same full path, including name)
    - String is composed of file name components, separated by slashes (e.g. "usr", "local", "bin", directories, etc.)
- A single non-directory file can be present in several different directories at once
    - ***Hard links***: ln *location1/name1* *location2/name2* makes it such that the file with name location1/name1 appears also present as location2/name2 [files only]
        - Is the same file - modifying one will also be reflected in the other
            - Essentially creates another "name" for the file
            - rm-ing a file only removes a name/directory entry, not a file itself
                - Similar to a pointer falling out of scope in C++
        - Are identified by unique internal IDs (shown by ls -i)
            - Also do not share metadata
        - Akin to a reference in C++
    - Consequence: a single Linux file can have multiple names
        - Link count akin to the degree of the file in the file system graph (# of directories pointing to it)
- ***Symbolic links***: ln -s *file* *link_name* creates a symbolic link pointing to the original file
    - Will point to a file, even if the file does not exist [dangling link]
        - Can store anything in a symbolic link, incl. directories, itself
    - During file name expansion, will replace the link with its contents
    - Akin to a pointer in C++

**Lisp Scripting**

- Lisp [LISt Processing]
    - Relatively old [1950s] + simple (root language - inspired other languages)
    - Originated as an AI language
        - Was originally slow, theoretical - made more practical over time
    - Variants: Lisp 1/1.5, Scheme, Emacs Lisp [Elisp]
        - Emacs Lisp is an ***extension language*** for Emacs - is not meant to be the primary driver of functionality (generally forgoes a main function, e.g.), used instead for small additions/changes to the main application [Emacs]
            - Similar to Chrome extensions, for Chrome
- Lisp scripting uses an object-oriented approach:
    - Lisp objects represented by [64-bit] pointers to values in memory
        - Done so that all sorts of values (i.e. of any size >=1 machine word) can be accommodated by a single syntax
        - Uses *tag bits* to denote the type of variable specified
            - Primitives - denoted by 3 tag bits at the end of the pointer
                - Put at the end of a pointer since [x86-64] addresses are aligned & byte-addressable (end 3 bits [8] are not used)
                - Exception: fixnums are represented by their actual values, rather than through pointers [end 3 tag bits indicate such]
                    - More memory-efficient
            - Other objects - denoted by tag bits at the end of the pointer (denoting a non-primitive) + tag bits at the head of the value (denoting the actual type)
        - Note: Javascript uses a similar approach
- Lisp object types:
    - Primitives: Numbers, strings, etc.
        - Numbers: fixnums (small integers - fit in a single machine word), bignums (larger integers), floats
        - Chars are essentially just an alternate notation for integers (?a returns ASCII integer encoding 97), rather than a distinct type
    - User interaction: Buffers, windows, frames, terminals

- Shell: Processes
- Atom (symbol)
  - Represents a single data point, equal [eq] only to itself
    - Any given name can only be associated with exactly zero or one atoms; thus, any two references to a given atom name will always be to the same object in memory
      - Implemented internally via hash table
    - Can be used to represent "concepts" in the context of AI, e.g.
  - Ex: t, nil
- Lists constructed via cons [construct]: (cons val1 val2) -> (val1 , val2)
  - car, cdr - return first and second values of a pair, respectively
    - If the second value of a pair is a pointer to remaining list elements, cdr will return all remaining list elements
    - Naming - historical IBM convention
  - nil stands for empty list
  - Lists can then be made out of value-ptr pairs (a la a linked list), nesting cons
    - nil as the pointer stands for end of list/no value
  - Can also be constructed via (list *values)
- Other notable types:
  - Hash tables (constructed as an empty table, then filled - keys of any type)
  - Char table (hash table that only take chars as keys - more efficient)
  - Markers (moveable pointer into buffer, moves along with contents of buffer when changes are made)
    - E.g. "Hello *mrkr*world" -> "Hello, *mrkr*world" [marker moves when string changes]
    - Can also integer index into buffers - does not move
  - Buffer - contains copy of the contents of a file
  - Frame, window - specifies display (frame - overall region, window - subregion)
  - Terminal
  - Process
  - Function: ( defun function_name (args) body )
    - Function call: ( function_name args )

        - Lambda expressions: (lambda (x) (x+1)) defines function -> call: ( (lambda (x) (x+1)) 27 ) [functions do not require names]
- Lisp arithmetic operations, comparisons, etc. utilize prefix notation
    - Ex:. (+ 10 5), (= 3  8)
        - Negative/positive infinity: (/ 1 -0.0), (/ 1 +0.0)
            - -0.0, +0.0 are numerically distinct values
        - NaN (/ 0.0 0.0) does not numerically equal any number, not even itself
    - = compares object numerical values; eq compares object bit patterns (as opposed to object values)
        - eq fails to compare numerical equality for pointer-represented numbers (i.e. only works for fixnums)
            - (eq *immediate* *immediate*) will construct and compare two different objects
        - "Equal" - general object value comparison
    - (setq *variable name* *value*) sets a variable
    - t [true] vs nil [false]
- Control structures [Emacs: special forms]
    - Are also called special forms - resemble functions, but are not so
        - Does not need to explicitly evaluate all the provided arguments before executing, e.g.

**Lisp control structures**

- Conditionals: (if (= a b) [then] (cons a b) [else] (cdr b))
    - Else part can be written as a sequence of expressions, to be executed in sequence, or zero expressions [then part must be a single expression]
    - Shorthand [no else]: (when/unless (condition) (do))
        - do can be a sequence of commands
    - Executing a sequence of commands - value returned is value of last expression
    - (cond ((condition1) expression) ((condition2) expression)...((true) expression)
- Not: (not (expression))
- And: (and (expression1) (expression2)...(expressionN))
    - Stops prematurely if an expression evaluates to False, similar to C++ &&
- Or: (or (expression1) (expression2)...(expressionN))

- Stops prematurely if an expression evaluates to True, similar to C++ ||
- While loops, recursion

**Lisp variables:**

- (setq *varname* *value*) assigns variables
- (let ((*varname* *value*))) declares a local variable
    - Can declare multiple local variables simultaneously: (let ((*var1* *val1*)) ((*var2* *val2))), e.g.
- Functions can have local variables

**Emacs programs**

- Emacs programs are represented as lists
    - Hence: same syntax for lists, functions + args
    - Function calls represented as lists also, though not all lists are valid function calls
    - Apostrophe [quoting] - an apostrophe before a list keeps list as data, rather than evaluating as function call
        - Lists: (let ((x '(3 4 5)))
            - Shorthand for (quote (3 4 5))
        - Quoting a function call, arithmetic expression, variable reference, etc. will store the function name, arithmetic expression, variable reference, etc. as a string
- C-] exits Lisp debugger
- Emacs scratch buffer - used to hold temporary values (C-x b *scratch*)
    - Can also be used to evaluate Lisp expressions (C-j; C-x C-e for general buffers)

**Python**

**History of Python**

- Python initially motivated by a desire to improve computer science education
- Historically [1980s] - BASIC (introduced as teaching language in 1962) was the entry language for programmers, taught in high school
    - Now largely dead, aside from Microsoft
    - Was originally simpler than FORTRAN, but became more complex over time + was difficult to teach [relied on go-tos, e.g.]
        - Practices in BASIC had to be un-taught to teach other languages + proper programming practice (e.g. indentation)
        - Involved teaching a large number of low-level functions
- ABC introduced as replacement to BASIC as a teaching language
    - Made indentation built-into the language
    - Came with an IDE, built-in library for higher-level functions (e.g. sort, hash tables)
    - Failed because it could not be used to get jobs (not enough users)
- Reminder: little languages: mix-and-match different “little languages” (e.g. sh, sed, awk, grep) to accomplish a task
    - Effective, but difficult to teach - requires knowing many languages
    - Attempts made to replace/unify it with a “big language”
        - Perl [scripting language, made by linguist Larry Wall] - “there’s more than one way to do it”
            - Inspired by English also being very flexible
- Python made as a reaction to Perl, descendant to ABC
    - Unified “little languages”, but tried to be more organized than Perl
        - “There is one best way to do it, and Python has it”

**Overview**

- Python not generally used as an extension language, a la Lisp
    - Python may be used as a “glue language” - used for converting data in one form to another, as cleanly/conveniently as possible
        - Relies on calls to higher level functions, which may or may not be implemented in other languages at a lower level
        - Emphasizes OOP very strongly
- Python syntax:
    - Requires indentation
    - Single quotes/double quotes evaluated identically
        - Three single quotes allows for strings to cross line breaks
    - Every entity is an *object*: has an **identity**, a **type**, and a **value**
        - **Identity** roughly corresponds to C++ addresses, but no pointers/references
            - id(object) returns an address as an integer, but can’t be used for anything else (i.e. no pointers)
        - **Type**, **value** stored together at some address
            - type(object) returns another object [of type Object]
        - Identity, type are immutable (address, typefield won’t change)
            - Value mutability depends on object type
- Everything in Python is an object, or rather a reference to an object
    - [Object] copying is just pointer reassignment
- Came with an original 7 built-in types: None [NoneType] (akin to nullptr), int/IntType, Float, Complex, Boolean, List [array], String [array of chars], Tuple [immutable list], Buffer [mutable String, a la buffer vs string in Elisp]
    - 1+2J returns complex number [J instead of i for readability]
        - Number types (int, float, complex, boolean) can be converted between each other, even bool to complex
            - No integer overflows in Python
    - Also contains its own internal types [Code]
- CPython converts Python to bytecode before running, allowing for optimizations

**Sequences (Python)**

- Sequence types (**List** + **Tuple**, **Buffer** + **String**) akin to C++ arrays
    - List and buffer are mutable; tuple and string are not
    - Indexed by integers from 0 to n-1 (n = len(sequence))

<u>**General Sequences**</u>

| Sequence Operations (Python) | |
|---|---|
| **Syntax** | **Result** |
| arr[i] | Returns value at index i |
| arr[-i] | Returns value at index (len(arr) - i + 1) |
| len(arr) | Returns length of arr |
| min(arr) | Find minimum value in arr |
| max(arr) | Finds maximum value in arr |
| list(arr) | Constructs new list from sequence elements |

<u>**Mutable Sequences (List, Buffer)**</u>

| Mutable Sequence Operations (Python) | |
|---|---|
| **Syntax** | **Result** |
| arr[i] = v | Sets value at index i to v |
| arr[i:] = arr2 | Splices in arr2 at index i |
| del arr [i] | Deletes value at index i |
| del arr[i:j] | Deletes value at indices i through j (incl. i) |
| arr.append(val) | Adds copy of val to end of the array |
| arr.extend(seq) | Appends elements from a sequence |
| arr.count(x) | Counts instances of x in arr |
| arr.index(x) | Finds first instance of x in arr |
| arr.insert(i, v) | Inserts v before $i^{th}$ element of arr |

| | |
|---|---|
| arr.pop(i) | Removes $i^{th}$ element and returns its value |
| arr.pop() | Equivalent to arr.pop(-1) |
| arr.remove(x) | Equivalent to arr.pop(arr.index(x)) |
| arr.reverse() | Reverses the elements of arr |
| arr.sort() | Sorts the elements of arr in-place |

**Lists**

- Lists implemented as an object with values: pointer to vector head in memory, length, alloc
    - *Length* counts current # of elements; *alloc* counts total allocated size of vector
        - Appending increments length; when length == alloc [list is full], will copy the list to another allocated vector with double the size
    - List operation performance
        - Append: O(1), amortized [averages out to O(1)]
            - Math of amortization: half the list cost 1 operation, a quarter cost 2 operations (copied once), an eighth 3, etc. ($(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots) = 2n$)
- Python List akin to a stack (push=append, pop)
- arr.sort(): Numbers [non-complex] compared directly, strings via string comparison, custom types via custom operator<

**Strings**

| String Operations (Python) | |
|---|---|
| **Syntax** | **Result** |
| str = “this is a string” | Creates a new string |
| s1.join([s2, s3, …]) | Returns a string with value: s2 + s1 + s3 + s1 + … |
| str.split(separator) | Divides str into an array, using separator as the delimiter |
| str.replace(old, new) | Replaces every instance of old in str, with new |

- Strings are immutable - mutable operations don’t apply
    - Operations work by constructing new strings, rather than overwriting the old one

**Mappings (Python)**

- Python mappings (notably Dictionary/dict) implemented internally via hash tables
    - dict[key] performs a hash lookup in dict with key
        - Key must be immutable (no lists, dictionaries, e.g.)
        - Using variables as keys will use the variable value as key
- Notes on dictionary operations
    - Using [] to index/delete on a nonexistent key returns a KeyError
    - dict.update(dict2) will overwrite any preexisting item in dict overlapping with dict2

| **Dictionary Operations (Python)** | |
|---|---|
| **Command** | **Result** |
| dict = {key: value, …} | Creates new dictionary, mapping keys to values |
| dict[key] | Returns value associated with key in dict |
| dict.get(key) | Returns dict[key] if key is valid, None if not |
| dict.get(key, val) | Returns dict[key] if key is valid, val if not |
| dict[key] = val | Maps key to val in dict (overwriting any prior mapping with key) |
| dict.has_key(key) | Looks up if key is in dict |
| del dict[key] | Deletes key-value pair from dict |
| len(dict) | Returns number of key-value pair |
| dict.keys() | Returns list of keys in dict |
| dict.values() | Returns list of values in dict |
| dict.items() | Returns list of key-value pairs as an array of tuples |
| dict.clear() | Removes all items from dict |
| dict.update(other_dict) | Copies every key-value pair in other_dict, to dict |
| dict.popitem() | Deletes some item and returns the corresponding key-item pair |

**Callables (Python)**

- Callables: Function, Method [class function], Class [calling Class akin to C++ constructor]
- Functions
    - Can be called with arguments in any order by specifying the argument called (e.g. f(arg2=val, arg1=val2, ...))
    - Can take in trailing arguments (e.g. f(a, b, *c) will return any trailing arguments not a, b in a tuple as c)
    - Can take in keyword arguments (e.g. f(a, b, **d), then calling f(a, b, key1=val1, key2=val2) will return d={key1: val1, key2=val2})
        - Can be used in conjunction with trailing arguments
- Classes
    - Syntax: class(*parent classes*)
        - Allows multiple inheritance
            - Conflicts: Inherits closest (depth-first, left-to-right)
            - Later parent classes "fills in" gaps of first parent class [effectively superclass]
    - Can put any number of methods, variables in definition
        - Is dynamic - can be changed during runtime
        - myClass.__dict__; instance variable matches names [of functions/methods] associated with class, to values
            - Can be modified, e.g. for use in *meta-programming* (building upon classes made by other people/libraries)
                - Only works in Python2; Python3 changed __dict__ to read-only MappingProxy (only string keys -> faster)
        - Defining methods with underscores defines built-in methods/operations
            - myClass.__init__: executed on object creation
            - myClass.__str__ defines the return of casting obj to string
            - myClass.__cmp__(self, other) comparison operator
                - -1 for <, 0 for =, 1 for >
                - Used for list.sort(), e.g.
                - myClass.__lt__ [less than]
            - myClass.__nonzero__

            - Used for if object: checks
        - myClass.__add__(self, other)
    - Class methods - first argument [self] is generally the instance for which the method is being called
        - Does not need to be called self
    - Can rename classes (e.g. class c; d = c)
    - Immutable classes can be made dictionary keys
        - def __hash__(self): define hash function returning integer

misc

    - Documentation puts square brackets around optional arguments
- Python string processing
    - Line = “str,10,10.2”; types = [str, int, float]; fields = [t(v) for t,v in zip(types, line.split(‘,’)) -> [(str, “str”), (int, ‘10’), (float, ‘10.2’)] -> fields = [‘str’, 10, 10.2]

**Client-Server/JavaScript**

- Node/React - built on JS [+POSIX+OS[+sh as a configuration language]]
    - Quoting issues, configuration in React
- Client-server system: clients and servers [separate computers] communicate over/through a network [e.g. the cloud]
    - Clients send queries to the server; server sends information back
    - (Other models exist for distributed computing)
        - Ex: Multiple clients, one server
            - Clients communicate only via the one server [may cause scaling issues with more clients]
        - Multiple clients, multiple servers
            - Servers are closely connected, can spread request load
        - Peer-to-peer: each peer [computer] maintains part of the state of the system, communicate with other peers via network
            - No single server; spreads server responsibilities across peers
        - Divide server into primary and secondary nodes: secondary nodes receive questions, send to primary node (primary node returns information)
            - Used in machine learning, e.g.
- Performance issues in distributed computing
    - *Throughput* - how much work [queries per second] is the overall system doing?
        - Primarily a server-side issue
    - *Latency* - how long is the delay between a request and a response?
        - Primarily a client-side issue
    - Balancing throughput vs latency - can improve one at the cost of the other
        - Batch requests (out-of-order execution) - can increase throughput by collecting requests until a number of sufficient related requests [querying same information] are received, then performing them all at once
            - Increases efficiency, since the information is in the RAM for each request; but hurts latency
        - Caching - Can decrease latency by client-side caching and reusing information received from past queries
            - Forgoes having to send a request and wait for a server response

- Stale caches - what happens when the information in the cache is out-of-date?
    - Cache validation - looking for a means of checking the validity of a cache, cheaply
- Preloading/prefetching - caching information a client has not requested, in the expectation they will request it soon
    - Aggressive caching

**The Internet**

- The Internet & alternatives:
  - Internet precursor: telephone system [landlines] operated on circuit switching
    - Many wires [individual phones] are connected to central offices; central offices communicate via long series of switches
      - Switch paths can traverse long distances geographically
      - A single set of wires is limited to a single connection (wires are reserved)
    - Cons: inefficient, no fallback in the event of failure between switches
      - 1960s [Cold War] - US military concerned about the possibility of attacks on US communication system
- Paul Baran [RAND corporation, ~1961] proposed ***packet switching***: conversations broken up into small pieces [packets, e.g. 1 KiB], each packet routed dynamically between central offices [rather than following a preplanned route]
  - Any two packets with the same origin & destination may take entirely different paths – no reservation of wires
  - Packet composed of two parts - ***header*** (a la metadata) and ***payload***
  - More robust in the presence of an unreliable network, more efficient [multiple connections can use the same wires]
    - Cons: worse latency [dynamic routing takes time, since each switch must determine next switch], data may not always be received in order of transmission
  - Used by Zoom, modern phones, e.g.
  - 3 problems for packet switching:
    - Packets may be received out-of-order
    - Packets can be lost during transmission
    - Packets can be duplicated
      - Usually due to network issues (e.g. a switch thinks a send failed, and sends again)
- Packet switching issues dealt with using ***protocols*** - communication between switches to handle the transmission of packets
  - Can be complicated

    - Complexity dealt with using layers of protcols
- *Protocol layers [high-level to low-level]*
    - ***Application layer*** - application-dependent series of protocols
        - Ex: World Wide Web [WWW] allows for asking for, receiving webpages; File Transfer Protocol [FTP] for transferring large files
    - ***Transport layer*** - views packets as a stream of data from origin to destination
        - Has protocols attempting to resolve 3 issues
        - Ignores low-level implementation
        - Intended to replicate circuit switching functionality
    - ***Internet layer*** - switches [routers] determine pathing from origin to destination through individual switches
        - Based off Internet protocol [IP]: packet header contains information about packet, what to do with it
    - ***Link layer*** - low-level [hardware-level], point-to-point communication between switches
        - Only deals with packets - not packet switching/routing
- Versions of Internet Protocol (IP):
    - **IPv4** (1983)
        - Team incl. J. Pastel (UCLA)
        - Specified contents of packet and packet header
            - Packet header contains:
                - Length of packet
                - Protocol number, indicating # of high-level protocol to use
                    - Intended to use IP as base layer for higher layers
                - Source, destination addresses
                    - Represented as 32-bit ints [4 decimal bytes each]
                    - Source address can be used to communicate transmission failure, e.g.
                - Checksum [16-bit] - used to check for bit errors in transmission
                - Time-to-live (TTL) [8-bit] - tells the router to drop the packet after a certain number of routers are traversed
                    - Decremented for each router traversed

- Used to prevent infinite router loops
    - Abuses/problems:
      - Source address can be faked by sender
      - Nonexistent destination addresses
  - **IPv6** (1998):
    - Made addresses larger + other changes
  - Many networks may involve both IPv4 and IPv6 machines, use gateways to convert IPv6 addresses to IPv4
    - Loses IPv6 features
- 2 major categories of transport layer protocols: ***User Datagram Protocol*** (**UDP**), ***Transmission Control Protocol*** (**TCP**, **TCP/IP**)
  - **UDP** (David Reed, MIT) forms a very thin layer atop IP
    - Easy to implement, but vulnerable to packet loss
      - App is responsible for dealing with packet loss, duplication, etc.
  - **TCP** (Vint Cert, [UCLA alum] Stanford; Bob Kohn, Princeton) provides reliable, ordered, error-checked stream of information
    - Divides data stream into packets [negotiating packet size with hardware]
    - Also handles retransmission, in the event of dropped packets; reassembly [reordering], in the event of out-or-order packet transmission
    - Handles flow control - avoids problem of greedy senders [senders shipping too much data to local central office, to be handled by global network]
- Application layer: WWW uses ***HyperText Transfer Protocol*** (**HTTP**), ***Real-Time Protocol*** (**RTP**)
  - Also uses other protocols (e.g. video protocols)
  - Defines protocols for classes of similar/related applications (e.g. browsers)
  - **WWW** and **HTTP** developed by Tim Berners-Lee, CERN (1991) to handle displaying of, and linking between [footnotes], digital copies of physics papers
    - Implemented via combination of **TCP**-like protocol (handling transfers) + **SGML** (data protocol) + new linking protocol
      - ***Standard Generalized Markup Language*** (**SGML**) developed as standard interchange language between publishers of books/papers
        - Had HTML-like syntax
- **HTTP** (original main Internet protocol) written as a client-server protocol for transfers, similar to TCP

- Operates in terms of requests and responses between client, server
    - Example request: GET /index.html HTTP/1.1 \n\n
    - Server response: Header + *webpage contents*
        - Header: 200 OK + *webpage metadata* \n
        - Metadata: length, content typefield (e.g. HTML; used to aid in client interpretation), content transfer encoding (e.g. UTF-8)
- HTTP 1: client setup, one request, one response, connection drops [TCP teardown]
    - Eventually evolved to handle multiple request/responses (HTTP 1.1)
- HTTP/2 (2015): made various changes for efficiency
    - Compressed headers (client-server had to agree on compression format)
        - Before HTTP/2: headers not compressed
    - Allowed for *server pushing* - server can send a response, even if client has not sent a request
    - *Pipelining* - connections kept alive after server response (no teardown) + client can issue multiple requests at once without waiting for a response
        - Issues: browser now has to deal with out-of-order responses, e.g. by labeling requests and responses
    - Multiplexing - using a single TCP connection for multiple conversations
        - Used in the case that a single client wants multiple sessions, e.g.
            - Requests include host/domain name/session of interest

- HTTP/individual TCP connections layered atop data streams [TCP] atop packets [IP]
    - Results in "*head of the line problem*" - TCP trying to preserve chronological order of sent packets means, in the case of packets not being sent/dropping, any later packets will not be received by the browser
    - HTTP/3 (2022): replaces TCP with QUIC protocol to resolve head-of-the-line
        - QUIC ("TCP v2 + UDP") - attempts to preserve chronological order of sent packets (a la TCP), but has ways to deal with packet dropping
        - Allowed for further multiplexing

**HTML**

- HTML derived from publishing format SGML

- Philosophy [SGML, sorta HTML] - the language should contain only the contents of the book, not any form of program to actually run them/display them
  - "Declarative, not imperative" - does not tell the browser what to do (e.g. styling), only what they should display (content)
  - Makes porting easier, since each browser may have its own command set
- SGML philosophy inherited (to some extent) by HTML

- Terminology of HTML
  - HTML elements consist of an opening tag (enclosed in <*tag name*>) and closing tag (enclosed in </*tag name*>), with text in between
    - Some *void elements* consist of only an opening tag (e.g. <br/>)
    - Opening tags may have attributes: <*tag name* *attributes*>
      - Attributes consist of name-value pairs of text
        - Names are distinct identifiers; values can be any string
      - Attributes may describe to a browser how an element should be displayed, or denote relationships between HTML elements
    - Elements can generally nest inside each other, in a hierarchical structure
      - Exception: raw text elements contain only text, paragraph elements cannot nest directly inside other paragraph elements
      - Elements are nodes of the tree; internal text are the leaves
  - HTML documents (also called resources) can point to other documents via links
    - <a> [anchor] has href [HTML reference] attribute, holds a URL (uniform resource locator) to another document
      - #*id* at end of URL - move user cursor to element with that ID
        - In the absence of a domain name (e.g. just href="#intro"), defaults to URL of current webpage
- HTML problems:
  - Possible errors in HTML:
    - Element lacks a closing tag
      - General mantra: be strict in what is generated, but be generous in what you accept
      - If adding a closing tag would fix error, add a closing tag
        - Omitting closing tags on the developer side can be done to reduce overhead, if there is no ambiguity

            - Element has an extra closing tag: things break
- Possible upgrades for HTML
    - Extending/changing HTML (e.g. by adding an attribute) to allow for additional functionality
        - Consequence: servers and clients may update at different times and potentially have slightly different versions of HTML
            - Servers on newer versions may send HTML that clients on older versions cannot parse
            - Methods for resolving:
                - Clients, upon seeing HTML they cannot parse, will make something up
                - Clients tell server the kind of HTML they can handle
                    - Server tries to use specified HTML format; get more complicated
                - Server tells client the version of HTML in response; client figures out what to do
                    - Clients have to support more HTML versions
                        - HTML elements may change, be removed between versions (e.g.)
                    - Used in the early web
    - HTML version denoted by Document Type Definition (DTD/Doctype)
        - Taken from SGML - different types of documents, publications may be shared, each having different needs, specified by Doctype
            - Doctype lists all elements + their allowed attributes, subelements, void-ness, raw text-ness, etc.
        - HTML Doctype also specifies whether closing tags are optional
        - Doctype standardization effort ultimately failed
            - Web evolved too rapidly - Doctype standards came too late
            - HTML became too complicated to express in DTD
    - HTML/5 [current version] - uses evolving standard in place of a static DTD
        - Commits are made to a GitHub repository
    - XML (eXtensible Meta Language) - barebones for HTML syntax, with no semantics
        - Simple way to send any form of tree-structured data

- Leverages browser infrastructure for parsing HTML, as a means of transmitting other data (e.g. payroll)
  - Element tag names, attr names, etc. all ignored
    - Insists on closing brackets
  - XHTML - “strict” HTML
    - Is valid XML - insists on closing brackets, e.g.

- ***Document object model*** (**DOM**) - HTML ships elements within an tree data structure, where nodes are elements (w/ attributes), leaves are text
  - **DOM** - object-oriented model for examining/manipulating said trees
    - Primarily used for JavaScript; theoretically intended for any OO language
    - Attempt at making HTML more imperative (rather than declarative)
      - DOM itself is not imperative, but using JavaScript makes it so
- ***Cascading Style Sheets*** (**CSS**) - separate HTML declarations into two categories: content (HTML) and presentation (appearance on the screen; CSS)
  - Presentation can be specified by user preferences, the element involved (e.g. paragraph attributes), ancestor elements, etc.
    - CSS provides a standard for combining preferences/styles
- Declarative philosophy is limited in its ability to handle more complicated interactions
  - Done using imperative technologies, namely code in JavaScript
- ***Javascript*** - programming language designed specifically for use in browsers
  - Is directly embedded into and run by web browsers
  - Can be hooked into HTML via the <script> tag
    - Can either import external files or write Javascript directly
  - Can modify elements in the DOM

- Programming paradigms - classifications of programming languages based on their features/functionalities
    - Methodologies/strategies languages follow, and the capabilities they have (can have multiple in one)
    - Can be classified by organization of code, style, syntax, grammar
- Web app vs website: web app more complex (interactive, integrated, authentication)
    - Interaction - self-explanatory
    - Integration - bringing together different components
    - Authentication - self-explanatory
- Layers:
    - Frontend - user-side interface
    - Backend - services & presentation logic; returns responses
    - Backend (database) - store & serve data

**Version Control**

- Configuration control - ensuring deployed software is properly setup (ops)
- Version control - (dev)
  - Tracks version history (history - versions)
    - Allows for reversion, version disambiguation, compatibility, ability to see past changes
  - Allows for working in different directions + changes in different areas of project
  - Bug reports
  - Merging projects
  - Project reviewers - reviewers can look at changes made, e.g.
- Similarities - record/develop variants of a single program
- Git - developed for Linux kernel
  - Contains object database recording variants
    - Commit object - records project state + metadata (time, author, etc.)
  - Indexes - records planned changes for next commit, etc.
    - Used in the context of day-to-day changes
  - Is a distributed database - multiple copies of the same project database can exist, and can differ; none are strictly "in charge"
- Info
  - Remote: # of objects received = # of objects in database (roughly)
  - Change history stored as a set of changes from the previous version
    - During initialization - must compute all changes from oldest to newest version
- Commands
  - Git init - initializes empty git repo [.git]
  - Git clone *repository link*
    - Creates copy of source files + copy of repository history (in .git) (40%/60% of space, possibly)
  - Git status - tells status of current repository, working project
    - Indicates current branch, sync status with upstream branch, staged & unstaged & untracked changes [for commit]
    - Finding current project state

- Git log - displays metainfo
  - Commit ID (40 hexadecimal digits), author, date, commit message
    - Commit ID associated with only one commit in the entirety of git
      - Computed as a cryptographically secure checksum of commit contents [SHA-1] - difficult to find a match with same checksum
    - Commit message human-generated: important for reviewers, other developers, etc.
      - Format convention: 1st line - key, 2nd line - empty, other lines - describe motivations for change
        - Other lines also include API changes
    - Can be identified within repositories using a prefix set of digits (initial 12, e.g.)
  - Git log *commit ID* - view status at certain commit
  - Git log *commitID1…commit ID2* - view entire history from commit1 (exclusive) to commit2 (inclusive)
  - Git log –pretty=*option* changes formatting
  - Git log *commit*@{num} - state of repository num changes before commit
- Git ls-files - lists all files under git control [working files] in current version (can be grep'd, e.g.)
  - Git grep *pattern* effectively greps through files in git ls-files
- Git diff - compares current state of working files vs state of files in repository
  - Git diff compares current working files vs plans for commit
    - Adding files via git add -> will no longer appear in diff
    - Option: –cached compares files in index to most recent commit
  - Git diff *commit1* *commit2* - compares files in different repository versions (omit commit2 -> compare to current state of working files)
  - Options: -w (ignore whitespace-only changes)
  - Git apply *patch*
- Git add *FILES* - stages FILES for next commit [adds to index]
- Git checkout
  - Can checkout individual files from current version - git checkout *path/filename*

- Git commit - enters editor for writing commit message; creates commit upon exiting editor
    - Copies index into current repository; does not necessarily change files
    - Projects can setup style rules for commits (e.g. message 1st line must contain colon [for describing project component, e.g.])
    - Options -a (add all files), -m (specify commit message within shell command)
- Git configurations
    - Options: -l (list all command variables)
        - Comes from/stored into: current git installation, project's repo [.git/config], personal preferences
    - Must be properly formatted
- Git show *commit* - similar to git log, for just the single commit
    - Git show *commit:FILE*
- Git reset reverts to HEAD, discarding all changes
    - Git reset –hard *commit* - reverts to said commit
- Git blame FILE - shows who last edited each line of the file
    - Git blame COMMIT -- FILE
    - Struggles with nonlinear history (e.g. in the case of merges)
- Git bisect - binary search to find specific bug-introducing version
    - Git bisect start
        - Git bisect bad; git checkout *good*; git bisect good
            - Git bisect bad *commit ID*
        - Git bisect start *bad* *good*
        - Git bisect run *any shell command*
            - Git bisect skip
    - Difficult for multiple branches
        - Git wants both maximal node (most # of ancestors), and minimal node (farthest away from HEAD) -> picks compromise
            - Has option for randomization
            - N = # ancestors; M = largest N in graph; bisect: max(N, M-N)
        - Can mark bad nodes to skip (e.g. broken versions)

        - Sporadic test – usually report good in the case of a working program/bad in the case of a non-working program, but sometimes fails to report the correct result
            - May result from performance testing/race conditions (tests dependent on many external variables)
    - Git checkout
        - Git checkout COMMIT - changes working files to match commit
    - Git switch - change branch [alt: git checkout -b]
    - Git cat-file -p *hash* – view contents associated with hash
        - -p [print contents], -t [print type]
        - HEAD^{tree} – tree associated with head [e.g.]
    - Git hash –stdin – generate unique ID based on particular string
        - Can take stdin, file as input
        - ID generated will be identical for the same string
        - Is mostly cryptographically secure checksum
            - Was originally used for verifying successful transmission of a file
            - Relatively difficult to generate two strings making the same checksum (roughly proportional to 2^*key length* (in bits) -> $2^{160}$)
            - Is meant to be generally collision-free
        - Git hash -w – write object into repository (.git/objects), as a blob
    - Git update-index – builds new tree from planned changes
        - Update-index –add –cacheinfo "permissions,hash,name" creates tree with blob in index
        - Git write-tree – copies index into repository [gives checksum for new tree]
- Git commands - remote servers
    - Git clone *URL*
        - Repository hosted on a git server; copied to local machine
            - Cloning local repository - .git directory of newer clone will use hard links to .git directory of older clone [saves space]
    - Git fetch - downloads data from remote repository without making changes to working files
        - Local repository stays on same commit, same index

  - Git pull downloads from remote repository (a la git fetch), but merges changes into commits - changes working files
  - Git remote - lists name of all remote, fetchable repositories
    - Default (upon cloning): origin
    - Git remote -v
- Branches allow for multiple versions of a project
  - Different teams may be working on different features independently
    - Ex: main (development) + feature branches
  - May have different tolerances for new features, depending on purpose
    - Ex: maint (bug fixes to previous versions) & old (older version of maint - only security fixes)
  - Commands
    - Merging two branches creates a merge commit - commit with two parent commits (combines changes in both branches)
    - Git checkout *branch* switches to a new branch - attaches commits to new branch
- Other
  - .gitignore - file distinguishing between files to be version-controlled, files not to be version-controlled
    - Contains set/list of patterns (similar to shell globbing patterns) to ignore
      - Pattern starting with slash - only matches files in current directory
      - Pattern starting without slash - matches in working directory & subdirectories
      - Pattern ending with slash - match only directories
    - Can be used to ignore build files, private files, e.g.
  - Commit arithmetic:
    - HEAD - latest commit
    - ^ after commit ID = preceding commit (HEAD^^ = commit two commits before latest commit)
- Git tags – named Git object pointing to a commit
  - Syntax: git tag -a *tagname commit* [optional: -m *tagmsg*]
    - git tag [show tags], git tag -l ‘*globbingpattern*’
  - Can be used to mark important commits or signify milestones, e.g.

        - Are shared when the repo is cloned
        - Are designed to be immutable – cannot be moved to a different commit, e.g.
    - Using a tag to a commit in git commands equivalent to using the commit hash
- Git branches –
    - Are essentially just lightweight, movable pointers to a commit
        - Will move as development occurs, commits are made
    - Making commits on a branch will sprout off a separate branch of a commit tree
        - Branches can sprout others branches; can have many branches
    - Motivations for branches:
        - Can have different feature branches – lets different features be developed in isolation until the feature is ready
            - Can change the same things other people are changing
        - Can have maintenance branches – separate branches from "mainline" development branches, usually older, more stable versions
            - Receive only bug fixes, i.e.
            - Security branches – even older, receive only security fixes
            - Drawback: may have to copy the same fix across multiple branches
        - Can fork – separate a piece of software into two separate versions, developed entirely independently
            - No attempts made to sync forks
            - May be used in the case of disagreements between developers regarding future directions, e.g.
    - Syntax: git branch b *commithash* creates branch at that commit
        - Alt: git checkout -b *branchname commithash*
            - git branch --track *branchname upstream*
            - git checkout -f *branchname* (switch branch, discarding current changes to working files)
            - Moves HEAD to the commit where branch was started
        - git branch [lists all branches]; git tag -l '*globbingpattern*'
        - git branch -d *branchname* [deletes a branch]
            - Deletes only the pointer to branch commits; actual commit objects are not initially affected [but may become orphaned]
                - -D – force delete [no warnings]

            - Git garbage collector will eventually kill orphans (git gc)
        - git branch -m *branchold branchnew* [renames a branch]
            - Can rename branches to have different names to avoid conflicts with upstream branches, e.g.
        - git diff *branch1 branch2* [compare two branches]
- Git
    - Merging – move the changes made in one branch into another ("combine")
        - Creates a merge commit in the merging-into branch – commit with multiple parents (such that both branches will point to the same commit)
            - Can merge multiple branches simultaneously
        - If the two branches have colliding changes (i.e. made their own different changes to the same lines), cause merge conflicts
            - Can be resolved manually, or automated via Linux *diff3* (compares three files/versions - runs diff twice)
                - Diff3: compares files B, A, C [in that order]
                    - A - oldest file (common ancestor of two branches), B and C - branched-off alternatives
                    - Alt: diff3 MYFILE OLDFILE YOURFILE
                - In the case that two versions are in agreement and one disagrees, diff3 picks disagreeing branch [wants to capture changes]
                    - Diff3 -m [show merged results]
                - diff smart enough to determine what regions have changed, what regions have not [even if differing regions are of different lengths]
                    - Developer of diff later worked for comparing DNA for Human Genomics Project
            - git merge essentially uses diff3 internally
        - git merge may result in a merged version with the wrong behavior even in cases where changes do not directly collide (do not change the same lines)
            - May want to compare both parent commits to new merge commit to see if the changes made sense (git diff branch1 merged; git diff branch2 merged; [git diff ancestor merged])

    - Rebasing – essentially "copying" the changes made in one branch (from the common ancestor) to another branch, such that the rebasing branch now appears to have been branched off of *otherbranch*/HEAD
        - Theoretically results in the same changes as a merge, but results in different-looking git history
        - (In rebasing branch) git rebase *branchname* [rebases changes made between common ancestor to head of rebasing branch, to the head of the branch named in the rebase command]
            - git rebase -i *branchname* [selectively choose which commits to apply to rebased branch, in what order, etc. - may risk collisions]
                - Can even rebase a branch on itself (from an older commit) to throw away future commits as desired
            - git rebase –ignore-date [changes dates of rebased commits to appear to have occurred on the date on which the rebase occurred, rather than when the original commits were made]
                - Otherwise [default]: rebase will keep original commit dates, even if that results in what appear to be out-of-order commits (i.e. rebased commits older than main commits)
            - Rebasing other onto main: essentially changes other/HEAD, from branching off an older commit in main, to appear to branch off main/HEAD instead [and have all the changes made in main in the interim period]
        - Sort of "rewrites history"
            - Motivation:
                - Simplifies git history (makes history appear to be a single line of sequential commits, rather than many merges)
                    - May want to rebase a branch against current master HEAD to show only the changes actually made by commits in the branch (i.e. won't also show "differences" caused by commits in master)
- Git stash – temporarily saves current set of changes to working files in the index
    - git stash push (saves current changes to index)
        - git stash list [show stashes]

        - git stash apply [apply stashed changes]
    - Can create multiple stashes
    - Can be used in the case of context switching (working on multiple branches)
    - Alternative: git diff >changes.diff (stores differences in file), patch -p1 <changes.diff (re-performs changes based on file)
- Version control philosophies
    - Git philosophy - only important files should be committed
        - Automatically computed files ignored, e.g.
    - Alternative - save everything
- Aside on git bisect - help bisect out
    - Make changes small and self-contained – break large commits (touching multiple source commits) into smaller ones before publishing
        - Makes changes more easily analyzed for bugs via git bisect, e.g.
        - Don't make any commits that break things (break the build)
- Aside on git status
    - Generally clean, but can become messy in the case of a merge/rebase/bisect, e.g.
- Git submodules – ways of combining different projects
    - Can make projects depend on one another to build/run, e.g.
        - Places dependency source code as submodule within depending directory; allows depending directory to access dependency's source code
    - Syntax: git submodule add *link to repo* [from within depending git directory]
        - Adds new type of object to depending directory's .git – submodule object, pointing to commit in dependency directory [dependency version]
            - Dependency's .git contents unaffected
    - Case: dependency project gets an update
        - Can choose to do nothing (continue on old version; insulated from updates)
        - Can alternatively update to newer version
            - Syntax: git submodule foreach git pull origin/main [e.g.]
                - foreach – run command for every submodule
                - Can just run git pull inside dependency directory instead
            - Need to then add submodule update to git commit: git add *submodule_name*
                - Updates only submodule object of depending directory

- Cloning a directory with submodules will not clone submodules automatically
  - Would need to explicitly initialize submodules: git submodule init
- Drawbacks – relies on close dependence on other people's repositories/projects
  - Cannot make own changes to submodules; only rely on others for fixes
  - May prefer to use higher-level importing tools (e.g. PyPi) instead, which assume the dependency is installed elsewhere
  - Submodules may be used to avoid needs for install dependency elsewhere (i.e. putting burden of dependency management on dev, not the customer)
    - Used more commonly in embedded systems without robust package managers, e.g.

**Git Internals**

- Git internals: porcelain [Git commands] vs plumbing [internals]
  - Cloning only involves copying a .git folder (faster); updating files involves resolving changes to get files to their current version (slower)
  - Contents of .git subdirectory:
    - .git/branches/ – obsolete, just kept for backwards compatibility
      - Formerly – listed branches [had small files describing branches]
    - .git/config – file describing repository configuration
      - Describes repository format, various settings
        - Bare – bare repository does not contain working files, only the .git repository [git ignores source files]
          - Only used for cloning/viewing history
      - Contains remote origin (where repo was cloned from, where to fetch from) + info about local branches + corresponding remotes, where to merge from
        - Transfer - determines how much checking git should be on pushes/pulls to/from upstream
      - Can be modified directly via text editor or git config command
        - git config *variable name* *value*
    - .git/description – obsolete, would be used by other applications (e.g. GitWeb) to describe repository; replaced by READMEs

            - .git/HEAD – lists current location
                - Gives a branch, or a checksum if not on a branch
            - .git/hooks/ – contains scripts + sample scripts written by developers
                - Hooks - scripts executed by scripts at various points of operation
                    - .sample files [sample scripts] ignored by git, just indicate nature of corresponding hooks
                    - Maintained manually by developers (e.g. created in scripts), not automatically by Git
                    - Ex: script to verify commit message formatting
                        - Commit messages, pre-commit, pre-push
            - .git/index – data structure [binary file] representing index ("future plans")
                - Filled in by git when files are added to index by git add, e.g.
            - .git/info/ –
                - info/exclude – file under Git control, detailing files to ignore
            - .git/logs/ – keep track of changes to branches
                - /logs/HEAD – file containing history of all commits/changes to current repository
                    - Is a metahistory: Tracks what was changed, by who, when, via what command
                - /logs/reflogs - temporary auxiliary file tracking changes by the user to the local git repository
                    - Expires after 90 days [changeable]; not shared with others
            - .git/objects/ – stores all Git objects
                - Git repo state stored via object-oriented database; each object associated with unique ID (40-digit hexadecimal)
                    - Stored in directories: .git/objects/*first 2 hex digits of ID*/*remaining digits of ID*
            - .git/refs/ – stores pointers to commit objects (branches, tags, remotes, etc.)
                - Stores branch heads
- **Git objects** – content for database
    - Objects named by their contents via hash; same contents -> same hash
        - Object naming divorced from the file system file [node] #s; means knowing contents of a git object -> knowing name/hash of git object

            - File #s map names to contents, but not vice versa
    - Ensures atomic updates – when pulling from git, multiple files are changed "simultaneously", e.g. [no intermediate state with half old, half new]
        - In a regular file system - would require two copies of the directory [old and new], re-pointing of symbolic link from old to new version
    - Generally meant to be immutable
- *Types of git objects:*
    - **Blob** - chunk of binary data, storing file contents/data [generally a file]
        - Does not store attributes, file name (just contents)
        - Is entirely defined by its data - same hashed contents create same blob, same SHA hash
        - Blob permissions (six digits): first two digits = type [directory vs file, e.g.], last three digits = three digits of chmod [user, group, other]
        - Blob - low-level representation of a git object
            - Git representation - contains "blob", null byte, contents of blob, newline; compressed via gzip algorithm into a blob
                - Gzip *stdin* compresses stdin; gzip -d decompresses
            - Source code generally fairly compressible
            - Compression used as a space-saver (takes some CPU time for compression/decompression, but saves space on disk + time for reading/writing from/to disk)
    - **Tree** - directory-like (references other trees, blobs]
        - Stores to pointers to blobs, other trees
            - Typically used to represent contents of a directory/subdirectory
        - Maps names to IDs [other git objects - blobs, trees]
    - **Commit** - points to a single tree [represents project state at a certain point in time]
        - Commit object - contains hash of tree [top-level directory of commit - of working files/directories], hashes of parent commits, author, committer, commit message
            - Newer commits point back to older commits (linked list)
            - Author/committer contain names, email addresses, possibly authentication string
        - Does not commit changes from previous commit to current commit

    - Tag - marker for a single commit

**Compression**

- Compression
    - Speed vs compression quality trade-offs
        - More RAM also generally leads to better compression
    - Git compression algorithms: Huffman coding vs dictionary coding
        - Huffman coding - basic, cheap, well-understood
            - Basic problem: how to send a symbol string to a recipient
                - Symbols encoded in bytes; transmitted as bit stream
                    - Common symbols given short bit encodings, shorter than the original byte-length symbol
                    - All other symbols become longer (e.g. 0, -> 100, -> 10101; as long as no bit encoding is a prefix of another, can later be decoded)
                        - Symbol bit encodings transmitted separately as a table of representation [i.e. of encodings]
            - Encoding process performed via binary tree
                - Create binary tree, with leaf nodes being symbols [containing the symbol + its rate of occurrence]
                - While nodes are not all connected: connect two rarest symbols with "supernode" (e.g. "j", "q" -> parent node "j or q")
                - Decoding can then traverse binary tree for each new symbol
            - Huffman – proved tree is optimal under assumptions (finite set of symbols, encoding into bits, known rates of occurrence)
            - May be used as a subroutine by higher-level algorithms
            - Dynamic Huffman coding – approach for Huffman coding in the case that rates of occurrence are not known/not assumed
                - Both compressor, decompressor begin with same, perfectly balanced Huffman tree (assume all characters occur equally, character bit representation = byte representation)
                    - Result: initial characters uncompressed

            - As compressor sends characters, both compressor and decompressor begin to fill in a Huffman tree on the fly based on the rates of occurrence of symbols in sent bits
                - Compressor and decompressor keep their own copies of the tree, but should be the same
                - Huffman trees stored in RAM
            - Less space-efficient than static coding, but more flexible
    - Dictionary coding -
        - Look for common strings of symbols (e.g. words), and find shorter encodings for those strings
            - Assigns integer encodings to each word
                - May be able to get away with 16 bits ($2^{16}$ words) rather than writing out all 48 bits, e.g.
            - Requires more memory for dictionary (word-encoding table), but better compression
        - Dynamic dictionary coding – analogous to dynamic Huffman coding
            - zlib uses a window over the input - sender uses last region before currently-being-compressed region, as a dictionary
                - If a word in the currently-being-compressed region matches one in the window, express as the offset from start of window to old word (integers)
                - Window moved forward as compression progresses
- zlib essentially Huffman-encodes output of dictionary coding

**History of Git**

- Motivation of version control – want to use small repository to record a large history
    - Cheaper than making full copies
- Principle of version control: save just one copy of each unchanged line in history, rather than making a new copy for each version
    - Later versions can obtain the line by looking at previous versions
    - Is an application-specific compression method (application: version control)
        - Uses a bit of CPU time to reduce memory consumption

- Approach: Files stored as an interleaved history – stores all lines that have ever appeared in the lines of the source code
  - When referring to individual copies, auxiliary table can simply refer to whichever lines in the interleaved history, were present in that version (contains line #s)
    - Can use line ranges to save space
    - Retrieving a version simply fetches the line numbers associated with that version in the auxiliary table
  - Downside: cost of version retrieval increases as the size of the history grows
    - Commits relatively expensive – must make new copies of the entire history
  - Used by SECS (proprietary AT&T Unix version control program)
- Revision Control System (RCS) – files + past versions stored in memory as a full copy of the latest version, plus reverse deltas to later versions
  - Cheap to retrieve latest version
  - Versions of a file in other branches accessed via reverse deltas to the common ancestor, then forward deltas to the branch head
    - Deltas stored as deletions/insertions at certain line numbers
      - Two-way deltas exist, but are more expensive
  - Developed by Walty Tichy @ Purdue
    - Drawbacks: couldn't handle multiple files at once
      - CVS: could handle multiple files
- CVS succeeded RCS; SVN (free) succeeded CVS; Bitkeeper (proprietary) combined elements of SVN, SECS
  - Git developed as an open-source alternative to Bitkeeper
- Git – operates on two-way deltas

**Low-Level Programming**

- Importance of low-level programming: ensures performance, reliability
- C - language designed to be compiled + run across platforms, providing low-level access to memory + language constructs readily analogous to machine instructions, all with minimal runtime support
- C to C++ loses:
  - Data types: Classes [OOP, inheritance], structs
  - Language features: Namespace control, overloads, cin/cout
  - Exception handling [C exception library - comparatively clumsy]
  - Memory allocation on heap (just malloc, free)
    - No double frees, just null tests [(ptr)]
      - free(0) - true
- Malloc [ptr=malloc(idk)], void* [generic pointer]
  - malloc(# bytes) vs calloc(num elements)
  - Issues: using before writing, not freeing [leak], double freeing, using after free
  - Malloc: SIZE_MAX, -1, 0 [some object size 0 - allocates 1 byte]
    - realloc(ptr, n) -> reallocates contents to new container size n
      - Truncates if needed
- Misc: C library, kernel: meta programs
  - time *pgm*: real/user/system [CPU] time
  - strace *pgm*: outputs any program system calls [-o *output file*]
  - valgrind *pgm*: looks for suspicious activity (e.g. loads from far up the stack, memory leaks, bad memory accesses, etc.)
    - Runs as an interpreted program, sort of
  - gdb
  - top/ps - lists processes (esp. resource-intensive)

**GCC Flags**

- Security: **-fstack-protector** [stack canaries]/-fstack-noprotector
  - **-mshstk**: Intel x86-64 protector (new)
    - Shadow stack - partial copy of return addresses, stored in *shadow area* of memory [only writable by certain things]

- Security: interpreting random bytes as jump instructions
  - mshstk: allows only jumps to endbranch instructions [new inst]
- Performance: **-O0**, **-O1**, .., **-O3** [optimize: faster, slower compilation, larger memory footprint, less human-readable code, may reveal bugs] (higher # -> more optimization; -O0 = none; -O2: general)
  - **__builtin_unreachable()** [C; C++]: uncallable function (results in undefined behavior; compiler may do anything)
    - Not an instruction in and of itself; indicates to the compiler unintended behavior
  - **-flto** - enables link-time optimization/whole-program optimization (GCC performs another optimization pass during the linking process)
    - Combination of -o [optimization], -f [may change instructions] flags
    - Optimizes each individual compiled file, knowing the contents of the other compiled files (but before linking)
      - May inline calls to short functions in another file, e.g.
    - Compiler may also find more bugs
  - Many optimization techniques are $O(n^2)$
- Performance attributes:
  - **__attribute__((__aligned__(4096)))** - ensures alignment of data with 4096-size pages
    - Formatted for inclusion at start of C program [e.g. # !__GNUC__; #define __attribute__(x); #endif; #_attribute__((__aligned…]
  - **function()__attribute((cold))** - tells compiler a function is rarely called
    - Stores function in memory/RAM - does not bring instructions into instruction cache unless called
      - Instructions [in instruction cache] jump to cold section [somewhere in memory] rarely, as needed
    - In the case of sequential code (i.e. no loops, conditionals) with a cold function, the entire code section will be evaluated as code
  - **function()__attribute((hot))** - tells compiler a function is often called
- Misc misc
  - ptrdiff_t (C) - type denoting differences between pointers

### Debugging

- Debugging (>50% of time):
  - ***Performance debugging*** - making a program smaller/faster
  - ***Correctness debugging*** - fixing incorrect behavior
    - May include performance debugging (e.g. in real-time applications)

### Static Checking

- ***Static checking*** - running a program to examine the program-to-test for correctness, without running the program-to-test [i.e. just looking at the code]
- Has many related compiler options, depending on the needs/types of concerns of a project
  - Notable: **-Wall** [W -> warning, static checks many common cases], **-Wextra** [-Wall + more checks]
    - -Wall: ***-Wcomment*** [checks for possible mistakes in comments], ***-Wparentheses*** [checks for possible mistakes with operator precedence], ***-Waddress*** [always-false pointer comparisons], + more
      - ***-Wstrict_aliasing*** [banned in Linux kernel] - warns against casting between pointers of different types
      - ***-Wmaybe-uninitialized*** - warns against accesses to possibly uninitialized variables
      - Arithmetic operators higher precedence than shifts; && > ||
    - ***-Wextra***: -Wtype-limits [comparisons exceeding max size of data types (i.e. impossible comparisons) - may not be “impossible” for all system]
    - ***-Wno-*warning-type**** removes certain warning types
  - Recent flag: ***-fanalyzer*** (like -Wmaybe-uninitialized, more checks + interprocedural)
    - Catches more errors, but expensive ($O(n^3)$) [should not combine w/ -flto]
- *C static checking:*
  - C: ***static_assert(*some boolean*)*** will only allow a program to be compiled if the boolean evaluates as true
    - Does not generate instructions, but must be able to be evaluated at compile time [constants only]
  - C keywords:

- ***_Noreturn*** declares a function that does not return [to the caller function] (e.g. an exit function) [ex: _Noreturn void exit(int status)]
    - Performed as a jump, rather than call instruction [faster]
    - Can also be used by compiler to improve checks elsewhere (e.g. a branch calling a noreturn means the branch condition will always be false anywhere else
- ***__attribute__((pure, access(readonly, parameter#, #bytesaccessed)))*** - calling a function twice with same arguments, same memory (i.e. same state in the #bytesaccessed after the parameter# [ptr]) will return the same value
    - Will not modify storage [memory]
    - Returned value depends only on current state of the program
    - Lets compiler reorder calls to pure functions if the program state is not changed between calls [allows for optimization] + improves static checking, knowing memory will not change
    - [[reproducible]] in C23
        - [[unsequenced]] – value independent of program state
- ***attribute((malloc(other_func,1)))*** – prints warnings if data is ever allocated by the function, but never freed [other_func never called]
    - Has limits [static analysis] – if another function is called and takes a pointer [possibly freeing it], compiler may just give up
        - Rust "does better"
- #ifdef – if a compiler does not see a line of code, it may throw warnings that would have been solved by said line
    - [[maybe_unused]] before a function parameter

- Static checking can be employed when going for especially refined programs (as opposed to quick, just functional programs)
    - Static analysis for performance –

**Dynamic Analysis**

- ***Profiling*** - running a program on "typical" input and tracking which parts of the program (e.g. which instructions/branches/functions) are called + how often
    - Can be used to determine hot vs cold functions - can be fed back into the compiler

        - Can also be used to determine areas of the code not covered in test cases
    - GCC coverage options [***--coverage***] - generate profiles, creating a histogram of times run for each instruction address
    - Issues: program runs more slowly when being profiled, "typicality" of input based on developer guesses [atypical -> potentially bad advice], takes time
- Static analysis for correctness
    - const on pointer (const char*, e.g.) – tells compiler program will not store through the pointer [allows for more compiler checks]
        - Caution – make sure annotations match actual intended functionality
- ***Dynamic analysis*** - changing the behavior of a program, to dynamically inspect itself for bugs even while it runs
    - Can detect a wider variety of bugs than static analysis
    - Downsides: incurs runtime performance cost, only detects errors that occur in a particular run [still not 100%]
- ***Runtime checking***
    - Behavior to multiplication overflow undefined for C, C++
        - Can be used for attacks – better to crash than to continue
    - *GCC options (**-fsanitize**):*
        - ***–fsanitize=undefined***: crash on undefined behavior
        - ***–fsanitize=address***: crash on bad pointer references
            - Catches subscript errors, dereferences to freed storage
            - Catches most, not all cases: low-level pointer manipulation, e.g.
            - Mutually exclusive with =undefined [difficult for GCC to do both]
        - ***–fsanitize=leak***: check for memory leaks
            - Faster + more accurate than valgrind, since fsanitize can see the source code
        - ***–fsanitize=thread***: dynamically detect race conditions in multithreaded [parallel] applications
        - ***-fwrapv***: reliable wrap-around on integer overflow (i.e. will always wrap around - no undefined overflow behavior)
            - Used in the Linux kernel for low-level manipulation
            - Side effect: cannot check for integer overflows

- **Portability checking** – the process of checking to ensure that a single program can run on many different systems
  - Can work across different OSes/ISAs, e.g.
  - GCC options:
    - –m32: generates a 32-bit executable [ex: 4-byte pointers]
      - If this fails, likely indicates portability bug (e.g. implicit assumption of 8-byte pointers)
- *Avoiding debugging:*
  - Test cases
    - ***Test-driven development*** – write tests first, then the code
      - "Every bug fix should come with a test case", e.g.
      - No commits without passing every test
    - ***Reliability checking*** [regression testing]
  - ***Defensive programming*** – assume other modules in the program may or may not be buggy, and program extra cautiously [with extra protections] as a result
    - ***Exception handling*** – try/catches, asserts, e.g.
    - Traces and logs – logging program activity in the event of a crash, e.g.
  - ***Barricades*** – separate app into organized, bug-free safe space, potentially dangerous "outside world" (bad data, e.g.)
    - Outside portion rigorously checks passed-in data via runtime checks; only allows approved data into safe space [no checks – more efficient]
  - ***Interpreters*** – programs that run other programs
    - Used in browsers (JavaScript interpreter), e.g.
    - Usually contain many runtime checks
    - Virtual machines – hardware support for hardware-level interpretation
- General debug advice:
  - Don't guess where bugs are; find them
  - Stabilize the failure first (make it reproducible)
    - Upon seeing the symptoms, locate the failure's case
- Debugger contains program execution history, exploration tool
  - Runs program under debugger control
  - Debugger exerts significant control over subsidiary programs/processes
  - Gcc -g3: generates extra info in executable to help debugger (e.g. gdb)

            - Avoid -O2 with -g3 [O2 may reorder code]
                - -Og – optimize for debugging (avoid reordering, e.g.)
            - Gdb: attach *process ID*
- *Gdb commands:*
    - Detach - let program run freely [cede control]
        - Cannot attach multiple gdbs to the same program
    - Exit - exit gdb
        - Lets program run freely [if already existing]
    - Run *arguments* - runs underlying program (with arguments)
        - Effectively suspends gdb until something happens (gdb waits)
    - C-c - suspends program, returning to gdb control
        - Program crashes place program into suspended state, allowing backtrace
    - Bt - gets a backtrace
        - Examines suspended program's state, namely contents of the stack + machine addresses
            - Prints all functions down to main
    - P *address* – prints contents of memory at address
        - Options: p/x = print hexadecimal, e.g.
    - Info registers – print registers
    - P *variable* = *value* – set variable value
        - P $*register* = *value* – set register value
        - P *variable* = *function* – calls function, sets variable value
        - P *function* – calls function
- Gdb - "program execution history explorer"
    - Breakpoints - pre-set locations to automatically suspend a program when encountered
        - Gdb b *address*; then run the program
            - Gdb b *function name*
        - Continue – continue after breakpoint
            - Continue 10 – continue for the next 10 breakpoints
                - Alt: step (single source code line), stepi (single machine instruction), next (like step, but upon function call – keeps running until function returns)

                - In the case of interweaved source code lines in machine instructions, step will stop at every change in source code line
            - Fin – finish program
            - U *address* – execute until the line is reached, or program finishes (sets temporary breakpoint)
        - Reverse continue [rc] – runs program backward until previous breakpoint
            - Requires starting gdb with special  option
        - Watch *variable* – stops program whenever the variable is modified
            - Sets a breakpoint on data, essentially
        - Checkpoint – sets checkpoint at current line (e.g. known “good” location)
            - Restart # – restarts at checkpoint number
        - Gdb actually modifies memory for the duration of the program
            - Breakpoints - temporarily adds an instruction before a breakpoint [in machine code] that crashes program; gdb then re-adds original instruction after crash upon continue
    - Disas – disassembles machine code
- Debugging gdb
    - GDB – call old, unbuggy version of gdb on itself [new, buggy version]

**Makefiles**

- Command `make` used for incremental builds (e.g. recompiling only one file in a project, out of many files)
    - Build configuration specified in Makefiles
        - Use combination of shell, "make" notation
- Makefile syntax:
    - *Make rule*: *target file name*: *included/dependency files* \n *shell commands*
        - Will only rebuild target files if at least one included file [dependency] is newer than the target file
            - Does not automatically build dependencies, if they do not exist; can be written such that `make` automatically builds all dependencies
    - Can declare, use variables (similar to shell variables, but no quotes in declaration; called via $(varname) rather than $varname)
- Command `make`
    - Builds first target, by default [no arguments]; with command-line arguments, can specify the make target (e.g. `make foo.o`)
    - `make clean` removes all target files (can be used for complete rebuild, e.g.)
    - Missing dependencies can cause issues; extra dependencies lead to inefficiencies
        - Dependency loops [circular dependencies] not forbidden (will run one of two ways), but discouraged
    - Flags: -j [build in parallel; -j10 = at most 10 jobs, e.g.]; -j = no limits]
        - make will automatically handle waiting for a target's dependencies to build before building the target
- *Makefiles for larger systems:*
    - ***Approach 1***: centralized Makefile at root, importing subsidiary Makefiles in the various subdirectories [include sub/sub1.mk]
    - ***Approach 2***: Makefile at root that cd's into and runs `make` within subdirectories (based on those subdirectories' individual Makefiles)
        - Primarily used in earlier versions of make

**Backups**

- Motivation: ~100 ZB of data is produced/year by the Internet; ~2/3 of it is copied
  - Devs use git; ops staff use backups
- *Backup strategies:*
  - Use **failure models** to determine what constitutes a failed backup
    - Potential models: hardware failure (e.g. disk drive failure) [partial vs whole-drive failure], accidental (or purposeful/malicious) file deletion/modification
    - Judging risk:
      - TBW denotes manufactured-rated, warranted amounts of safe writes to a flash drive from ordinary wear
      - Disk drives have AFR (annualized failure rate – generally sub-1%)
  - May not need to back up everything: can omit OS installation files, temporary files (e.g. /tmp/ – generally stored in RAM)
    - Do back up: most files, incl. data [file contents], metadata
  - **Break backup into different units**
    - Break data, metadata into discrete byte blocks of a specified size – faster for a backup system to run at a lower level, not need to worry about distinct file boundaries
    - Alt: files as units (simpler to users, but less efficient)
  - **Time units** - how often to backup
    - Compromise between backing up every change, never backing up: back up on important changes/at important times
  - **Garbage collection** - reclaiming storage from unneeded backups
- Can save data via reducing backup frequency, reducing storage of unimportant data, more aggressive garbage collection
  - Can use cheaper, slower drives; remote backup services
  - Automation of cheaper backups:
    - ***Data grooming*** – removal of unnecessary data
    - ***Deduplication*** – looking for duplicates of stored data, and removing them (replaces them with pointers to the original, essentially)

- ***Incremental backup*** – later backups copy only changed data relative to the previous backup (rather than copying all data)
    - Same philosophy as git (inspired git)
    - Ex: if backing up by file, back up only recently changed files
    - Inherently more fragile – losing a single past backup prevents retracing
  - Can **compress** file data for smaller backups
- Other techniques
  - Can **encrypt data** for greater security
  - Can stage **different levels** of copies on different storage devices
- **Backup recovery** – backups are useless unless they can be recovered from
  - Require testing to make sure backups are sufficient for restarting from even in the case of a complete data loss, e.g.
- Strategies for file-system-controlled backups
  - ***Versioning file systems*** – similar to git, but run on a filesystem (see: TENEX)
    - Slightly worse than git, but automatic – no user input needed
    - Keeps copies of older versions of files
      - May be kept to important versions only to save space
      - File version added as a suffix to the file name (lowest number -> most recent backup; no suffix -> latest version)
        - Multiple versions of the file will be seen upon ls -l
        - Saving the file bumps version number
    - Possible downsides: changes to multiple files, system overhead from storing/updating copies
      - Losing a past version makes it difficult to reclaim the rest
      - Hard to understand for users
  - ***Snapshotting file systems*** – taking pictures of the state of the entire file system at single points in time
    - File system states stored as read-only copies of the entire file system tree
    - Snapshotting operation made cheaper via copy on write (CoW) – full copies of files are only performed when the file has been modified
      - File systems modeled via trees of addressed block-structured objects; a new snapshot is just a pointer to a tree

        - Updates to a file create minimal amounts of new objects (typically just copies of old ones) to be created
    - More popular (used on SEASnet, e.g. – creates .snapshot subdirectories)
        - Note: not listed by ls -al (but still cd-into-able)
    - Drawbacks: still susceptible to losing parts of history, incurs overhead

**"Cloud Computing"**

- The name "cloud computing" originated as a joke ("the server is just somewhere off in space – the cloud"); to some extent a marketing term
- [Technical aspects of] ***Cloud computing*** – an on-demand, self-service method for clients to gain access to servers with a low barrier of entry
    - Intended to be rapidly scalable (elastic) for users – clients should be able to spin up servers quickly as needed
        - Broad network access – should be (relatively) location independent
    - Multitenancy – shared resources (server pool) distributed across many tenants
    - Downsides: service providers must keep track of client server usage (incurs overhead)
- Possible implementations:
    - ***Infrastructure-as-a-service*** (**IaaS**) – service provider provides actual physical hardware + networking infrastructure; customer is responsible for actually configuring + running it
        - Customer provides + maintains OS, apps, etc.
        - Provided by AWS, e.g.
    - ***Platform-as-a-service*** (**PaaS**) – service provider provides infrastructure + middleware: preconfigured OS, standard apps (e.g. Python)
        - Customer provides only custom software
        - Creates some hassle for the service provider: might need to tailor software versions to client requests, e.g.
        - Provided by Google App Engine + AWS, e.g.
    - ***Software-as-a-service*** (**SaaS**) – service provider provides platform + entire software stack (i.e. programs to be run)
        - Customer responsible only for configuring the provided software (e.g. tweaking settings); does not need to write any software
        - Provided by Salesforce, e.g.
- **Virtual machine** – "software pretending to be hardware"
    - VMs may emulate other hardware
        - Can be written such that the emulated environment is entirely insulated from the outside environment (provides a barrier)

        - Drawback: Incurs large amounts of overhead (performance)
    - VMs may instead take a hardware-supported approach to emulation: simulated machine/architecture is limited to match that of the actual hardware
        - Allows for utilization of hardware support for most common virtualized instructions (faster than straight emulation)
            - Some commands (e.g. halt) may not be hardware-supported
        - Can be used to emulate programs running on the same ISA/hardware (e.g. Linux atop macOS)
            - Drawback: still has some overhead, relative to running natively
    - ***Containers*** – VMs specialized for virtualizing the same OS as the physical machine
        - Can share hardware instructions + shared OS files (e.g. /lib, /usr/bin)
            - Stores only one copy of kernel, executables
            - Executables stored in read-only memory
        - Containers (notably Docker) – standard approach for most cloud computing platforms, IaaS excepted
            - VMs insulated – client cannot see what others are doing, e.g.
            - Alternative to giving each client a process on the same server, e.g.
- Docker plumbing:
    - Objects:
        - Containers – environments to run applications
        - Images – templates for building containers
        - Services – let containers scale across multiple Docker domains
            - Can coordinate large numbers of containers into a single service
    - Registries – repositories for Docker images
        - Can pull, push images
    - Dockerd[aemon] – Docker daemon responsible for containers
        - Containers implemented as Linux processes controlled by the Docker daemon (running at a root level), essentially
        - Docker shell commands operate through requests to the daemon
- Docker porcelain:
    - Docker maintenance/management programs responsible for load balancing, managing resource constraints, logging/monitoring systems and system errors, security (avoiding break-ins to containers), updates to a running system

        - Updates to a running system: blue/green upgrades (installing updates on a new set of containers separate from the running containers, then switching over when the new version is ready)
        - Programs: Kubernetes, cloud-specific programs (e.g. AWS: Elastic Container Service/ECS)
- Alternate model to containerization: “serverless” computing
    - Serverless computing: writing code ignorant of the server to be run on, then allowing the infrastructure to route transactions to the appropriate server
        - Servers maintained by service provider
        - Allow for simpler code – can ignore server state (down vs up?), server location, e.g.
        - Downside: not knowing which server the code will be run on makes caching data on a single server more difficult (performance downside)

**Software & The Law**

- Law & ethics (law vs ethics)
- Types of law: commercial law (to do with contracts, business disputes), criminal law (crimes, injury), international law (international agreements), admiralty law (ships at sea)
    - Have differences (unanimous consent vs majority opinion in juries, e.g.)
- Commercial law: most software falls under purview of copyright law (protecting creative works), patent law (inventions), trade secrets, trademarks (logos, short descriptions of the product), personal data (modern field), future: AI law
    - Different expirations: copyright (life of author + 75 years), patents (20 years), trade secrets (indefinite)
    - Enforcement: sue [for $$] under commercial law for using intellectual property without permission, e.g.
        - Alternative: technical protections – preventing trade secrets from leaking via simply limiting access to them, e.g.
- License – legal permits [grants of permission] to use/copy/etc. software
    - Free software/open-source software – use licenses to further the goals of the organization, rather than for commercial purposes
    - Approaches:
        - Public domain – no rules on usage; given away (openest approach)
        - Academic license – requires citations of sources [BSD, e.g.]
        - Reciprocal license – similar to academic license; any changes built upon the original must also be redistributed under reciprocal license [GPL, e.g.]
        - Forced contribution – any changes must be given back to, owned by the original authors (Apple, Microsoft, etc.)